CSS Flexible Box Layout Module Level 1

W3C Candidate Recommendation, 19 November 2018

This version:

https://www.w3.org/TR/2018/CR-css-flexbox-1-20181119/

Latest published version:

https://www.w3.org/TR/css-flexbox-1/

Editor's Draft:

https://drafts.csswg.org/css-flexbox-1/

Previous Versions:

https://www.w3.org/TR/2018/CR-css-flexbox-1-20181108/
https://www.w3.org/TR/2017/CR-css-flexbox-1-20171019/
https://www.w3.org/TR/2016/CR-css-flexbox-1-20160526/
https://www.w3.org/TR/2016/CR-css-flexbox-1-20160301/
https://www.w3.org/TR/2015/WD-css-flexbox-1-20150514/
https://www.w3.org/TR/2014/WD-css-flexbox-1-20140925/
https://www.w3.org/TR/2014/WD-css-flexbox-1-20140325/
https://www.w3.org/TR/2012/CR-css3-flexbox-20120918/
https://www.w3.org/TR/2012/WD-css3-flexbox-20120612/
https://www.w3.org/TR/2012/WD-css3-flexbox-20120322/
https://www.w3.org/TR/2011/WD-css3-flexbox-20111129/
https://www.w3.org/TR/2011/WD-css3-flexbox-20110322/
https://www.w3.org/TR/2009/WD-css3-flexbox-20090723/

Test Suite:

http://test.csswg.org/suites/css-flexbox-1_dev/nightly-unstable/

Editors:

Tab Atkins Jr. (Google)
Elika J. Etemad / fantasai (Invited Expert)
Rossen Atanassov (Microsoft)

Former Editors:

Alex Mogilevsky (Microsoft Corporation)
L. David Baron (Mozilla)
Neil Deakin (Mozilla Corporation)
Ian Hickson (formerly of Opera Software)

David Hyatt (formerly of Netscape Corporation)

Suggest an Edit for this Spec:
GitHub Editor

Issue Tracking:
GitHub Issues

WPT Path Prefix:
css/css-flexbox/

Abstract

The specification describes a CSS box model optimized for user interface design. In the flex layout model, the children of a flex container can be laid out in any direction, and can "flex" their sizes, either growing to fill unused space or shrinking to avoid overflowing the parent. Both horizontal and vertical alignment of the children can be easily manipulated. Nesting of these boxes (horizontal inside vertical, or vertical inside horizontal) can be used to build layouts in two dimensions.

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

Status of this document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at https://www.w3.org/TR/.

This document was produced by the CSS Working Group as a Candidate Recommendation. This document is intended to become a W3C Recommendation. This document will remain a Candidate Recommendation at least until 19 December 2018 in order to ensure the opportunity for wide review.

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

A preliminary implementation report is available.

Publication as a Candidate Recommendation does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

This document was produced by a group operating under the 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 February 2018 W3C Process Document.

For changes since the last draft, see the Changes section.

Table of Contents

§ 1. Introduction

This section is not normative.

CSS 2.1 defined four layout modes — algorithms which determine the size and position of boxes based on their relationships with their sibling and ancestor boxes:

- block layout, designed for laying out documents
- inline layout, designed for laying out text
- table layout, designed for laying out 2D data in a tabular format
- positioned layout, designed for very explicit positioning without much regard for other elements in the document

This module introduces a new layout mode, ***flex layout***, which is designed for laying out more complex applications and webpages.

§ 1.1. Overview

This section is not normative.

Flex layout is superficially similar to block layout. It lacks many of the more complex text- or document-centric properties that can be used in block layout, such as floats and columns. In return it gains simple and powerful tools for distributing space and aligning content in ways that web apps and complex web pages often need. The contents of a flex container:

- can be laid out in any flow direction (leftwards, rightwards, downwards, or even upwards!)
- can have their display order 'reversed' or rearranged at the style layer (i.e., visual order can be independent of source and speech order)
- can be laid out linearly along a single (main) axis or wrapped into multiple lines along a secondary (cross) axis
- can "flex" their sizes to respond to the available space
- can be aligned with respect to their container or each other on the secondary (cross)
- can be dynamically collapsed or uncollapsed along the main axis while preserving the container's cross size

EXAMPLE 1

Here's an example of a catalog where each item has a title, a photo, a description, and a purchase button. The designer's intention is that each entry has the same overall size, that the photo be above the text, and that the purchase buttons aligned at the bottom, regardless of the length of the item's description. Flex layout makes many aspects of this design easy:

- The catalog uses flex layout to lay out rows of items horizontally, and to ensure that items within a row are all equal-height. Each entry is then itself a column flex container, laying out its contents vertically.
- Within each entry, the source document content is ordered logically with the title first, followed by the description and the photo. This provides a sensible ordering for speech rendering and in non-CSS browsers. For a more compelling visual presentation, however, 'order' is used to pull the image up from later in the content to the top, and 'align-self' is used to center it horizontally.
- An 'auto' margin above the purchase button forces it to the bottom within each entry box, regardless of the height of that item's description.

```
#deals {
  display: flex;         /* Flex layout so items 'have equal height'  */
  flex-flow: row wrap;   /* Allow items to wrap into multiple lines */
}
.sale-item {
  display: flex;         /* Lay out each item using flex layout */
  flex-flow: column;     /* Lay out item's contents vertically  */
}
.sale-item > img {
  order: -1;             /* Shift image before other content (in visual order) *
  align-self: center;    /* 'Center the image cross-wise (horizontally)'
}
.sale-item > button {
  margin-top: auto;      /* Auto top margin pushes button to bottom */
}
```

```
<section id="deals">
  <section class="sale-item">
    <h1>Computer Starter Kit</h1>
    <p>This is the best computer money can buy, if you don't have much money.
    <ul>
      <li>Computer
```

```
        <li>Monitor
        <li>Keyboard
        <li>Mouse
      </ul>
      <img src="images/computer.jpg"
           alt="You get: a white computer with matching peripherals.">
      <button>BUY NOW</button>
    </section>
    <section class="sale-item">
      …
    </section>
    …
  </section>
```

Computer Starter Kit

This is the best computer money can buy, if you don't have much money.

- Computer
- Monitor
- Keyboard
- Mouse

BUY NOW

Printer

Only capable of printing ASCII art.

- Paper and ink not included.

BUY NOW

Figure 1 *An example rendering of the code above.*

§ 1.2. Module interactions

This module extends the definition of the 'display' property [CSS21], adding a new block-level and new inline-level display type, and defining a new type of formatting context along with properties to control its layout. None of the properties defined in this module apply to the '::first-line' or '::first-letter' pseudo-elements.

The CSS Box Alignment Module extends and supercedes the definitions of the alignment properties ('justify-content', 'align-items', 'align-self', 'align-content') introduced here.

§ 2. Flex Layout Box Model and Terminology

A ***flex container*** is the box generated by an element with a computed 'display' of 'flex' or 'inline-flex'. In-flow children of a flex container are called ***flex items*** and are laid out using the flex layout model.

Unlike block and inline layout, whose layout calculations are biased to the block and inline flow directions, flex layout is biased to the ***flex directions***. To make it easier to talk about flex layout, this section defines a set of flex flow–relative terms. The 'flex-flow' value and the writing mode determine how these terms map to physical directions (top/right/bottom/left), axes (vertical/horizontal), and sizes (width/height).

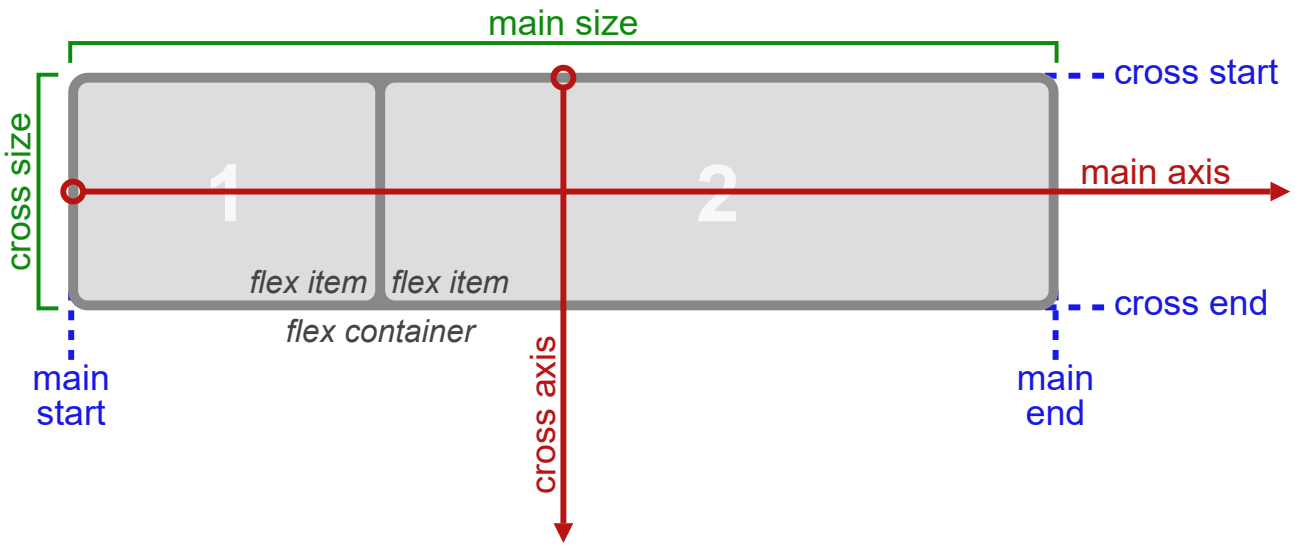


Figure 2 *An illustration of the various directions and sizing terms as applied to a 'row' flex container.*

main axis
main dimension

The ***main axis*** of a flex container is the primary axis along which flex items are laid out. It extends in the ***main dimension***.

main-start
main-end

The flex items are placed within the container starting on the ***main-start*** side and going toward the ***main-end*** side.

main size
main size property

The width or height of a flex container or flex item, whichever is in the main dimension, is that box's ***main size***. Its ***main size property*** is thus either its 'width' or 'height' property, whichever is in the main dimension. Similarly, its ***min*** and ***max main size properties*** are its

'min-width'/'max-width' or 'min-height'/'max-height' properties, whichever is in the main dimension, and determine its ***min*/*max main size***.

cross axis
cross dimension
The axis perpendicular to the main axis is called the ***cross axis***. It extends in the ***cross dimension***.

cross-start
cross-end
Flex lines are filled with items and placed into the container starting on the ***cross-start*** side of the flex container and going toward the ***cross-end*** side.

cross size
cross size property
The width or height of a flex container or flex item, whichever is in the cross dimension, is that box's ***cross size***. Its ***cross size property*** is thus either its 'width' or 'height' property, whichever is in the cross dimension. Similarly, its ***min*** and ***max cross size properties*** are its 'min-width'/'max-width' or 'min-height'/'max-height' properties, whichever is in the cross dimension, and determine its ***min*/*max cross size***.

Additional sizing terminology used in this specification is defined in CSS Intrinsic and Extrinsic Sizing. [CSS-SIZING-3]

§ 3. Flex Containers: the 'flex' and 'inline-flex' 'display' values

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

'flex'
This value causes an element to generate a flex container box that is block-level when placed in flow layout.

'inline-flex'
This value causes an element to generate a flex container box that is inline-level when placed in flow layout.

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

with the margins of its contents. Flex containers form a containing block for their contents exactly like block containers do. [CSS21] The 'overflow' property applies to flex containers.

Flex 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 flex layout. In particular:

- 'float' and 'clear' do not create floating or clearance of flex item, and do not take it out-of-flow.
- 'vertical-align' has no effect on a flex item.
- the '::first-line' and '::first-letter' pseudo-elements do not apply to flex containers, and flex containers do not contribute a first formatted line or first letter to their ancestors.

If an element's specified 'display' is 'inline-flex', then its 'display' property computes to 'flex' in certain circumstances: the table in CSS 2.1 Section 9.7 is amended to contain an additional row, with 'inline-flex' in the "Specified Value" column and 'flex' in the "Computed Value" column.

§ 4. Flex Items

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

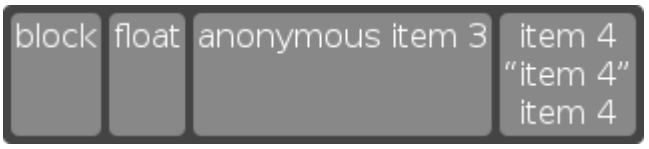
Each in-flow child of a flex container becomes a flex item, and each contiguous sequence of child text runs is wrapped in an anonymous block container flex item. However, if the entire sequence of child text runs contains only white space (i.e. characters that can be affected by the 'white-space' property)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

EXAMPLE 2

Examples of flex items:

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

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

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

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

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

Figure 3 *Flex items determined from above code block*



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

Note also that the anonymous item’s box is unstyleable, since there is no element to assign style rules to. Its contents will however inherit styles (such as font settings) from the flex container.

A flex item establishes an independent formatting context for its contents. However, flex items themselves are ***flex-level*** boxes, not block-level boxes: they participate in their container’s flex formatting context, not in a block formatting context.

✧ ✧ ✧

> Note: Authors reading this spec may want to skip past the following box-generation and static position details.

The ‘display’ value of a flex item is blockified: if the specified ‘display’ of an in-flow child of an element generating a flex 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 flex item, it is blockified first, and so anonymous box creation will not happen. For example, two contiguous flex items with ‘display: table-cell’ will become two separate ‘display: block’ flex items, instead of being wrapped into a single anonymous table.

In the case of flex items with ‘display: table’, the table wrapper box becomes the flex item, and the ‘order’ and ‘align-self’ properties apply to it. The contents of any caption boxes contribute to the calculation of the table wrapper box’s min-content and max-content sizes. However, like ‘width’ and ‘height’, the ‘flex’ longhands apply to the table box as follows: the flex item’s final size is calculated by performing layout as if the distance between the table wrapper box’s edges and the table box’s content edges were all part of the table box’s border+padding area, and the table box were the flex item.

§ 4.1. Absolutely-Positioned Flex Children

As it is out-of-flow, an absolutely-positioned child of a flex container does not participate in flex layout.

The static position of an absolutely-positioned child of a flex container is determined such that the child is positioned as if it were the sole flex item in the flex container, assuming both the child and the flex container were fixed-size boxes of their used size. For this purpose, ‘auto’ margins are treated as zero.

In other words, the static-position rectangle of an absolutely-positioned child of a flex container is the flex container's content box, where the ***static-position rectangle*** is the alignment container used to determine the static-position offsets of an absolutely-positioned box.

(In block layout the static position rectangle corresponds to the position of the "hypothetical box" described in CSS2.1§10.3.7. Since it has no alignment properties, CSS2.1 always uses a block-start inline-start alignment of the absolutely-positioned box within the static-position rectangle. Note that this definition will eventually move to the CSS Positioning module.)

EXAMPLE 3

The effect of this is that if you set, for example, 'align-self: center;' on an absolutely-positioned child of a flex container, auto offsets on the child will center it in the flex container's cross axis.

However, since the absolutely-positioned box is considered to be "fixed-size", a value of 'stretch' is treated the same as 'flex-start'.

§ 4.2. Flex Item Margins and Paddings

The margins of adjacent flex items do not collapse.

Percentage margins and paddings on flex items, like those on block boxes, are resolved against the inline size of their containing block, e.g. left/right/top/bottom percentages all resolve against their containing block's *width* in horizontal writing modes.

Auto margins expand to absorb extra space in the corresponding dimension. They can be used for alignment, or to push adjacent flex items apart. See Aligning with 'auto' margins.

§ 4.3. Flex Item Z-Ordering

Flex items paint 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').

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

§ 4.4. Collapsed Items

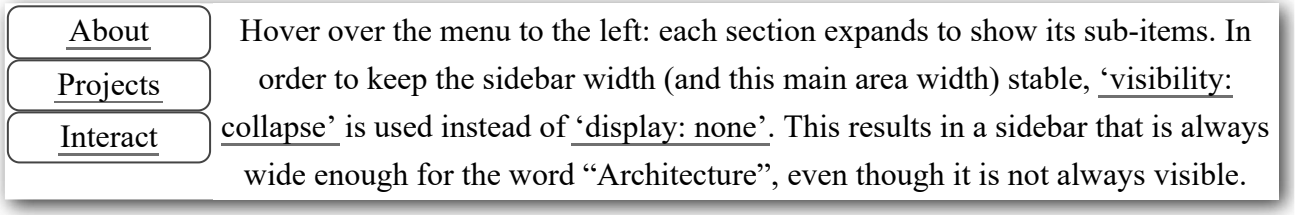
Specifying ‘visibility:collapse’ on a flex item causes it to become a ***collapsed flex item***, producing an effect similar to ‘visibility:collapse’ on a table-row or table-column: the collapsed flex item is removed from rendering entirely, but leaves behind a "strut" that keeps the flex line’s cross-size stable. Thus, if a flex container has only one flex line, dynamically collapsing or uncollapsing items may change the flex container’s main size, but is guaranteed to have no effect on its cross size and won’t cause the rest of the page’s layout to "wobble". Flex line wrapping *is* re-done after collapsing, however, so the cross-size of a flex container with multiple lines might or might not change.

Though collapsed flex items aren’t rendered, they do appear in the formatting structure. Therefore, unlike on ‘display:none’ items [CSS21], effects that depend on a box appearing in the formatting structure (like incrementing counters or running animations and transitions) still operate on collapsed items.

EXAMPLE 4

In the following example, a sidebar is sized to fit its content. 'visibility: collapse' is used to dynamically hide parts of a navigation sidebar without affecting its width, even though the widest item ("Architecture") is in a collapsed section.

Figure 4 *Sample live rendering for example code below*




```
@media (min-width: 60em) {
  /* two column layout only when enough room (relative to default text size) */
  div { display: flex; }
  #main {
    flex: 1;         /* Main takes up all remaining space */
    order: 1;        /* Place it after (to the right of) the navigation */
    min-width: 12em; /* Optimize main content area sizing */
  }
}
/* menu items use flex layout so that visibility:collapse will work */
nav > ul > li {
  display: flex;
  flex-flow: column;
}
/* dynamically collapse submenus when not targetted */
nav > ul > li:not(:target):not(:hover) > ul {
  visibility: collapse;
}
```

```
<div>
  <article id="main">
    Interesting Stuff to Read
  </article>
  <nav>
    <ul>
      <li id="nav-about"><a href="#nav-about">About</a>
        …
      <li id="nav-projects"><a href="#nav-projects">Projects</a>
```

```
          <ul>
            <li><a href="…">Art</a>
            <li><a href="…">Architecture</a>
            <li><a href="…">Music</a>
          </ul>
        <li id="nav-interact"><a href="#nav-interact">Interact</a>
          …
      </ul>
    </nav>
  </div>
  <footer>
  …
```

To compute the size of the strut, flex layout is first performed with all items uncollapsed, and then re-run with each collapsed item replaced by a strut that maintains the original cross-size of the item's original line. See the Flex Layout Algorithm for the normative definition of how 'visibility:collapse' interacts with flex layout.

> Note: Using 'visibility:collapse' on any flex items will cause the flex layout algorithm to repeat partway through, re-running the most expensive steps. It's recommended that authors continue to use 'display:none' to hide items if the items will not be dynamically collapsed and uncollapsed, as that is more efficient for the layout engine. (Since only part of the steps need to be repeated when 'visibility' is changed, however, 'visibility: collapse' is still recommended for dynamic cases.)

§ 4.5. Automatic Minimum Size of Flex Items

> Note: The 'auto' keyword, representing an automatic minimum size, is the new initial value of the 'min-width' and 'min-height' properties. The keyword was previously defined in this specification, but is now defined in the CSS Sizing module.

To provide a more reasonable default minimum size for flex items, the used value of a main axis automatic minimum size on a flex item that is not a scroll container is a ***content-based minimum size***; for scroll containers the automatic minimum size is zero, as usual.

In general, the content-based minimum size of a flex item is the smaller of its content size suggestion and its specified size suggestion. However, if the box has an aspect ratio and no specified size, its content-based minimum size is the smaller of its content size suggestion and its transferred size

suggestion. If the box has neither a specified size suggestion nor an aspect ratio, its content-based minimum size is the content size suggestion.

The content size suggestion, specified size suggestion, and transferred size suggestion used in this calculation account for the relevant min/max/preferred size properties so that the content-based minimum size does not interfere with any author-provided constraints, and are defined below:

specified size suggestion
: If the item's computed main size property is definite, then the specified size suggestion is that size (clamped by its max main size property if it's definite). It is otherwise undefined.

transferred size suggestion
: If the item has an intrinsic aspect ratio and its computed cross size property is definite, then the transferred size suggestion is that size (clamped by its min and max cross size properties if they are definite), converted through the aspect ratio. It is otherwise undefined.

content size suggestion
: The content size suggestion is the min-content size in the main axis, clamped, if it has an aspect ratio, by any definite min and max cross size properties converted through the aspect ratio, and then further clamped by the max main size property if that is definite.

For the purpose of calculating an intrinsic size of the box (e.g. the box's min-content size), a content-based minimum size causes the box's size in that axis to become indefinite (even if e.g. its 'width' property specifies a definite size). Note this means that percentages calculated against this size will behave as auto.

Nonetheless, although this may require an additional layout pass to re-resolve percentages in some cases, this value (like the 'min-content', 'max-content', and 'fit-content' values defined in [CSS-SIZING-3]) does not prevent the resolution of percentage sizes within the item.

> 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 flex sizing 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 into an overflow zone, and thereby making lines of text gratuitously 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.)

§ 5. Ordering and Orientation

The contents of a flex container can be laid out in any direction and in any order. This allows an author to trivially achieve effects that would previously have required complex or fragile methods, such as hacks using the 'float' and 'clear' properties. This functionality is exposed through the 'flex-direction', 'flex-wrap', and 'order' properties.

Note: The reordering capabilities of flex 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 for non-CSS UAs and for linear models such as speech and sequential navigation. See Reordering and Accessibility and the Flex Layout Overview for examples that use this dichotomy to improve accessibility.

Authors *must not* use 'order' or the '*-reverse' values of 'flex-flow'/'flex-direction' as a substitute for correct source ordering, as that can ruin the accessibility of the document.

§ 5.1. Flex Flow Direction: the 'flex-direction' property

Name:	***'flex-direction'***
Value:	row | row-reverse | column | column-reverse
Initial:	row
Applies to:	flex containers
Inherited:	no
Percentages:	n/a
Computed value:	specified keyword
Canonical order:	per grammar
Animation type:	discrete

The ‘flex-direction’ property specifies how flex items are placed in the flex container, by setting the direction of the flex container’s main axis. This determines the direction in which flex items are laid out.

‘row’

The flex container’s main axis has the same orientation as the inline axis of the current writing mode. The main-start and main-end directions are equivalent to the inline-start and inline-end directions, respectively, of the current writing mode.

‘row-reverse’

Same as ‘row’, except the main-start and main-end directions are swapped.

‘column’

The flex container’s main axis has the same orientation as the block axis of the current writing mode. The main-start and main-end directions are equivalent to the block-start and block-end directions, respectively, of the current writing mode.

‘column-reverse’

Same as ‘column’, except the main-start and main-end directions are swapped.

> Note: The reverse values do not reverse box ordering: like ‘writing-mode’ and ‘direction’ [CSS3-WRITING-MODES], they only change the direction of flow. Painting order, speech order, and sequential navigation orders are not affected.

§ 5.2. Flex Line Wrapping: the ‘flex-wrap’ property

Name:	***'flex-wrap'***
Value:	nowrap | wrap | wrap-reverse
Initial:	nowrap
Applies to:	flex containers
Inherited:	no
Percentages:	n/a
Computed value:	specified keyword
Canonical order:	per grammar
Animation type:	discrete

The 'flex-wrap' property controls whether the flex container is single-line or multi-line, and the direction of the cross-axis, which determines the direction new lines are stacked in.

'nowrap'
: The flex container is single-line.

'wrap'
: The flex container is multi-line.

'wrap-reverse'
: Same as 'wrap'.

For the values that are not 'wrap-reverse', the cross-start direction is equivalent to either the inline-start or block-start direction of the current writing mode (whichever is in the cross axis) and the cross-end direction is the opposite direction of cross-start. When 'flex-wrap' is 'wrap-reverse', the cross-start and cross-end directions are swapped.

§ 5.3. Flex Direction and Wrap: the 'flex-flow' shorthand

Name:	***'flex-flow'***
Value:	<'flex-direction'> || <'flex-wrap'>
Initial:	see individual properties
Applies to:	see individual properties
Inherited:	see individual properties
Percentages:	see individual properties
Computed value:	see individual properties
Animation type:	see individual properties
Canonical order:	per grammar

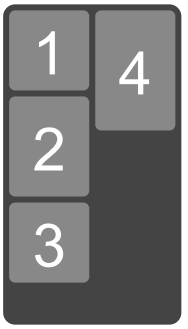
The 'flex-flow' property is a shorthand for setting the 'flex-direction' and 'flex-wrap' properties, which together define the flex container's main and cross axes.

EXAMPLE 5

Some examples of valid flows in an English (left-to-right, horizontal writing mode) document:

```
div { flex-flow: row; }
/* Initial value. Main-axis is inline, no wrapping.
   (Items will either shrink to fit or overflow.) */
```

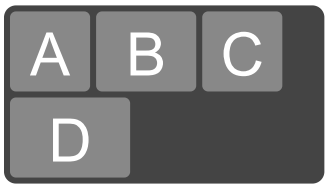

```
div { flex-flow: column wrap; }
/* Main-axis is block-direction (top to bottom)
   and lines wrap in the inline direction (rightwards). */
```

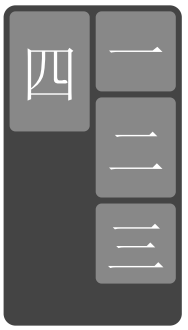



```
div { flex-flow: row-reverse wrap-reverse; }
/* Main-axis is the opposite of inline direction
   (right to left). New lines wrap upwards. */
```

Note that the ‘flex-flow’ directions are writing mode sensitive. In vertical Japanese, for example, a ‘row’ flex container lays out its contents from top to bottom, as seen in this example:

English	**Japanese**
`flex-flow: row wrap;` `writing-mode: horizontal-tb;`	`flex-flow: row wrap;` `writing-mode: vertical-rl;`





§ 5.4. Display Order: the 'order' property

Flex items are, by default, displayed and laid out in the same order as they appear in the source document. The 'order' property can be used to change this ordering.

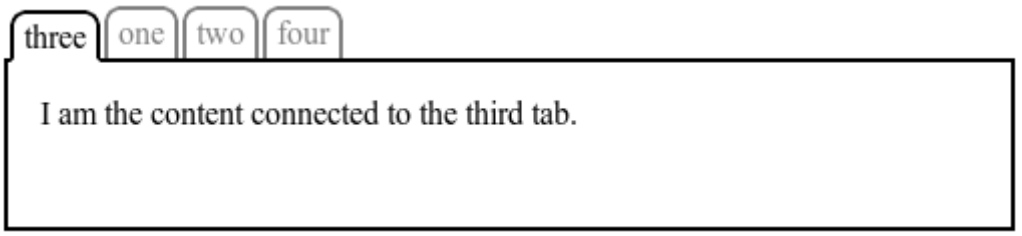
Name:	***'order'***
Value:	<integer>
Initial:	0
Applies to:	flex items
Inherited:	no
Percentages:	n/a
Computed value:	specified integer
Canonical order:	per grammar
Animation type:	by computed value type

The 'order' property controls the order in which flex items appear within the flex container, by assigning them to ordinal groups. It takes a single ***'<integer>'*** value, which specifies which ordinal group the flex item belongs to.

A flex container lays out its content in ***order-modified document order***, starting from the lowest numbered ordinal group and going up. Items with the same ordinal group are laid out in the order they appear in the source document. This also affects the painting order [CSS21], exactly as if the flex items were reordered in the source document. Absolutely-positioned children of a flex container are treated as having 'order: 0' for the purpose of determining their painting order relative to flex items.

EXAMPLE 6

The following figure shows a simple tabbed interface, where the tab for the active pane is always first:



This could be implemented with the following CSS (showing only the relevant code):

```
.tabs {
  display: flex;
}
.tabs > .current {
  order: -1; /* Lower than the default of 0 */
}
```

Unless otherwise specified by a future specification, this property has no effect on boxes that are not flex items.

§ 5.4.1. Reordering and Accessibility

The 'order' property *does not* affect ordering in non-visual media (such as speech). Likewise, 'order' does not affect the default traversal order of sequential navigation modes (such as cycling through links, see e.g. `tabindex` [HTML]).

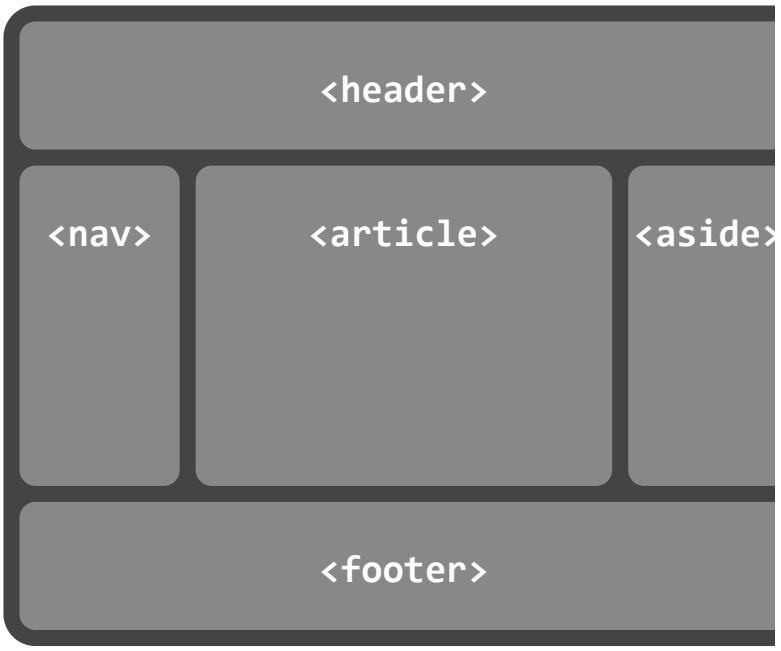
Authors *must* use 'order' only for visual, not logical, reordering of content. Style sheets that use 'order' to perform logical reordering are non-conforming.

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 'order' is used to tailor the visual order. (Since visual perception is two-dimensional and non-linear, the desired visual order is not always logical.)

EXAMPLE 7

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. 'order' makes this trivial. For example, take the following sketch of a page's code and desired layout:

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



This layout can be easily achieved with flex layout:

```
main { display: flex; }
main > article { order: 2; min-width: 12em; flex:1; }
main > nav     { order: 1; width: 200px; }
main > aside   { order: 3; width: 200px; }
```

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: 600px) {
  /* Too narrow to support three columns */
  main { flex-flow: column; }
  main > article, main > nav, main > aside {
    /* Return them to document order */
    order: 0; width: auto;
  }
}
```

(Further use of multi-line flex containers to achieve even more intelligent wrapping left as an exercise for the reader.)

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' 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 8

For example, a tool might offer both drag-and-drop reordering of flex 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 perform drag-and-drop reordering at the DOM layer. In some cases, however, the author may want different visual orderings per screen size. The tool could offer this functionality by using 'order' together with media queries, but also tie the smallest screen size's ordering to the underlying DOM order (since this is most likely to be a logical linear presentation order) while using 'order' to determine the visual presentation order in other size ranges.

This tool would be conformant, whereas a tool that only ever used 'order' to handle drag-and-drop reordering (however convenient it might be to implement it that way) would be non-conformant.

> Note: User agents, including browsers, accessible technology, and extensions, may offer spatial navigation features. This section does not preclude respecting the 'order' property when determining element ordering in such spatial navigation modes; indeed it would need to be considered for such a feature to work. But 'order' is not the only (or even the primary) CSS property that would need to be considered for such a spatial navigation feature. A well-implemented spatial navigation feature would need to consider all the layout features of CSS that modify spatial relationships.

§ 6. Flex Lines

Flex items in a flex container are laid out and aligned within ***flex lines***, hypothetical containers used for grouping and alignment by the layout algorithm. A flex container can be either single-line or multi-line, depending on the 'flex-wrap' property:

- A ***single-line flex container*** (i.e. one with 'flex-wrap: nowrap') lays out all of its children in a single line, even if that would cause its contents to overflow.
- A ***multi-line flex container*** (i.e. one with 'flex-wrap: wrap' or 'flex-wrap: wrap-reverse') breaks its flex items across multiple lines, similar to how text is broken onto a new line when it gets too wide to fit on the existing line. When additional lines are created, they are stacked in the flex container along the cross axis according to the 'flex-wrap' property. Every line contains at least one flex item, unless the flex container itself is completely empty.

EXAMPLE 9

This example shows four buttons that do not fit side-by-side horizontally, and therefore will wrap into multiple lines.

```
#flex {
  display: flex;
  flex-flow: row wrap;
  width: 300px;
}
.item {
  width: 80px;
}


<div id="flex">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
  <div class="item">4</div>
</div>
```

Since the container is 300px wide, only three of the items fit onto a single line. They take up 240px, with 60px left over of remaining space. Because the 'flex-flow' property specifies a multi-line flex container (due to the 'wrap' keyword appearing in its value), the flex container will create an additional line to contain the last item.

Figure 5 *An example rendering of the multi-line flex container.*

Once content is broken into lines, each line is laid out independently; flexible lengths and the 'justify-content' and 'align-self' properties only consider the items on a single line at a time.

In a multi-line flex container (even one with only a single line), the cross size of each line is the minimum size necessary to contain the flex items on the line (after alignment due to 'align-self'), and the lines are aligned within the flex container with the 'align-content' property. In a single-line flex

container, the cross size of the line is the cross size of the flex container, and ‘align-content’ has no effect. The main size of a line is always the same as the main size of the flex container’s content box.

EXAMPLE 10

Here’s the same example as the previous, except that the flex items have all been given ‘flex: auto’. The first line has 60px of remaining space, and all of the items have the same flexibility, so each of the three items on that line will receive 20px of extra width, each ending up 100px wide. The remaining item is on a line of its own and will stretch to the entire width of the line, i.e. 300px.

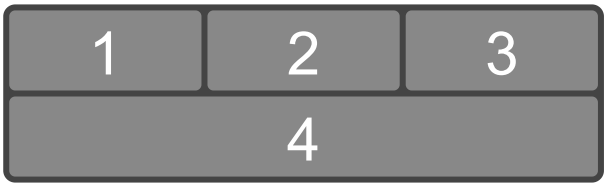


Figure 6 *A rendering of the same as above, but with the items all given ‘flex: auto’.*

§ 7. Flexibility

The defining aspect of flex layout is the ability to make the flex items “flex”, altering their width/height to fill the available space in the main dimension. This is done with the ‘flex’ property. A flex container distributes free space to its items (proportional to their flex grow factor) to fill the container, or shrinks them (proportional to their flex shrink factor) to prevent overflow.

A flex item is ***fully inflexible*** if both its ‘flex-grow’ and ‘flex-shrink’ values are zero, and ***flexible*** otherwise.

§ 7.1. The ‘flex’ Shorthand

Name:	***'flex'***
Value:	none | [<'flex-grow'> <'flex-shrink'>? || <'flex-basis'>]
Initial:	0 1 auto
Applies to:	flex items
Inherited:	no
Percentages:	see individual properties
Computed value:	see individual properties
Animation type:	by computed value type
Canonical order:	per grammar

The 'flex' property specifies the components of a ***flexible length***: the ***flex factors*** (grow and shrink) and the flex basis. When a box is a flex item, 'flex' is consulted *instead of* the main size property to determine the main size of the box. If a box is not a flex item, 'flex' has no effect.

'<'flex-grow'>'

This <number> component sets 'flex-grow' longhand and specifies the ***flex grow factor***, which determines how much the flex item will grow relative to the rest of the flex items in the flex container when positive free space is distributed. When omitted, it is set to '1'.

▶ **Flex values between 0 and 1 have a somewhat special behavior: when the sum of the flex values on the line is less than 1, they will take up less than 100% of the free space.**

'<'flex-shrink'>'

This <number> component sets 'flex-shrink' longhand and specifies the ***flex shrink factor***, which determines how much the flex item will shrink relative to the rest of the flex items in the flex container when negative free space is distributed. When omitted, it is set to '1'.

> Note: The flex shrink factor is multiplied by the flex base size when distributing negative space. This distributes negative space in proportion to how much the item is able to shrink, so that e.g. a small item won't shrink to zero before a larger item has been noticeably reduced.

'<'flex-basis'>'

This component sets the 'flex-basis' longhand, which specifies the ***flex basis***: the initial main size of the flex item, before free space is distributed according to the flex factors.

<'flex-basis'> accepts the same values as the 'width' and 'height' properties (except that 'auto' is treated differently) plus the 'content' keyword:

'auto'

When specified on a flex item, the 'auto' keyword retrieves the value of the main size property as the used 'flex-basis'. If that value is itself 'auto', then the used value is 'content'.

'content'

Indicates an automatic size based on the flex item's content. (It is typically equivalent to the max-content size, but with adjustments to handle aspect ratios, intrinsic sizing constraints, and orthogonal flows; see details in §9 Flex Layout Algorithm.)

> Note: This value was not present in the initial release of Flexible Box Layout, and thus some older implementations will not support it. The equivalent effect can be achieved by using 'auto' together with a main size ('width' or 'height') of 'auto'.

<'width'>

For all other values, 'flex-basis' is resolved the same way as for 'width' and 'height'.

When omitted from the 'flex' shorthand, its specified value is '0'.

'none'

The keyword 'none' expands to '0 0 auto'.

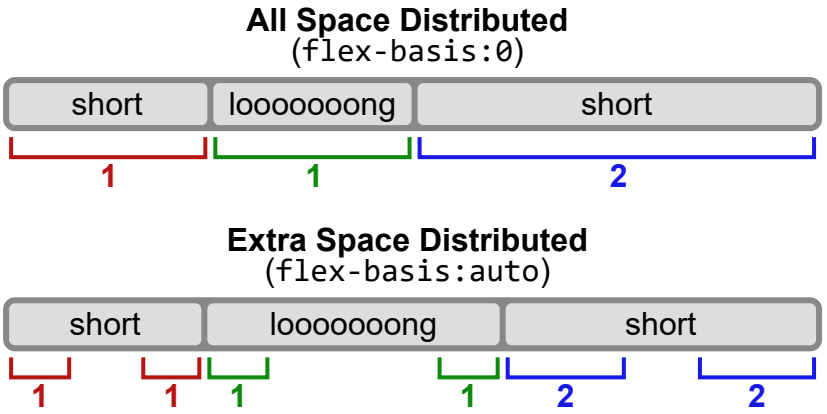


Figure 7 *A diagram showing the difference between "absolute" flex (starting from a basis of zero) and "relative" flex (starting from a basis of the item's content size). The three items have flex factors of '1', '1', and '2', respectively: notice that the item with a flex factor of '2' grows twice as fast as the others.*

The initial values of the 'flex' components are equivalent to flex: 0 1 auto.

> Note: The initial values of 'flex-grow' and 'flex-basis' are different from their defaults when omitted in the 'flex' shorthand. This is so that the 'flex' shorthand can better accommodate the most common cases.

A unitless zero that is not already preceded by two flex factors must be interpreted as a flex factor. To avoid misinterpretation or invalid declarations, authors must specify a zero <'flex-basis'> component with a unit or precede it by two flex factors.

§ 7.1.1. Basic Values of 'flex'

This section is informative.

The list below summarizes the effects of the four 'flex' values that represent most commonly-desired effects:

¶ **'flex: initial'**

Equivalent to 'flex: 0 1 auto'. (This is the initial value.) Sizes the item based on the 'width'/'height' properties. (If the item's main size property computes to 'auto', this will size the flex item based on its contents.) Makes the flex item inflexible when there is positive free space, but allows it to shrink to its minimum size when there is insufficient space. The alignment abilities or 'auto' margins can be used to align flex items along the main axis.

'flex: auto'

Equivalent to ‘flex: 1 1 auto’. Sizes the item based on the ‘width’/‘height’ properties, but makes them fully flexible, so that they absorb any free space along the main axis. If all items are either ‘flex: auto’, ‘flex: initial’, or ‘flex: none’, any positive free space after the items have been sized will be distributed evenly to the items with ‘flex: auto’.

‘flex: none’

Equivalent to ‘flex: 0 0 auto’. This value sizes the item according to the ‘width’/‘height’ properties, but makes the flex item fully inflexible. This is similar to ‘initial’, except that flex items are not allowed to shrink, even in overflow situations.

‘flex: <positive-number>’

Equivalent to ‘flex: <positive-number> 1 0’. Makes the flex item flexible and sets the flex basis to zero, resulting in an item that receives the specified proportion of the free space in the flex container. If all items in the flex container use this pattern, their sizes will be proportional to the specified flex factor.

By default, flex items won’t shrink below their minimum content size (the length of the longest word or fixed-size element). To change this, set the ‘min-width’ or ‘min-height’ property. (See §4.5 Automatic Minimum Size of Flex Items.)

§ 7.2. Components of Flexibility

Individual components of flexibility can be controlled by independent longhand properties.

> **Authors are encouraged to control flexibility using the ‘flex’ shorthand rather than with its longhand properties directly, as the shorthand correctly resets any unspecified components to accommodate common uses.**

§ 7.2.1. The ‘flex-grow’ property

Name:	***'flex-grow'***
Value:	<number>
Initial:	0
Applies to:	flex items
Inherited:	no
Percentages:	n/a
Computed value:	specified number
Canonical order:	per grammar
Animation type:	by computed value type

Authors are encouraged to control flexibility using the 'flex' shorthand rather than with 'flex-grow' directly, as the shorthand correctly resets any unspecified components to accommodate common uses.

The 'flex-grow' property sets the flex grow factor to the provided ***'<number>'***. Negative numbers are invalid.

§ 7.2.2. The 'flex-shrink' property

Name:	***'flex-shrink'***
Value:	<number>
Initial:	1
Applies to:	flex items
Inherited:	no
Percentages:	n/a
Computed value:	specified value
Canonical order:	per grammar
Animation type:	number

Authors are encouraged to control flexibility using the 'flex' shorthand rather than with 'flex-shrink' directly, as the shorthand correctly resets any unspecified components to accommodate common uses.

The 'flex-shrink' property sets the flex shrink factor to the provided ***'<number>'***. Negative numbers are invalid.

§ 7.2.3. The 'flex-basis' property

Name:	***'flex-basis'***
Value:	content | <'width'>
Initial:	auto
Applies to:	flex items
Inherited:	no
Percentages:	relative to the flex container's inner main size
Computed value:	specified keyword or a computed <length-percentage> value
Canonical order:	per grammar
Animation type:	by computed value type

Authors are encouraged to control flexibility using the 'flex' shorthand rather than with 'flex-basis' directly, as the shorthand correctly resets any unspecified components to accommodate common uses.

The 'flex-basis' property sets the flex basis. It accepts the same values as the 'width' and 'height' property, plus 'content'.

For all values other than 'auto' and 'content' (defined above), 'flex-basis' is resolved the same way as 'width' in horizontal writing modes [CSS21], except that if a value would resolve to 'auto' for 'width', it instead resolves to 'content' for 'flex-basis'. For example, percentage values of 'flex-basis' are resolved against the flex item's containing block (i.e. its flex container); and if that containing block's size is indefinite, the used value for 'flex-basis' is 'content'. As another corollary, 'flex-basis' determines the size of the content box, unless otherwise specified such as by 'box-sizing' [CSS3UI].

§ 8. Alignment

After a flex container's contents have finished their flexing and the dimensions of all flex items are finalized, they can then be aligned within the flex container.

The ‘margin’ properties can be used to align items in a manner similar to, but more powerful than, what margins can do in block layout. Flex items also respect the alignment properties from CSS Box Alignment, which allow easy keyword-based alignment of items in both the main axis and cross axis. These properties make many common types of alignment trivial, including some things that were very difficult in CSS 2.1, like horizontal and vertical centering.

> Note: While the alignment properties are defined in CSS Box Alignment [CSS-ALIGN-3], Flexible Box Layout reproduces the definitions of the relevant ones here so as to not create a normative dependency that may slow down advancement of the spec. These properties apply only to flex layout until CSS Box Alignment Level 3 is finished and defines their effect for other layout modes. Additionally, any new values defined in the Box Alignment module will apply to Flexible Box Layout; in otherwords, the Box Alignment module, once completed, will supercede the definitions here.

§ 8.1. Aligning with ‘auto’ margins

This section is non-normative. The normative definition of how margins affect flex items is in the Flex Layout Algorithm section.

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

- During calculations of flex bases and flexible lengths, auto margins are treated as ‘0’.
- Prior to alignment via ‘justify-content’ and ‘align-self’, any positive free space is distributed to auto margins in that dimension.
- Overflowing boxes ignore their auto margins and overflow in the end direction.

> Note: If free space is distributed to auto margins, the alignment properties will have no effect in that dimension because the margins will have stolen all the free space left over after flexing.

EXAMPLE 11

One use of ‘auto’ margins in the main axis is to separate flex items into distinct "groups". The following example shows how to use this to reproduce a common UI pattern - a single bar of actions with some aligned on the left and others aligned on the right.

Figure 8 *Sample rendering of the code below.*


```
nav > ul {
  display: flex;
}
nav > ul > #login {
  margin-left: auto;
}
```

```
<nav>
  <ul>
    <li><a href=/about>About</a>
    <li><a href=/projects>Projects</a>
    <li><a href=/interact>Interact</a>
    <li id="login"><a href=/login>Login</a>
  </ul>
</nav>
```

EXAMPLE 12

The figure below illustrates the difference in cross-axis alignment in overflow situations between using ‘auto’ margins and using the ‘alignment properties’.

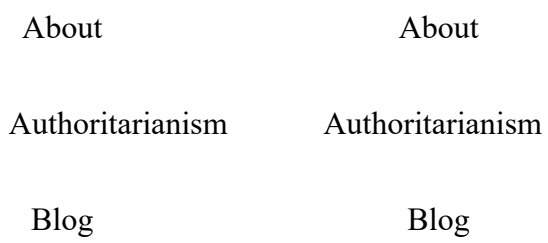


Figure 9 *The items in the figure on the left are centered with margins, while those in the figure on the right are centered with ‘align-self’. If this column flex container was placed against the left edge of the page, the margin behavior would be more desirable, as the long item would be fully readable. In other circumstances, the true centering behavior might be better.*

§ 8.2. Axis Alignment: the ‘justify-content’ property

Name:	***'justify-content'***
Value:	flex-start | flex-end | center | space-between | space-around
Initial:	flex-start
Applies to:	flex containers
Inherited:	no
Percentages:	n/a
Computed value:	specified keyword
Canonical order:	per grammar
Animation type:	discrete

The 'justify-content' property aligns flex items along the main axis of the current line of the flex container. This is done *after* any flexible lengths and any auto margins have been resolved. Typically it helps distribute extra free space leftover when either all the flex items on a line are inflexible, or are flexible but have reached their maximum size. It also exerts some control over the alignment of items when they overflow the line.

'flex-start'

Flex items are packed toward the start of the line. The main-start margin edge of the first flex item on the line is placed flush with the main-start edge of the line, and each subsequent flex item is placed flush with the preceding item.

'flex-end'

Flex items are packed toward the end of the line. The main-end margin edge of the last flex item is placed flush with the main-end edge of the line, and each preceding flex item is placed flush with the subsequent item.

'center'

Flex items are packed toward the center of the line. The flex items on the line are placed flush with each other and aligned in the center of the line, with equal amounts of space between the main-start edge of the line and the first item on the line and between the main-end edge of the line

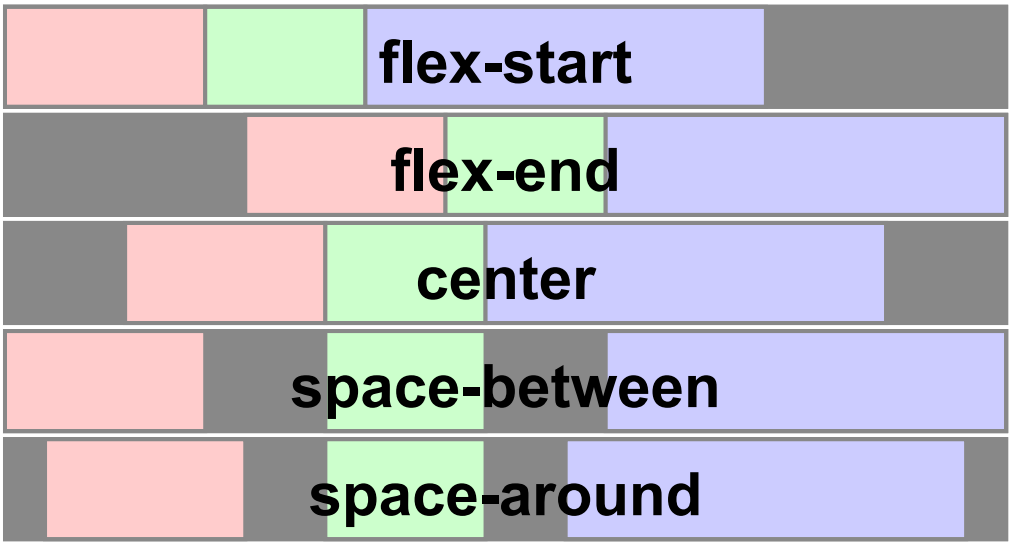
and the last item on the line. (If the leftover free-space is negative, the flex items will overflow equally in both directions.)

'space-between'

Flex items are evenly distributed in the line. If the leftover free-space is negative or there is only a single flex item on the line, this value is identical to 'flex-start'. Otherwise, the main-start margin edge of the first flex item on the line is placed flush with the main-start edge of the line, the main-end margin edge of the last flex item on the line is placed flush with the main-end edge of the line, and the remaining flex items on the line are distributed so that the spacing between any two adjacent items is the same.

'space-around'

Flex items are evenly distributed in the line, with half-size spaces on either end. If the leftover free-space is negative or there is only a single flex item on the line, this value is identical to 'center'. Otherwise, the flex items on the line are distributed such that the spacing between any two adjacent flex items on the line is the same, and the spacing between the first/last flex items and the flex container edges is half the size of the spacing between flex items.



An illustration of the five 'justify-content' keywords and their effects on a flex container with three colored items.

§ 8.3. Cross-axis Alignment: the 'align-items' and 'align-self' properties

Name:	***'align-items'***
Value:	flex-start | flex-end | center | baseline | stretch
Initial:	stretch
Applies to:	flex containers
Inherited:	no
Percentages:	n/a
Computed value:	specified keyword
Canonical order:	per grammar
Animation type:	discrete

Name:	***'align-self'***
Value:	auto | flex-start | flex-end | center | baseline | stretch
Initial:	auto
Applies to:	flex items
Inherited:	no
Percentages:	n/a
Computed value:	specified keyword
Canonical order:	per grammar
Animation type:	discrete

Flex items can be aligned in the cross axis of the current line of the flex container, similar to 'justify-content' but in the perpendicular direction. 'align-items' sets the default alignment for all of the flex container's items, including anonymous flex items. 'align-self' allows this default alignment to be overridden for individual flex items. (For anonymous flex items, 'align-self' always matches the value of 'align-items' on their associated flex container.)

If either of the flex item's cross-axis margins are 'auto', 'align-self' has no effect.

Values have the following meanings:

'auto'

Defers cross-axis alignment control to the value of 'align-items' on the parent box. (This is the initial value of 'align-self'.)

'flex-start'

The cross-start margin edge of the flex item is placed flush with the cross-start edge of the line.

'flex-end'

The cross-end margin edge of the flex item is placed flush with the cross-end edge of the line.

'center'

The flex item's margin box is centered in the cross axis within the line. (If the cross size of the flex line is less than that of the flex item, it will overflow equally in both directions.)

'baseline'

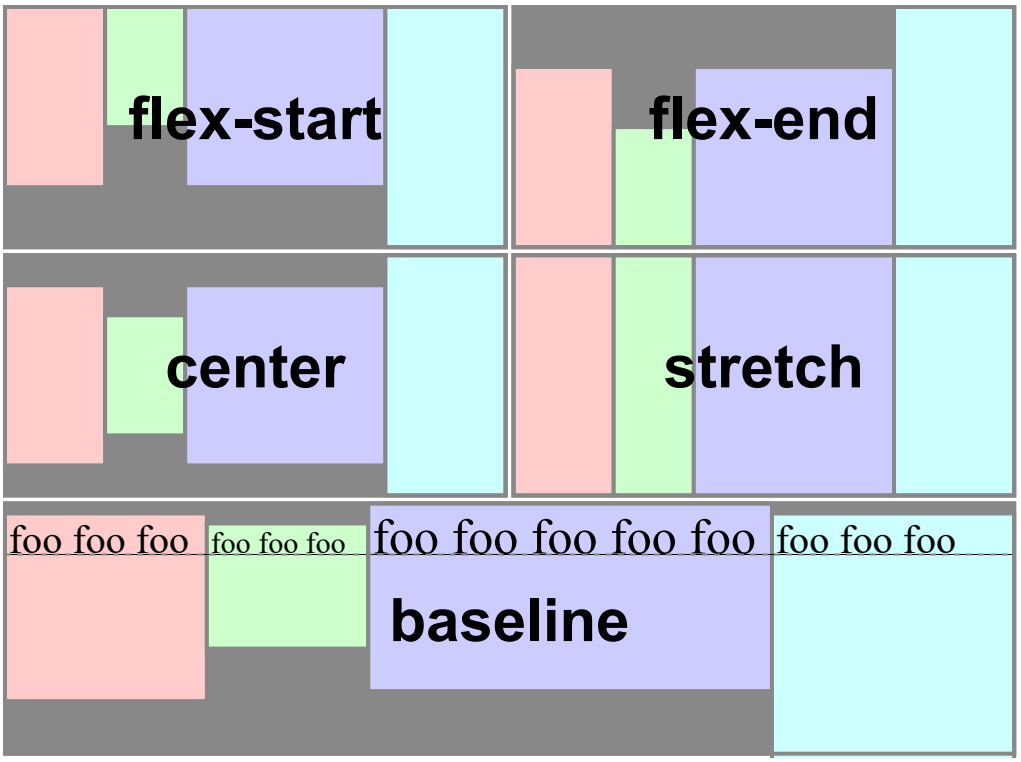
The flex item ***participates in baseline alignment***: all participating flex items on the line are aligned such that their baselines align, and the item with the largest distance between its baseline and its cross-start margin edge is placed flush against the cross-start edge of the line. If the item does not have a baseline in the necessary axis, then one is synthesized from the flex item's border box.

'stretch'

If the cross size property of the flex item computes to 'auto', and neither of the cross-axis margins are 'auto', the flex item is ***stretched***. Its used value is the length necessary to make the cross size of the item's margin box as close to the same size as the line as possible, while still respecting the constraints imposed by 'min-height'/'min-width'/'max-height'/'max-width'.

> Note: If the flex container's height is constrained this value may cause the contents of the flex item to overflow the item.

The cross-start margin edge of the flex item is placed flush with the cross-start edge of the line.



An illustration of the five 'align-items' keywords and their effects on a flex container with four colored items.

§ 8.4. Packing Flex Lines: the 'align-content' property

Name:	***'align-content'***
Value:	flex-start | flex-end | center | space-between | space-around | stretch
Initial:	stretch
Applies to:	multi-line flex containers
Inherited:	no
Percentages:	n/a
Computed value:	specified keyword
Canonical order:	per grammar
Animation type:	discrete

The 'align-content' property aligns a flex container's lines within the flex container when there is extra space in the cross-axis, similar to how 'justify-content' aligns individual items within the main-axis. Note, this property has no effect on a single-line flex container. Values have the following meanings:

'flex-start'
: Lines are packed toward the start of the flex container. The cross-start edge of the first line in the flex container is placed flush with the cross-start edge of the flex container, and each subsequent line is placed flush with the preceding line.

'flex-end'
: Lines are packed toward the end of the flex container. The cross-end edge of the last line is placed flush with the cross-end edge of the flex container, and each preceding line is placed flush with the subsequent line.

'center'
: Lines are packed toward the center of the flex container. The lines in the flex container are placed flush with each other and aligned in the center of the flex container, with equal amounts of space between the cross-start content edge of the flex container and the first line in the flex container, and between the cross-end content edge of the flex container and the last line in the flex container. (If the leftover free-space is negative, the lines will overflow equally in both directions.)

'space-between'

Lines are evenly distributed in the flex container. If the leftover free-space is negative or there is only a single flex line in the flex container, this value is identical to 'flex-start'. Otherwise, the cross-start edge of the first line in the flex container is placed flush with the cross-start content edge of the flex container, the cross-end edge of the last line in the flex container is placed flush with the cross-end content edge of the flex container, and the remaining lines in the flex container are distributed so that the spacing between any two adjacent lines is the same.

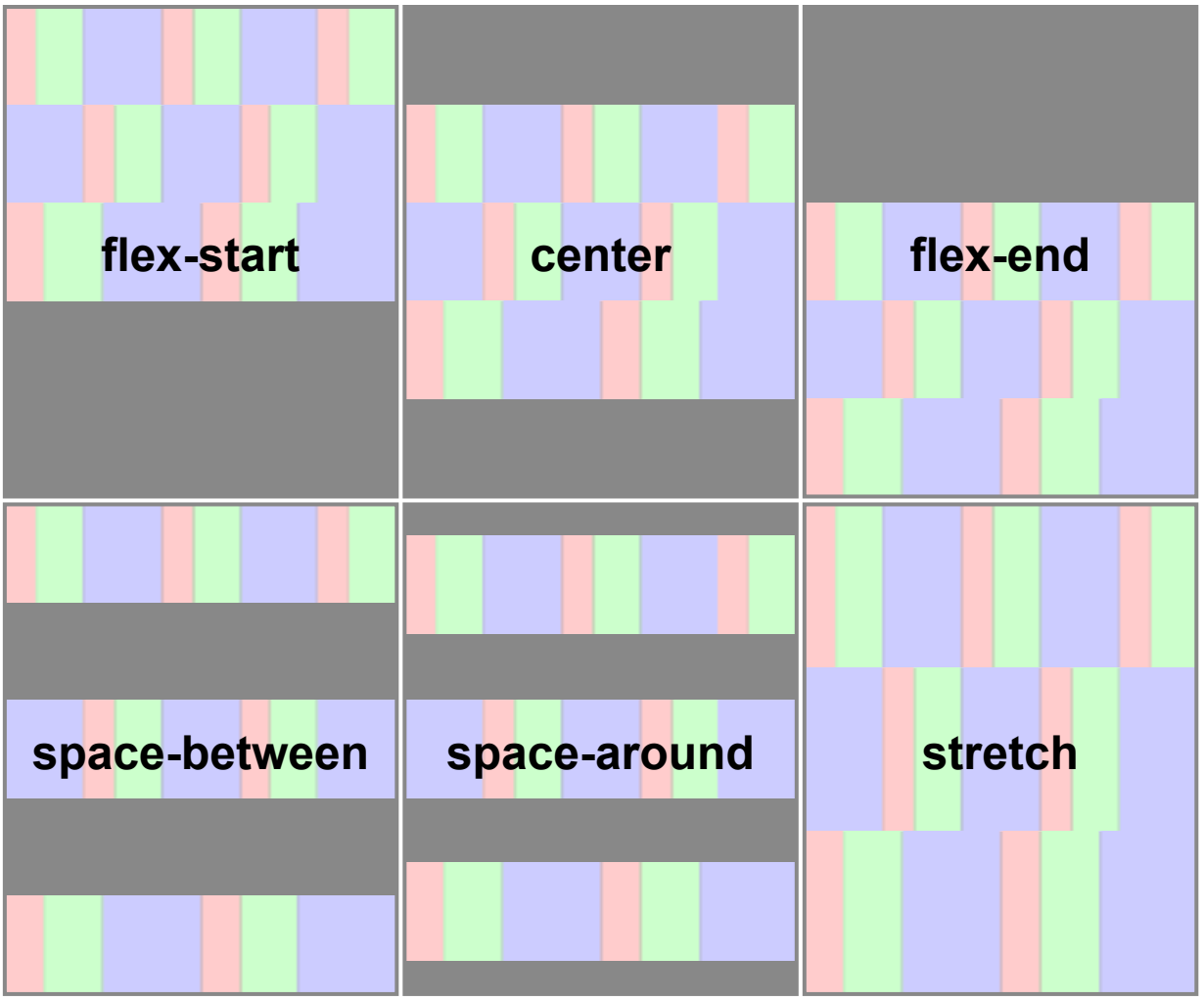
'space-around'

Lines are evenly distributed in the flex container, with half-size spaces on either end. If the leftover free-space is negative this value is identical to 'center'. Otherwise, the lines in the flex container are distributed such that the spacing between any two adjacent lines is the same, and the spacing between the first/last lines and the flex container edges is half the size of the spacing between flex lines.

'stretch'

Lines stretch to take up the remaining space. If the leftover free-space is negative, this value is identical to 'flex-start'. Otherwise, the free-space is split equally between all of the lines, increasing their cross size.

Note: Only multi-line flex containers ever have free space in the cross-axis for lines to be aligned in, because in a single-line flex container the sole line automatically stretches to fill the space.



An illustration of the 'align-content' keywords and their effects on a multi-line flex container.

§ 8.5. Flex Container Baselines

In order for a flex container to itself participate in baseline alignment (e.g. when the flex container is itself a flex item in an outer flex container), it needs to submit the position of the baselines that will best represent its contents. To this end, the baselines of a flex container are determined as follows (after reordering with 'order', and taking 'flex-direction' into account):

first/last *main-axis baseline set*

When the inline axis of the flex container matches its main axis, its baselines are determined as follows:

1. If any of the flex items on the flex container's startmost/endmost flex line participate in baseline alignment, the flex container's first/last main-axis baseline set is generated from the shared alignment baseline of those flex items.

2. Otherwise, if the flex container has at least one flex item, the flex container's first/last main-axis baseline set is generated from the alignment baseline of the startmost/endmost flex item. (If that item has no alignment baseline parallel to the flex container's main axis, then one is first synthesized from its border edges.)
3. Otherwise, the flex container has no first/last main-axis baseline set, and one is synthesized if needed according to the rules of its alignment context.

first/last *cross-axis baseline set*

When the inline axis of the flex container matches its cross axis, its baselines are determined as follows:

1. If the flex container has at least one flex item, the flex container's first/last cross-axis baseline set is generated from the alignment baseline of the startmost/endmost flex item. (If that item has no alignment baseline parallel to the flex container's cross axis, then one is first synthesized from its border edges.)
2. Otherwise, the flex container has no first/last cross-axis baseline set, and one is synthesized if needed according to the rules of its alignment context.

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 flex 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.

§ 9. Flex Layout Algorithm

This section contains normative algorithms detailing the exact layout behavior of a flex container and its contents. The algorithms here are written to optimize readability and theoretical simplicity, and may not necessarily be the most efficient. Implementations may use whatever actual algorithms they wish, but must produce the same results as the algorithms described here.

> Note: This section is mainly intended for implementors. Authors writing web pages should generally be served well by the individual property descriptions, and do not need to read this section unless they have a deep-seated urge to understand arcane details of CSS layout.

The following sections define the algorithm for laying out a flex container and its contents.

> Note: Flex layout works with the flex items in order-modified document order, not their original document order.

§ 9.1. Initial Setup

¶ 1. **Generate anonymous flex items** as described in §4 Flex Items.

§ 9.2. Line Length Determination

¶ 2. **Determine the available main and cross space for the flex items.** For each dimension, if that dimension of the flex container's content box is a definite size, use that; if that dimension of the flex container is being sized under a min or max-content constraint, the available space in that dimension is that constraint; otherwise, subtract the flex container's margin, border, and padding from the space available to the flex container in that dimension and use that value. This might result in an infinite value.

> EXAMPLE 13
>
> For example, the available space to a flex item in a floated 'auto'-sized flex container is:
>
> - the width of the flex container's containing block minus the flex container's margin, border, and padding in the horizontal dimension
> - infinite in the vertical dimension

¶ 3. **Determine the *flex base size* and *hypothetical main size* of each item:**

A. If the item has a definite used flex basis, that's the flex base size.

B. If the flex item has ...

- an intrinsic aspect ratio,
- a used flex basis of 'content', and
- a definite cross size,

then the flex base size is calculated from its inner cross size and the flex item's intrinsic aspect ratio.

C. If the used flex basis is 'content' or depends on its available space, and the flex container is being sized under a min-content or max-content constraint (e.g. when performing automatic table layout [CSS21]), size the item under that constraint. The flex base size is the item's resulting main size.

D. Otherwise, if the used flex basis is 'content' or depends on its available space, the available main size is infinite, and the flex item's inline axis is parallel to the main axis, lay the item out using the rules for a box in an orthogonal flow [CSS3-WRITING-MODES]. The flex base size is the item's max-content main size.

> Note: This case occurs, for example, in an English document (horizontal writing mode) containing a column flex container containing a vertical Japanese (vertical writing mode) flex item.

E. Otherwise, size the item into the available space using its used flex basis in place of its main size, treating a value of 'content' as 'max-content'. If a cross size is needed to determine the main size (e.g. when the flex item's main size is in its block axis) and the flex item's cross size is 'auto' and not definite, in this calculation use 'fit-content' as the flex item's cross size. The flex base size is the item's resulting main size.

When determining the flex base size, the item's min and max main sizes are ignored (no clamping occurs). Furthermore, the sizing calculations that floor the content box size at zero when applying 'box-sizing' are also ignored. (For example, an item with a specified size of zero, positive padding, and 'box-sizing: border-box' will have an outer flex base size of zero—and hence a negative inner flex base size.)

The hypothetical main size is the item's flex base size clamped according to its used min and max main sizes (and flooring the content box size at zero).

¶ 4. **Determine the main size of the flex container** using the rules of the formatting context in which it participates. For this computation, 'auto' margins on flex items are treated as '0'.

§ 9.3. Main Size Determination

¶ 5. **Collect flex items into flex lines:**

- If the flex container is single-line, collect all the flex items into a single flex line.
- Otherwise, starting from the first uncollected item, collect consecutive items one by one until the first time that the *next* collected item would not fit into the flex container's inner

main size (or until a forced break is encountered, see §10 Fragmenting Flex Layout). If the very first uncollected item wouldn't fit, collect just it into the line.

For this step, the size of a flex item is its outer hypothetical main size. (Note: This can be negative.)

Repeat until all flex items have been collected into flex lines.

> Note that the "collect as many" line will collect zero-sized flex items onto the end of the previous line even if the last non-zero item exactly "filled up" the line.

6. **Resolve the flexible lengths** of all the flex items to find their used main size. See §9.7 Resolving Flexible Lengths.

§ 9.4. Cross Size Determination

7. **Determine the *hypothetical cross size* of each item** by performing layout with the used main size and the available space, treating 'auto' as 'fit-content'.

8. **Calculate the cross size of each flex line.**

   If the flex container is single-line and has a definite cross size, the cross size of the flex line is the flex container's inner cross size.

   Otherwise, for each flex line:

   1. Collect all the flex items whose inline-axis is parallel to the main-axis, whose 'align-self' is 'baseline', and whose cross-axis margins are both non-'auto'. Find the largest of the distances between each item's baseline and its hypothetical outer cross-start edge, and the largest of the distances between each item's baseline and its hypothetical outer cross-end edge, and sum these two values.
   2. Among all the items not collected by the previous step, find the largest outer hypothetical cross size.
   3. The used cross-size of the flex line is the largest of the numbers found in the previous two steps and zero.

      If the flex container is single-line, then clamp the line's cross-size to be within the container's computed min and max cross sizes. Note that if CSS 2.1's definition of min/max-width/height applied more generally, this behavior would fall out automatically.

9. **Handle 'align-content: stretch'.** If the flex container has a definite cross size, 'align-content' is 'stretch', and the sum of the flex lines' cross sizes is less than the flex container's inner cross size, increase the cross size of each flex line by equal amounts such that the sum of their cross sizes exactly equals the flex container's inner cross size.

10. **Collapse 'visibility:collapse' items.** If any flex items have 'visibility: collapse', note the cross size of the line they're in as the item's ***strut size***, and restart layout from the beginning.

    In this second layout round, when collecting items into lines, treat the collapsed items as having zero main size. For the rest of the algorithm following that step, ignore the collapsed items entirely (as if they were 'display:none') except that after calculating the cross size of the lines, if any line's cross size is less than the largest *strut size* among all the collapsed items in the line, set its cross size to that *strut size*.

    Skip this step in the second layout round.

11. **Determine the used cross size of each flex item.** If a flex item has 'align-self: stretch', its computed cross size property is 'auto', and neither of its cross-axis margins are 'auto', the used outer cross size is the used cross size of its flex line, clamped according to the item's used min and max cross sizes. Otherwise, the used cross size is the item's hypothetical cross size.

    If the flex item has 'align-self: stretch', redo layout for its contents, treating this used size as its definite cross size so that percentage-sized children can be resolved.

    > Note that this step does not affect the main size of the flex item, even if it has an intrinsic aspect ratio.

9.5. Main-Axis Alignment

12. **Distribute any remaining free space.** For each flex line:

    1. If the remaining free space is positive and at least one main-axis margin on this line is 'auto', distribute the free space equally among these margins. Otherwise, set all 'auto' margins to zero.
    2. Align the items along the main-axis per 'justify-content'.

9.6. Cross-Axis Alignment

13. **Resolve cross-axis 'auto' margins.** If a flex item has 'auto' cross-axis margins:

- If its outer cross size (treating those 'auto' margins as zero) is less than the cross size of its flex line, distribute the difference in those sizes equally to the 'auto' margins.
- Otherwise, if the block-start or inline-start margin (whichever is in the cross axis) is 'auto', set it to zero. Set the opposite margin so that the outer cross size of the item equals the cross size of its flex line.

¶ 14. **Align all flex items along the cross-axis** per 'align-self', if neither of the item's cross-axis margins are 'auto'.

¶ 15. **Determine the flex container's used cross size:**

- If the cross size property is a definite size, use that, clamped by the used min and max cross sizes of the flex container.
- Otherwise, use the sum of the flex lines' cross sizes, clamped by the used min and max cross sizes of the flex container.

¶ 16. **Align all flex lines** per 'align-content'.

§ 9.7. Resolving Flexible Lengths

To resolve the flexible lengths of the items within a flex line:

1. **Determine the used flex factor**. Sum the outer hypothetical main sizes of all items on the line. If the sum is less than the flex container's inner main size, use the flex grow factor for the rest of this algorithm; otherwise, use the flex shrink factor.
2. **Size inflexible items.** Freeze, setting its ***target main size*** to its hypothetical main size…
   - any item that has a flex factor of zero
   - if using the flex grow factor: any item that has a flex base size greater than its hypothetical main size
   - if using the flex shrink factor: any item that has a flex base size smaller than its hypothetical main size
3. **Calculate *initial free space*.** Sum the outer sizes of all items on the line, and subtract this from the flex container's inner main size. For frozen items, use their outer target main size; for other items, use their outer flex base size.
4. Loop:
   a. **Check for flexible items.** If all the flex items on the line are frozen, free space has been distributed; exit this loop.

b. **Calculate the *remaining free space*** as for initial free space, above. If the sum of the unfrozen flex items' flex factors is less than one, multiply the initial free space by this sum. If the magnitude of this value is less than the magnitude of the remaining free space, use this as the remaining free space.

c. **Distribute free space proportional to the flex factors.**

If the remaining free space is zero
Do nothing.

If using the flex grow factor
Find the ratio of the item's flex grow factor to the sum of the flex grow factors of all unfrozen items on the line. Set the item's target main size to its flex base size plus a fraction of the remaining free space proportional to the ratio.

If using the flex shrink factor
For every unfrozen item on the line, multiply its flex shrink factor by its inner flex base size, and note this as its ***scaled flex shrink factor***. Find the ratio of the item's scaled flex shrink factor to the sum of the scaled flex shrink factors of all unfrozen items on the line. Set the item's target main size to its flex base size minus a fraction of the absolute value of the remaining free space proportional to the ratio. Note this may result in a negative inner main size; it will be corrected in the next step.

Otherwise
Do nothing.

d. **Fix min/max violations.** Clamp each non-frozen item's target main size by its used min and max main sizes and floor its content-box size at zero. If the item's target main size was made smaller by this, it's a max violation. If the item's target main size was made larger by this, it's a min violation.

e. **Freeze over-flexed items.** The total violation is the sum of the adjustments from the previous step `∑(clamped size - unclamped size)`. If the total violation is:

Zero
Freeze all items.

Positive
Freeze all the items with min violations.

Negative
Freeze all the items with max violations.

f. Return to the start of this loop.

5. Set each item's used main size to its target main size.

§ 9.8. Definite and Indefinite Sizes

Although CSS Sizing [CSS-SIZING-3] defines definite and indefinite lengths, Flexbox has several additional cases where a length can be considered ***definite***:

1. If a single-line flex container has a definite cross size, the outer cross size of any stretched flex items is the flex container's inner cross size (clamped to the flex item's min and max cross size) and is considered definite.
2. If the flex container has a definite main size, a flex item's post-flexing main size is treated as definite, even though it can rely on the indefinite sizes of any flex items in the same line.
3. Once the cross size of a flex line has been determined, items in auto-sized flex containers are also considered definite for the purpose of layout; see step 11.

> Note: The main size of a fully inflexible item with a definite flex basis is, by definition, definite.

§ 9.9. Intrinsic Sizes

The intrinsic sizing of a flex container is used to produce various types of content-based automatic sizing, such as shrink-to-fit logical widths (which use the 'fit-content' formula) and content-based logical heights (which use the max-content size).

See [CSS-SIZING-3] for a definition of the terms in this section.

§ 9.9.1. Flex Container Intrinsic Main Sizes

The **max-content main size of a flex container** is the smallest size the flex container can take while maintaining the max-content contributions of its flex items, insofar as allowed by the items' own flexibility:

1. For each flex item, subtract its outer flex base size from its max-content contribution size. If that result is positive, divide by its flex grow factor floored at 1; if negative, divide by its scaled flex shrink factor having floored the flex shrink factor at 1. This is the item's *max-content flex fraction*.
2. Place all flex items into lines of infinite length.
3. Within each line, find the largest *max-content flex fraction* among all the flex items. Add each item's flex base size to the product of its flex grow factor (or scaled flex shrink factor, if the

chosen *max-content flex fraction* was negative) and the chosen *max-content flex fraction*, then clamp that result by the max main size floored by the min main size.

4. The flex container's max-content size is the largest sum of the afore-calculated sizes of all items within a single line.

The **min-content main size** of a *single-line* flex container is calculated identically to the max-content main size, except that the flex item's min-content contribution is used instead of its max-content contribution. However, for a *multi-line* container, it is simply the largest min-content contribution of all the flex items in the flex container.

▶ **Implications of this algorithm when the sum of flex is less than 1**

§ 9.9.2. Flex Container Intrinsic Cross Sizes

The **min-content/max-content cross size** of a *single-line* flex container is the largest min-content contribution/max-content contribution (respectively) of its flex items.

For a *multi-line* flex container, the min-content/max-content cross size is the sum of the flex line cross sizes resulting from sizing the flex container under a cross-axis min-content constraint/max-content constraint (respectively). However, if the flex container is 'flex-flow: column wrap;', then it's sized by first finding the largest min-content/max-content cross-size contribution among the flex items (respectively), then using that size as the available space in the cross axis for each of the flex items during layout.

> Note: This heuristic for 'column wrap' flex containers gives a reasonable approximation of the size that the flex container should be, with each flex item ending up as min(*item's own max-content*, *maximum min-content among all items*), and each flex line no larger than its largest flex item. It's not a *perfect* fit in some cases, but doing it completely correct is insanely expensive, and this works reasonably well.

§ 9.9.3. Flex Item Intrinsic Size Contributions

The **main-size min-content contribution of a flex item** is the larger of its *outer* min-content size and outer preferred size (its 'width'/'height' as appropriate) if that is not 'auto', clamped by its flex base size as a maximum (if it is not growable) and/or as a minimum (if it is not shrinkable), and then further clamped by its min/max main size.

The **main-size max-content contribution of a flex item** is the larger of its *outer* max-content size and outer preferred size (its 'width'/'height' as appropriate) clamped by its flex base size as a maximum (if it is not growable) and/or as a minimum (if it is not shrinkable), and then further clamped by its min/max main size.

§ 10. Fragmenting Flex Layout

Flex containers can break across pages between items, between lines of items (in multi-line mode), and inside items. The 'break-*' properties apply to flex containers as normal for block-level or inline-level boxes. This section defines how they apply to flex items and the contents of flex items. See the CSS Fragmentation Module for more context [CSS3-BREAK].

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.) For readability, in this section the terms "row" and "column" refer to the relative orientation of the flex container with respect to the block flow direction of the fragmentation context, rather than to that of the flex container itself.

The exact layout of a fragmented flex container is not defined in this level of Flexible Box Layout. However, breaks inside a flex container are subject to the following rules (interpreted using order-modified document order):

- In a row flex container, the 'break-before' and 'break-after' values on flex items are propagated to the flex line. The 'break-before' values on the first line and the 'break-after' values on the last line are propagated to the flex container.

  > Note: Break propagation (like 'text-decoration' propagation) does not affect computed values.

- In a column flex container, the 'break-before' values on the first item and the 'break-after' values on the last item are propagated to the flex container. Forced breaks on other items are applied to the item itself.
- A forced break inside a flex item effectively increases the size of its contents; it does not trigger a forced break inside sibling items.
- In a row flex container, Class A break opportunities occur between sibling flex lines, and Class C break opportunities occur between the first/last flex line and the flex container's content edges. In a column flex container, Class A break opportunities occur between sibling flex items, and Class C break opportunities occur between the first/last flex items on a line and the flex container's content edges. [CSS3-BREAK]

- When a flex container is continued after a break, the space available to its flex items (in the block flow direction of the fragmentation context) is reduced by the space consumed by flex container fragments on previous pages. The space consumed by a flex 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.
- If the first fragment of the flex container is not at the top of the page, and none of its flex items fit in the remaining space on the page, the entire fragment is moved to the next page.
- When a multi-line column flex container breaks, each fragment has its own "stack" of flex lines, just like each fragment of a multi-column container has its own row of column boxes.
- Aside from the rearrangement of items imposed by the previous point, UAs should attempt to minimize distortion of the flex container with respect to unfragmented flow.

§ 10.1. Sample Flex Fragmentation Algorithm

This informative section presents a possible fragmentation algorithm for flex containers. Implementors are encouraged to improve on this algorithm and provide feedback to the CSS Working Group.

EXAMPLE 14

This algorithm assumes that pagination always proceeds only in the forward direction; therefore, in the algorithms below, alignment is mostly ignored prior to pagination. Advanced layout engines may be able to honor alignment across fragments.

single-line column flex container

1. Run the flex layout algorithm (without regards to pagination) through Cross Sizing Determination.
2. Lay out as many consecutive flex items or item fragments as possible (but at least one or a fragment thereof), starting from the first, until there is no more room on the page or a forced break is encountered.
3. If the previous step ran out of room and the free space is positive, the UA may reduce the distributed free space on this page (down to, but not past, zero) in order to make room for the next unbreakable flex item or fragment. Otherwise, the item or fragment that does not fit is pushed to the next page. The UA should pull up if more than 50% of the fragment would have fit in the remaining space and should push otherwise.
4. If there are any flex items or fragments not laid out by the previous steps, rerun the flex layout algorithm from Line Length Determination through Cross Sizing Determination with the next page's size and *all* the contents (including those already laid out), and return to the previous step, but starting from the first item or fragment not already laid out.
5. For each fragment of the flex container, continue the flex layout algorithm from Main-Axis Alignment to its finish.

It is the intent of this algorithm that column-direction single-line flex containers paginate very similarly to block flow. As a test of the intent, a flex container with 'justify-content:start' and no flexible items should paginate identically to a block with in-flow children with same content, same used size and same used margins.

multi-line column flex container

1. Run the flex layout algorithm *with* regards to pagination (limiting the flex container's maximum line length to the space left on the page) through Cross Sizing Determination.
2. Lay out as many flex lines as possible (but at least one) until there is no more room in the flex container in the cross dimension or a forced break is encountered:

1. Lay out as many consecutive flex items as possible (but at least one), starting from the first, until there is no more room on the page or a forced break is encountered. Forced breaks *within* flex items are ignored.
2. If this is the first flex container fragment, this line contains only a single flex item that is larger than the space left on the page, and the flex container is not at the top of the page already, move the flex container to the next page and restart flex container layout entirely.
3. If there are any flex items not laid out by the first step, rerun the flex layout algorithm from Main Sizing Determination through Cross Sizing Determination using only the items not laid out on a previous line, and return to the previous step, starting from the first item not already laid out.

3. If there are any flex items not laid out by the previous step, rerun the flex layout algorithm from Line Sizing Determination through Cross Sizing Determination with the next page's size and only the items not already laid out, and return to the previous step, but starting from the first item not already laid out.
4. For each fragment of the flex container, continue the flex layout algorithm from Main-Axis Alignment to its finish.

> If a flex item does not entirely fit on a single page, it will *not* be paginated in multi-line column flex containers.

single-line row flex container

1. Run the entire flex layout algorithm (without regards to pagination), except treat any 'align-self' other than 'flex-start' or 'baseline' as 'flex-start'.
2. If an unbreakable item doesn't fit within the space left on the page, and the flex container is not at the top of the page, move the flex container to the next page and restart flex container layout entirely.
3. For each item, lay out as much of its contents as will fit in the space left on the page, and fragment the remaining content onto the next page, rerunning the flex layout algorithm from Line Length Determination through Main-Axis Alignment into the new page size using *all* the contents (including items completed on previous pages).

   > Any flex items that fit entirely into previous fragments still take up space in the main axis in later fragments.

4. For each fragment of the flex container, rerun the flex layout algorithm from Cross-Axis Alignment to its finish. For all fragments besides the first, treat 'align-self' and 'align-content' as being 'flex-start' for all item fragments and lines.
5. If any item, when aligned according to its original 'align-self' value into the combined cross size of all the flex container fragments, would fit entirely within a single flex container fragment, it may be shifted into that fragment and aligned appropriately.

multi-line row flex container

1. Run the flex layout algorithm (without regards to pagination), through Cross Sizing Determination.
2. Lay out as many flex lines as possible (but at least one), starting from the first, until there is no more room on the page or a forced break is encountered.
   If a line doesn't fit on the page, and the line is not at the top of the page, move the line to the next page and restart the flex layout algorithm entirely, using only the items in and following this line.

   If a flex item itself causes a forced break, rerun the flex layout algorithm from Main Sizing Determination through Main-Axis Alignment, using only the items on this and following lines, but with the item causing the break automatically starting a new line in the line breaking step, then continue with this step. Forced breaks *within* flex items are ignored.
3. If there are any flex items not laid out by the previous step, rerun the flex layout algorithm from Line Length Determination through Main-Axis Alignment with the next page's size and only the items not already laid out. Return to the previous step, but starting from the first line not already laid out.
4. For each fragment of the flex container, continue the flex layout algorithm from Cross Axis Alignment to its finish.

§ Appendix A: Axis Mappings

This appendix is non-normative.

Axis Mappings for 'ltr' + 'horizontal-tb' Writing Mode (e.g. English)

'flex-flow'	main axis	start	end	cross axis	start	end
'row' + 'nowrap'/'wrap'	horizontal	left	right	vertical	top	bottom
'row-reverse' + 'nowrap'/'wrap'		right	left			
'row' + 'wrap-reverse'		left	right		bottom	top
'row-reverse' + 'wrap-reverse'		right	left			
'column' + 'nowrap'/'wrap'	vertical	top	bottom	horizontal	left	right
'column-reverse' + 'nowrap'/'wrap'		bottom	top			
'column' + 'wrap-reverse'		top	bottom		right	left
'column-reverse' + 'wrap-reverse'		bottom	top			

Axis Mappings for 'rtl' + 'horizontal-tb' Writing Mode (e.g. Farsi)

'flex-flow'	main axis	main-start	main-end	cross axis	cross-start	cross-end
'row' + 'nowrap'/'wrap'	horizontal	right	left	vertical	top	bottom
'row-reverse' + 'nowrap'/'wrap'		left	right			
'row' + 'wrap-reverse'		right	left		bottom	top

'flex-flow'	main axis	main-start	main-end	cross axis	cross-start	cross-end
'row-reverse' + 'wrap-reverse'	vertical	left	right	horizontal	right	left
'column' + 'nowrap'/'wrap'		top	bottom			
'column-reverse' + 'nowrap'/'wrap'		bottom	top			
'column' + 'wrap-reverse'		top	bottom		left	right
'column-reverse' + 'wrap-reverse'		bottom	top			

Axis Mappings for 'ltr' + 'vertical-rl' Writing Mode (e.g. Japanese)

'flex-flow'	main axis	start	end	cross axis	start	end
'row' + 'nowrap'/'wrap'	vertical	top	bottom	horizontal	right	left
'row-reverse' + 'nowrap'/'wrap'		bottom	top			
'row' + 'wrap-reverse'		top	bottom		left	right
'row-reverse' + 'wrap-reverse'		bottom	top			
'column' + 'nowrap'/'wrap'	vertical	right	left	horizonal	top	bottom
'column-reverse' + 'nowrap'/'wrap'		left	right			

‘flex-flow’	main axis	start	end	cross axis	start	end
‘column’ + ‘wrap-reverse’		right	left		bottom	top
‘column-reverse’ + ‘wrap-reverse’		left	right			

§ Acknowledgments

Thanks for feedback and contributions to

Erik Anderson, Christian Biesinger, Tony Chang, Phil Cupp, Arron Eicholz, James Elmore, Andrew Fedoniouk, Brian Heuston, Shinichiro Hamaji, Daniel Holbert, Ben Horst, John Jansen, Brad Kemper, Kang-hao Lu, Markus Mielke, Peter Moulder, Robert O’Callahan, Christoph Päper, Ning Rogers, Peter Salas, Elliott Sprehn, Morten Stenshorne, Christian Stockwell, Ojan Vafai, Eugene Veselov, Greg Whitworth, Boris Zbarsky.

§ Changes

This section documents the changes since previous publications.

§ Changes since the 16 October 2017 CR

A Disposition of Comments is also available.

¶ • Removed the option for flex-item block-axis margins and paddings to be resolved against the block dimension; they must be resolved against the inline dimension, as for blocks. (Issue 2085)

¶ • Floored flex item’s min-content contribution by their preferred size (Issue 2353) and cleaned up associated wording to be more precise.

> The main-size min-content contribution of a flex item is the larger of its *outer* min-content size and outer preferred size (its 'width'/'height' as appropriate) if that is not 'auto' , clamped by its flex base size as a maximum (if it is not growable) and/or as a minimum (if it is not shrinkable), and then further clamped by its min/max main size properties.
>
> The main-size max-content contribution of a flex item is the larger of its *outer* max-content size and outer ~~specified~~ preferred size (its 'width'/'height' as appropriate ~~,~~) if that is ~~definite~~ not 'auto' , clamped by its flex base size as a maximum (if it is not growable) and/or as a minimum (if it is not shrinkable), and then further clamped by its min/max main size properties.

¶
- Added some (effectively informative) prose and a cross-reference to more clearly define 'flex-basis: content'.

> Indicates ~~automatic sizing~~ an automatic size based on the flex item's content. (It is typically equivalent to the max-content size, but with adjustments to handle aspect ratios, intrinsic sizing constraints, and orthogonal flows; see details in §9 Flex Layout Algorithm.)

¶
- Moved the definition of the 'auto' keyword for 'min-width' and 'min-height' to [CSS-SIZING-3]. The definition of what an automatic minimum size for flex items is remains here. (Issue 1920, Issue 2103)

¶
- Altered the computation of 'auto' in 'min-width' and 'min-height' such that it always computes to itself—although its resolved value remains zero on CSS2 display types. (Issue 2230, Issue 2248)

¶
- Clarified that min/max clamping is according to the used value of the min/max size properties—which in the case of tables with 'auto' layout, is floored by the table's min-content size. (Issue 2442)

¶
- Clarified that break propagation does not affect computed values and that order-modified document order is used. (Issue 2614)

> The exact layout of a fragmented flex container is not defined in this level of Flexible Box Layout. However, breaks inside a flex container are subject to the following rules (interpreted using order-modified document order) :
>
> - In a row flex container, the 'break-before' and 'break-after' values on flex items are propagated to the flex line. The 'break-before' values on the first line and the 'break-after' values on the last line are propagated to the flex container.
>
>> Note: Break propagation (like 'text-decoration' propagation) does not affect computed values.

¶ - Clarified that if the automatic minimum size resolves directly to zero rather than being a content-based minimum size, it does not cause indefiniteness.

> For the purpose of calculating an intrinsic size of the element (e.g. the element's min-content size), ~~this value~~ a content-based minimum size causes the element's size in that axis to become indefinite (even if e.g. its 'width' property specifies a definite size).

¶ - Editorial improvements to the automatic minimum size. (Issue 2385)
- Some minor editorial fixes and clarifications, including updates to vocabulary to match updates to other CSS modules.

§ Changes since the 26 May 2016 CR

A Disposition of Comments is also available.

§ Substantive Changes and Bugfixes

¶ - To allow flex factors to actually represent absolute ratios of flex item sizes as was originally intended (see various examples), removed the flooring of content-box sizes at zero for the purpose of finding the item's flex base size, since this type of ratio requires a flex base size of zero, which would otherwise only be possible if margins, borders, and padding are also all zero. (The flooring remains in effect, alongside the min and max size constraints, in calculating the hypothetical and final sizes of the item.) (Issue 316)

> When determining the flex base size, the item's min and max main size properties are ignored (no clamping occurs). Furthermore, the sizing calculations that floor the content box size at zero when applying 'box-sizing' are also ignored. (For example, an item with a specified size of zero, positive padding, and 'box-sizing: border-box' will have an outer flex base size of zero—and hence a negative inner flex base size.)
>
> The hypothetical main size is the item's flex base size clamped according to its min and max main size properties (and flooring the content box size at zero) .

> **Fix min/max violations.** Clamp each non-frozen item's target main size by its min and max main size properties and floor its content-box size at zero . If the item's target main size was made smaller by this, it's a max violation. If the item's target main size was made larger by this,

¶

- To prevent empty flex items in shrink-to-fit containers from collapsing to zero even when given a specified size, the specified size is now accounted for in calculating its max-content contribution in §9.9.3 Flex Item Intrinsic Size Contributions. (Issue 1435)

> The **main-size max-content contribution of a flex item** is the larger of its *outer* max-content size and specified size (its 'width'/'height' as appropriate, if that is definite) , clamped by its flex base size as a maximum (if it is not growable) and/or as a minimum (if it is not shrinkable), and then further clamped by its min/max main size properties.

¶

- Since at least two implementations ended up allowing percentages inside flex items with indefinite flex basis to resolve anyway, removed the condition requiring definite flex basis. (Issue 1679)

> If ~~a flex item has a definite flex basis and~~ the flex container has a definite main size, ~~its~~ a flex item's post-flexing main size is treated as definite (even though it might technically rely on the ~~sizes of indefinite siblings to resolve its flexed main size~~ the indefinite sizes of any flex items in the same line).

¶

- For ease of implementation, 'auto' value of 'align-self' now computes to itself always. See related previous change requiring this computation for absolutely-positioned elements. (Issue 440, Issue 644)

Computed value:	~~'auto' computes to parent's 'align-items' value; otherwise~~ as specified

…

~~On absolutely positioned elements, a value of 'auto' computes to itself. On all other elements, a value of 'auto' for 'align-self' computes to the value of 'align-items' on the element's parent, or 'stretch' if the element has no parent.~~

- Change flex items in orthogonal flows and flex items without a baseline to both synthesize their alignment baseline from the flex item's border box. (Issue 373)
- Fix main/cross error in definition of cross-axis baseline set. (Issue 792)

> Otherwise, the flex container has no first/last ~~main~~ cross -axis baseline set…

- Restore accidentally-deleted text about tables as flex items. See anonymous box change. (Issue 547)

> In the case of flex items with 'display: table', the table wrapper box becomes the flex item, and the 'order' and 'align-self' properties apply to it. The contents of any caption boxes contribute to the calculation of the table wrapper box's min-content and max-content sizes. However, like 'width' and 'height', the 'flex' longhands apply to the table box as follows: the flex item's final size is calculated by performing layout as if the distance between the table wrapper box's edges and the table box's content edges were all part of the table box's border+padding area, and the table box were the flex item.

- Clarified that auto margins are treated as zero for the purpose of calculating a absolutely-positioned flex container child's static position. (Issue 665)

> For this purpose, a value of 'align-self: auto' is treated identically to 'start' , and 'auto' margins are treated as zero .

- When clamping by the max main size property in the calculation of the flex container's intrinsic size, be sure to floor by the min main size property. (Issue 361)

> Within each line, find the largest *max-content flex fraction* among all the flex items. Add each item's flex base size to the product of its flex grow factor (or scaled flex shrink factor, if the chosen *max-content flex fraction* was negative) and the chosen *max-content flex fraction*, then clamp that result ing item size according to ~~by the max and min main size propert y ies .~~

¶

- Added missing edits for change that made 'order' not apply to absolutely-positioned children of a flex container. (Issue 1439)

> **Applies to:** | flex items ~~and absolutely-positioned children of flex containers~~
>
> The 'order' property controls the order in which ~~children of a flex container~~ flex items appear within the flex container, by assigning them to ordinal groups. …
>
> Absolutely-positioned children of a flex container are treated as having 'order: 0' for the purpose of determining their painting order relative to flex items.

> Unless otherwise specified by a future specification, this property has no effect on boxes that are not ~~children of a flex container~~ flex items .

¶

- Take 'flex-direction' into account when determining first/last baseline of the flex container. (Issue 995)

> To this end, the baselines of a flex container are determined as follows (after reordering with 'order' , and taking 'flex-direction' into account):
>
> …
>
> 1. If any of the flex items on the flex container's ~~first/last~~ startmost/endmost flex line participate in baseline alignment, the flex container's first/last main-axis baseline set …
> 2. Otherwise, if the flex container has at least one flex item, the flex container's first/last main-axis baseline set is generated from the alignment baseline of the ~~first/last~~ startmost/endmost flex item. …
> 3. Otherwise, the flex container has no first/last main-axis baseline set, …
>
> …
>
> 1. If the flex container has at least one flex item, the flex container's first/last cross-axis baseline set is generated from the alignment baseline of the ~~first/last~~ startmost/endmost flex item. …

¶ • Define 'align-content: space-between' handling of a single flex line as equivalent to 'start'. (Issue 718)

> Lines are evenly distributed in the flex container. If the leftover free-space is negative or there is only a single flex line in the flex container, this value is identical to 'flex-start'.

¶ • Fixed error in axis mapping table. (Issue 205)

¶ • Restored definition of the automatic minimum size of boxes with neither specified size nor aspect ratio, which was lost in earlier rewrite. (Issue 671)

§ Clarifications

¶ • Made sure that main size and cross size are defined for flex containers as well as for flex items. (Issue 981)

¶ • Tweaked final clarifying sentence of note about spatial navigation. (Issue 1677)

> User agents, including browsers, accessible technology, and extensions, may offer spatial navigation features. This section does not preclude respecting the 'order' property when determining element ordering in such spatial navigation modes; indeed it would need to be considered for such a feature to work. ~~However a UA that uses 'order' in determining sequential navigation, but does not otherwise account for spatial relationships among elements (as expressed by the various layout features of CSS including and not limited to flex layout), is non-conforming.~~ But 'order' is not the only (or even the primary) CSS property that would need to be considered for such a spatial navigation feature. A well-implemented spatial navigation feature would need to consider all the layout features of CSS that modify spatial relationships.

- Miscellaneous trivial editorial improvements.

§ Changes since the 1 March 2016 CR

A Disposition of Comments is also available.

§ Substantive Changes and Bugfixes

¶ - Define how percentages are handled when calculating intrinsic automatic minimum sizes. (Issue 3)

> For the purpose of calculating an intrinsic size of the element (e.g. the element's min-content size), this value causes the element's size in that axis to become indefinite (even if e.g. its 'width' property specifies a definite size). Note this means that percentages calculated against this size will be treated as 'auto'.
>
> Nonetheless, although this may require an additional layout pass to re-resolve percentages in some cases, this value (like the 'min-content', 'max-content', and 'fit-content' values defined in [CSS-SIZING-3]) does not prevent the resolution of percentage sizes within the item.

¶ - Switched definite and indefinite to refer to the (more correct) definitions in [CSS-SIZING-3] instead of defining them inline in this module. (Issue 10)

¶ - Abspos children of a flexbox no longer respond to the 'order' property. (Issue 12)

- Updated §8.5 Flex Container Baselines to account for baseline sets and last-baseline alignment. (Issue 13))

§ Clarifications

- Clarify that spatial navigation modes are allowed to handle 'order'. (Issue 1)

> User agents, including browsers, accessible technology, and extensions, may offer spatial navigation features. This section does not preclude respecting the 'order' property when determining element ordering in such spatial navigation modes; indeed it would need to be considered for such a feature to work. However a UA that uses 'order' in determining sequential navigation, but does not otherwise account for spatial relationships among elements (as expressed by the various layout features of CSS including and not limited to flex layout), is non-conforming.

- Cross-reference an additional case of definiteness in §9.8 Definite and Indefinite Sizes (Issue 2)

> Once the cross size of a flex line has been determined, items in auto-sized flex containers are also considered definite for the purpose of layout; see step 11.

- Improve wording for how unresolveable percentage flex basis values transmute to 'content'. (Issue 6)

> For all values other than 'auto' and 'content' (defined above), 'flex-basis' is resolved the same way as 'width' in horizontal writing modes [CSS21], except that if a value would resolve to 'auto' for 'width', it instead resolves to 'content' for 'flex-basis' . For example, percentage values of 'flex-basis' are resolved against the flex item's containing block (i.e. its flex container); and if that containing block's size is indefinite, ~~the result is the same as a main size of 'auto' (which in this case is treated as 'content')~~ the used value for 'flex-basis' is 'content' .

- Clarify that inflexible items with a definite flex basis have a definite size. (Issue 8, Issue 11)

> A flex item is fully inflexible if both its 'flex-grow' and 'flex-shrink' values are zero, and flexible otherwise.

> > The main size of a fully inflexible item with a definite flex basis is, by definition, definite.

- Reworded definition of the 'auto' value to be easier to understand. (Issue 9)

> On a flex item whose 'overflow' is 'visible' in the main axis, when specified on the flex item's main-axis min-size property, ~~the following table gives the minimum size …~~ specifies an automatic minimum size.
>
> In general, the automatic minimum size … defined below:

- Slightly reworded the section on determining the static position of absolutely-positioned children to be clearer.
- Adjusted format of Animatable lines to be clearer about animating keywords.
- Miscellaneous trivial editorial improvements.

Changes since the 14 May 2015 LCWD

A Disposition of Comments is also available.

Substantive Changes and Bugfixes

- Revert 'flex' shorthand change of omitted 'flex-basis' back to '0', since that was a hacky way of solving an intrinsic size problem, and isn't needed (and gives bad results) given a correct implementation of §9.9 Intrinsic Sizes. (Issue 13)

> When omitted from the 'flex' shorthand, its specified value is '0%'.

> **'flex: <positive-number>'**
> Equivalent to 'flex: <positive-number> 1 0%'.

- Changed flex item determination to operate on each element directly, and not on its anonymous wrapper box, if any. (Issue 6)

  > - ~~'float' and 'clear' have no effect on a flex item,~~ 'float' and 'clear' do not create floating or clearance of flex item, and do not take it out-of-flow. ~~However, the 'float' property can still affect box generation by influencing the 'display' property's computed value.~~
  >
  > ~~Some values of 'display' trigger the creation of anonymous boxes around the original box. It's the outermost box—the direct child of the flex container box—that becomes a flex item. For example, given two contiguous child elements with 'display: table-cell', the anonymous table wrapper box generated around them [CSS21] becomes the flex item.~~
  >
  > ~~In the case of flex items with 'display: table', the table wrapper box becomes the flex item, and the 'order' and 'align-self' properties apply to it. The contents of any caption boxes contribute to the calculation of the table wrapper box's min-content and max-content sizes. However, like 'width' and 'height', the 'flex' longhands apply to the table box as follows: the flex item's final size is calculated by performing layout as if the distance between the table wrapper box's edges and the table box's content edges were all part of the table box's border+padding area, and the table box were the flex item.~~
  >
  > > Note: Some values of 'display' normally trigger the creation of anonymous boxes around the original box. If such a box is a flex item, it is blockified first, and so anonymous box creation will not happen. For example, two contiguous flex items with 'display: table-cell' will become two separate 'display: block' flex items, instead of being wrapped into a single anonymous table.

- Defined that any size adjustment imposed by a box's 'min-width: auto' is consulted when percentage-sizing any of its contents. (Issue 3)

~~In order to prevent cycling sizing, the 'auto' value of 'min-height' and 'max-height' does not factor into the percentage size resolution of the box's contents. For example, a percentage-height block whose flex item parent has 'height: 120em; min-height: auto' will size itself against 'height: 120em' regardless of the impact that 'min-height' might have on the used size of the flex item.~~

Although this may require an additional layout pass to re-resolve percentages in some cases, the 'auto' value of 'min-width' and 'min-height' (like the 'min-content', 'max-content', and 'fit-content' values defined in [CSS-SIZING-3]) does not prevent the resolution of percentage sizes within the item.

- Correct instrinsic sizing rules to handle inflexible items. (Issue 1)

> The main-size min-content/max-content contribution of a flex item is its outer ~~hypothetical main size when sized under a min-content/max-content constraint (respectively)~~ The **main-size min-content/max-content contribution of a flex item** is its *outer* min-content/max-content size, clamped by its flex base size as a maximum (if it is not growable) and/or as a minimum (if it is not shrinkable), and then further clamped by its min/max main size properties .

- Correct errors in flex container main-axis intrinsic sizing. (Issue 1)

> The max-content main size of a flex container is the smallest size the flex container can take while maintaining the max-content contributions of its flex items:
>
> 1. For each flex item, subtract its outer flex base size from its max-content contribution size ~~, then divide by its flex grow factor, floored at 1, or by its scaled flex shrink factor (if the result was negative, flooring the flex shrink factor at 1 if necessary)~~ . If that result is not zero, divide it by (if the result was positive) its flex grow factor floored at 1, or (if the result was negative) by its scaled flex shrink factor, having floored the flex shrink factor at 1. This is the item's *max-content flex fraction*.

- Correct errors in flex container cross-axis intrinsic sizing, and specify commonly-implemented min-content sizing heuristic for multi-line column flex containers. (Issue 12)

~~The min-content cross size and max-content cross size of a flex container are the cross size of the flex container after performing layout into the given available main-axis space and infinite available cross-axis space.~~

The min-content/max-content cross size of a *single-line* flex container is the largest min-content contribution/max-content contribution (respectively) of its flex items.

For a *multi-line* flex container, the min-content/max-content cross size is the sum of the flex line cross sizes resulting from sizing the flex container under a cross-axis min-content constraint/max-content constraint (respectively). However, if the flex container is 'flex-flow: column wrap;', then it's sized by first finding the largest min-content/max-content cross-size contribution among the flex items (respectively), then using that size as the available space in the cross axis for each of the flex items during layout.

> This heuristic for 'column wrap' flex containers gives a reasonable approximation of the size that the flex container should be, with each flex item ending up as min(*item's own max-content*, *maximum min-content among all items*), and each flex line no larger than its largest flex item. It's not a *perfect* fit in some cases, but doing it completely correct is insanely expensive, and this works reasonably well.

¶

- Add explicit conformance criteria on authoring tools to keep presentation and DOM order in sync unless author explicitly indicates a desire to make them out-of-sync. (Issue 8)

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' 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 15

For example, a tool might offer both drag-and-drop reordering of flex 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 perform drag-and-drop reordering at the DOM layer. In some cases, however, the author may want different visual orderings per screen size. The tool could offer this functionality by using 'order' together with media queries, but also tie the smallest screen size's ordering to the underlying DOM order (since this is most likely to be a logical linear presentation order) while using 'order' to determine the visual presentation order in other size ranges.

This tool would be conformant, whereas a tool that only ever used 'order' to handle drag-and-drop reordering (however convenient it might be to implement it that way) would be non-conformant.

¶ • Defined that an 'align-self' or 'justify-self' value of 'auto' computes to itself on absolutely-positioned elements, for consistency with future extensions of these properties in [CSS-ALIGN-3]. (Issue 5)

On absolutely positioned elements, a value of 'auto' computes to itself. On all other elements, a ~~A~~ value of 'auto' for 'align-self' computes to the value of 'align-items' on the element's parent, or 'stretch' if the element has no parent.

¶ • Revert change to make percentage margins and padding relative to their own axes; instead allow both behaviors. (Issue 11, Issue 16)

~~Percentage margins and paddings on~~ flex items ~~are always resolved against their respective dimensions; unlike blocks, they do not always resolve against the inline dimension of their containing block.~~

Percentage margins and paddings on flex 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 flex items entirely, as they will get different behavior in different browsers.

¶

- Handle min/max constraints in sizing flex items.

  - **Determine the available main and cross space for the flex items.** For each dimension, if that dimension of the flex container's content box is a definite size, use that; if that dimension of the flex container is being sized under a min or max-content constraint, the available space in that dimension is that constraint; otherwise, subtract the flex container's margin, border, and padding from the space available to the flex container in that dimension and use that value.

¶

- Correct negation in flex container fragmentation rule: previous definition implied 'break-inside: avoid' behavior in all cases. (Issue 5)

  - If the first fragment of the flex container is not at the top of the page, and ~~some~~ none of its flex items ~~don't~~ fit in the remaining space on the page, the entire fragment is moved to the next page.

§ Clarifications

- Miscellaneous minor editorial improvements and fixes to errors in examples.

§ Changes since the 25 September 2014 LCWD

A Disposition of Comments is also available.

§ Substantive Changes and Bugfixes

¶ • Reverted 'flex-basis: auto' to its original meaning. Added 'flex-basis: content' keyword to explicitly specify automatic content-based sizing. (Issue 10)

¶ • Made applicability of 'align-content' depend on wrappability rather than number of resulting flex lines. (Issue 4)

> ~~When a flex container has multiple lines,~~ In a multi-line flex container (even one with only a single line), the cross size of each line is the minimum size necessary [...] ~~When a flex container (even a multi-line one) has only one line,~~ In a single-line flex container, the cross size of the line is the cross size of the flex container, and 'align-content' has no effect.

> Note, this property has no effect ~~when the flex container has only a single line.~~ on a single-line flex container.

> Only ~~flex containers with multiple lines~~ multi-line flex containers ever have free space in the cross-axis for lines to be aligned in, because in a ~~flex container with a single line~~ single-line flex container the sole line automatically stretches to fill the space.

> If the flex container ~~has only one flex line (even if it's a multi-line flex container)~~ is single-line and has a definite cross size, the cross size of the flex line is the flex container's inner cross size.

> If the flex container ~~has only one flex line (even if it's a multi-line flex container),~~ is single-line, then clamp the line's cross-size to be within the container's computed min and max cross-size properties.

- Removed text that asserted forced breaking behavior, replaced with reference to fragmentation section. This resolves a conflict in the spec. (Issue 18)

> collect consecutive items one by one until the first time that the next collected item would not fit into the flex container's inner main size, (or until a forced break is encountered , see §10 Fragmenting Flex Layout) . [...] ~~A break is forced wherever the CSS2.1 'page-break-before'/'page-break-after' [CSS21] or the CSS3 'break-before'/'break-after' [CSS3-BREAK] properties specify a fragmentation break.~~

- Change the flex shrink factor to multiply by the *inner* (not outer) flex base size. (Issue 9)

> For every unfrozen item on the line, multiply its flex shrink factor by its ~~outer~~ inner flex base size, and note this as its scaled flex shrink factor.

- Add back in missing "n" in "neither"... (Issue 6)

> If the cross size property of the flex item computes to 'auto', and n either of the cross-axis margins are 'auto', the flex item is stretched.

- Specify that the flex container's main size must also be definite for a flex item's flexed main size to be definite. (Issue 20)

> [If] ... the flex item has a definite flex basis, and the flex container has a definite main size, the flex item's main size must be treated as definite ...

- Remove the requirement that the flex basis be 'content' for the specified size to be defined. The specified size should always win if it is smaller than the intrinsic size. This is particularly important to maintain author expectations for, e.g. `<img src="…" width=40 height=40 title="100x100 image">`. (Issue 25)

> If the item's ~~computed 'flex-basis' is 'content' and its~~ computed main size property is definite, then the specified size is that size

- Remove the requirement that anonymous block creation (for things like 'display: table-cell') occur *before* flex item blockification. (Instead, all children now blockify immediately, consistent with abspos/float behavior.)

§ Clarifications

- Clarify that flex base size is unclamped. (Issue 21)

> When determining the flex base size, the item's min and max main size properties are ignored (no clamping occurs).
>
> The hypothetical main size is the item's flex base size clamped according to its min and max main size properties.

- Restored normative status of note about table wrapper boxes normative; it had been accidentally changed in the previous draft. (Issue 2)
- Removed references to 'display' property longhands, since they will be removed from CSS Display Level 3.
- Change wording to not imply an unnecessary layout pass. (Issue 22)

> Otherwise, ~~lay out~~ size the item into the available space using its used flex basis in place of its main size, treating a value of 'content' as 'max-content'.

- Renamed "clamped size" to "specified size" in the definition of 'height: auto'.
- Various trivial fixes.

Changes since the 25 March 2014 LCWD

A Disposition of Comments is also available.

Substantive Changes and Bugfixes

The following significant changes were made since the 25 March 2014 Last Call Working Draft

- Fixed errors (missing negation, unspecified axis) in definition of 'min-width: auto'. (Issues 11, 18, 30)

> On a flex item whose 'overflow' is ~~not~~ 'visible' in the main axis ,

- Expanded and rewrote definition of 'min-width: auto' to add special handling of items with intrinsic ratios. (Issues 16 and 28)

> On a flex item whose 'overflow' is not 'visible', the following table gives the minimum size: '**[see table]**'
> ~~this keyword specifies as the minimum size the smaller of:~~
>
> - ~~the min-content size, or~~
> - ~~the computed 'width'/'height', if that value is definite.~~

- Adjusted 'min-width: auto' to only apply the computed main size as a minimum in cases where the flex basis was retrieved from the main size property. (Issue 19)

> … is defined if the item's computed flex-basis is 'auto' and its computed main size property is definite …

- Defined that any size adjustment imposed by a box's 'min-width: auto' is not consulted when percentage-sizing any of its contents. (Issue 27) This change was later reverted with an opposite definition.

> In order to prevent cycling sizing, the 'auto' value of 'min-height' and 'max-height' does not factor into the percentage size resolution of the box's contents. For example, a percentage-height block whose flex item parent has 'height: 120em; min-height: auto' will size itself against 'height: 120em' regardless of the impact that 'min-height' might have on the used size of the flex item.

- Introduced extra 'main-size' keyword to 'flex-basis' so that "lookup from main-size property" and "automatic sizing" behaviors could each be explicitly specified. (Issue 20) This change was later reverted with an alternative proposal solving the same problem by instead introducing the 'content' keyword.
- Defined flex items with a definite flex basis to also be definite in the main axis, allowing resolution of percentage-sized children even when the item itself is flexible. (Issue 26)

> If a percentage is going to be resolved against a flex item's main size, and the flex item has a definite flex basis, the main size must be treated as definite for the purpose of resolving the percentage, and the percentage must resolve against the flexed main size of the flex item (that is, after the layout algorithm below has been completed for the flex item's flex container, and the flex item has acquired its final size).

¶ • Clamp a single line flexbox's line cross size to the container's own min/max, even when the container's size is indefinite. (Issue 9)

  - The used cross-size of the flex line is the largest of the numbers found in the previous two steps and zero.

    If the flex container has only one flex line (even if it's a multi-line flex container), then clamp the line's cross-size to be within the container's computed min and max cross-size properties. Note that if CSS 2.1's definition of min/max-width/height applied more generally, this behavior would fall out automatically.

¶ • Fixed various errors in the new Resolving Flexible Lengths section (see March 2014 rewrite to create continuity between 'flex: 0' and 'flex: 1') and reverted the editorial structure to match the old Candidate Recommendation. (Issues 3, 4, 8, 10, 15)

¶ • Fixed max-content sizing of flex containers to account for flexing behavior by normalizing per flex fraction rather than merely summing the max-content sizes of the flex items. (Issue 39)

¶ • Updated 'flex' property to accept animations always, now that the discontinuity between 0 and non-0 values has been fixed. (Issue 5)

§ Clarifications

The following significant changes were made since the 25 March 2014 Last Call Working Draft

¶ • Clarified how the static position of an absolutely-positioned child of a flex container is calculated by introducing an explanation of the effect more closely tied with CSS2.1 concepts and terminology. (Issue 12)

> ~~Its~~ The static position of an absolutely-positioned child of a flex container is ~~calculated by first doing full flex layout without the absolutely-positioned children, then positioning each absolutely-positioned child~~ determined such that the child is positioned as if it were the sole flex item in the flex container, assuming both the child and the flex container were fixed-size boxes of their used size.
>
> In other words, the static position of an absolutely positioned child of a flex container is determined *after flex layout* by setting the child's static-position rectangle to the flex container's content box, then aligning the absolutely positioned child within this rectangle according to the 'justify-content' value of the flex container and the 'align-self' value of the child itself.

¶

- Clarified application of 'order' to absolutely-positioned children of the flex container. (Note, this behavior was later rescinded.)

> An absolutely-positioned child of a flex container does not participate in flex layout ~~beyond the reordering step~~ . However, it does participate in the reordering step (see 'order'), which has an effect in their painting order.

> The order property controls the order in which ~~flex items~~ children of a flex container appear within their flex container…
>
> Unless otherwise specified by a future specification, this property has no effect on boxes that are not ~~flex items~~ children of a flex container .

> > Note: Absolutely-positioned children of a flex container do not participate in flex layout, but are reordered together with any flex item children.

¶

- Clarified what a stretched flex item is for the purposes of special behavior (like definiteness). (Issue 25)

> If the cross size property of the flex item computes to auto, and either of the cross-axis margins are auto, the flex item is stretched. Its ~~its~~ used value …

§ Changes since the 18 September 2012 Candidate Recommendation

A Disposition of Comments is also available.

§ Substantive Changes and Bugfixes

The following significant changes were made since the 18 September 2012 Candidate Recommendation:

¶
- Changed the behavior of the new 'auto' initial value of 'min-width'/'min-height' to
  - Take into account whether 'overflow' is 'visible', since when 'overflow' is explicitly handled, it is confusing (and unnecessary) to force enough size to show all the content.
  - Take into account the specified 'width'/'height', so that the implied minimum is never greater than the specified size.
  - Compute to itself (not to 'min-content') on flex items, since they are no longer equivalent (due to above changes).

  (Issue 19)

  > **auto**
  >
  > ~~When used as the value of a flex item's min main size property, this keyword indicates a minimum of the min-content size, to help ensure that the item is large enough to fit its contents.~~
  >
  > ~~It is intended that this will compute to the 'min-content' keyword when the specification defining it ([CSS-SIZING-3]) is sufficiently mature.~~
  >
  > On a flex item whose 'overflow' is not 'visible', this keyword specifies as the minimum size the smaller of:
  >
  > - the min-content size, or
  > - the computed 'width'/'height', if that value is definite.

¶
- Specified that percentage margins/paddings on flex items are resolved against their respective dimensions, not the inline dimension of the containing block like blocks do. (Issue 16)

> Percentage margins and paddings on flex items are always resolved against their respective dimensions; unlike blocks, they do not always resolve against the inline dimension of their containing block.

¶ • Pass definiteness of a single-line flex container's size through to any stretched items. (Issue 3)

> As a special case for handling stretched flex items, if a single-line flex container has a definite cross size, the outer cross size of any flex items with 'align-self: stretch' is the flex container's inner cross size (clamped to the flex item's min and max cross size) and is considered definite.

¶ • Allow percentages inside a stretched auto-height flex item to resolve by requiring a relayout pass. (Issue 3)

> If the flex item has 'align-self: stretch', redo layout for its contents, treating this used size as its definite cross size so that percentage-sized children can be resolved.
>
> > Note that this step does not affect the main size of the flex item, even if it has an intrinsic aspect ratio.

¶ • Allow intrinsic aspect ratios to inform the main-size calculation. (Issue 8)

> If the flex item has ...
>
> - an intrinsic aspect ratio,
> - a flex basis of 'auto', and
> - a definite cross size
>
> then the flex base size is calculated from its inner cross size and the flex item's intrinsic aspect ratio.

¶ • Define hypothetical main size when the main size depends on the cross size. (Issue 23)

> If a cross size is needed to determine the main size (e.g. when the flex item's main size is in its block axis) and the flex item's cross size is 'auto' and not definite, in this calculation use 'fit-content' as the flex item's cross size.

¶

- Defined the intrinsic sizes of flex containers.

> **Determine the main size of the flex container** using its main size property. ~~In this calculation, the min-content main size of the flex container is the maximum of the flex container's items' min-content size contributions, and the max-content main size of the flex container is the sum of the flex container's items' max-content size contributions. The min-content/max-content main size contribution of an item is its outer hypothetical main size when sized under a min-content/max-content constraint (respectively).~~ For this computation, 'auto' margins on flex items are treated as '0'.

> The max-content main size of a flex container is the sum of the flex container's items' max-content contributions in the main axis. The min-content main size of a single-line flex container is the sum of the flex container's items' min-content contributions in the main axis; for a multi-line container, it is the largest of those contributions.
>
> The min-content cross size and max-content cross size of a flex container are the cross size of the flex container after performing layout into the given available main-axis space and infinite available cross-axis space.
>
> The main-size min-content/max-content contribution of a flex item is its outer hypothetical main size when sized under a min-content/max-content constraint (respectively).
>
> See [CSS-SIZING-3] for a definition of the terms in this section.

¶

- Correct an omission in the flex-line size determination, so a single-line flexbox will size to its contents if it doesn't have a definite size.

> If the flex container has only one flex line (even if it's a multi-line flex container) and has a definite cross size , the cross size of the flex line is the flex container's inner cross size.

¶

- Flex lines have their size floored at 0. (Issue 2)

> The used cross-size of the flex line is the ~~larger~~ largest of the numbers found in the previous two steps and zero .

- Flex items paint like inline blocks rather than blocks. (Issue 18)

> Flex items paint exactly the same as ~~block-level elements in the normal flow~~ inline blocks [CSS21] .

- An omitted ‘flex-basis’ component of the ‘flex’ shorthand now resolves to ‘0%’ instead of ‘0px’. Because percentages resolved against indefinite sizes behave as ‘auto’, this gives better behavior in shrink-wrapped flex containers. (Issue 20)

> When omitted from the ‘flex’ shorthand, its specified value is ‘0%’ ~~the length zero~~ .

> **‘flex: <positive-number>’**
>
> Equivalent to ‘flex: <positive-number> 1 0px0%’.

Note: This change was reverted.

- Defined that an unresolvable percentage flex base size is treated as ‘auto’.

> percentage values of ‘flex-basis’ are resolved against the flex item’s containing block, i.e. its flex container, and if that containing block’s size is indefinite, the result is ~~undefined~~ the same as a main size of ‘auto’ .

- Simplified the static position of abspos children of flex containers to be consistent with Grid Layout. (Issue 6)

An absolutely-positioned child of a flex container does not participate in flex layout beyond the reordering step.

~~However, if both 'left' and 'right' or both 'top' and 'bottom' are 'auto', then the used value of those properties are computed from its static position, as follows:~~

~~If both 'left' and 'right' are 'auto', the flex item must be positioned so that its main-start or cross-start edge (whichever is in the horizontal axis) is aligned with the static position. If both 'top' and 'bottom' are 'auto', the flex item must be positioned so that its main-start or cross-start edge (whichever is in the vertical axis) is aligned with the static position.~~

~~In the main axis,~~

1. ~~If there is a subsequent in-flow flex item on the same flex line, the static position is the outer main-start edge of that flex item.~~
2. ~~Otherwise, if there is a preceding in-flow flex item on the same flex line, the static position is the outer main-end edge of that flex item.~~
3. ~~Otherwise, the static position is determined by the value of 'justify-content' on the flex container as if the static position were represented by a zero-sized flex item.~~

~~In the cross axis,~~

1. ~~If there is a preceding in-flow flex item, the static position is the cross-start edge of the flex line that item is in.~~
2. ~~Otherwise, the static position is the cross-start edge of the first flex line.~~

> ~~The static position is intended to more-or-less match the position of an anonymous 0×0 in-flow 'flex-start'-aligned flex item that participates in flex layout, the primary difference being that any packing spaces due to 'justify-content: space-around' or 'justify-content: space-between' are suppressed around the hypothetical item: between it and the next item if there is a real item after it, else between it and the previous item (if any) if there isn't.~~

Its static position is calculated by first doing full flex layout without the absolutely-positioned children, then positioning each absolutely-positioned child as if it were the sole flex item in the flex container, assuming both the child and the flex container were fixed size boxes of their used size.

> EXAMPLE 16
> For example, by default, the static position of an absolutely positioned child aligns it to the main-start/cross-start corner, corresponding to the default values of ‘justify-content’ and ‘align-content’ on the flex container. Setting ‘justify-content:center’ on the flex container, however, would center it in the main axis.

¶ • Changed algorithm for resolving flexible lengths to make behavior continuous as the sum of the flex factors approaches zero. (No change for a sum ≥ 1.) (Issue 30) Replaces this section with this one.

§ Clarifications

The following significant clarifications were also made:

¶ • Absolutely positioned children of a flex container are no longer called "flex items" (to avoid terminology confusion). (??)

> **Name:** order
> **Applies to:** flex items and absolutely-positioned children of flex containers

> Re-order the flex items and absolutely positioned flex container children according to their ‘order’.

¶ • Clarified that ‘float’ still affects the computed ‘display’ (which may affect box-fixup rules that run prior to flex item determination). (Issue 7)

> ‘float’ and ‘clear’ have no effect on a flex item , and do not take it out-of-flow. However, the ‘float’ property can still affect box generation by influencing the ‘display’ property’s computed value.

¶ • Clarify what is meant by “white space”. (Issue 26)

> However, an anonymous flex item that contains only white space (i.e. characters that can be affected by the ‘white-space’ property) is not rendered, as if it were ‘display:none’.

- Clarified that table anonymous box generation occurs in place of computed value conversion for internal table elements.
- Clarified interaction of flex item determination with ‘display-inside’ / ‘display-outside’ (the new longhands of ‘display’ defined in the CSS Display Module Level 3).

> If the specified ‘display-outside’ of an in-flow child of an element that generates a flex container is ‘inline-level’, it computes to ‘block-level’. (This effectively converts any inline ‘display’ values to their block equivalents.)

Note: This change was reverted.

- Clarified that ‘overflow’ applies to flex containers.
- Clarified that ‘::first-line’ and ‘::first-letter’ pseudo-elements do not apply to flex containers (because they are not block containers).
- Clarify that ‘stretch’ checks for the *computed* value of the cross-size property being ‘auto’, which means that percentage cross-sizes that behave as ‘auto’ (because they don’t resolve against definite sizes) aren’t stretched. (Issue 5)

> **stretch**
> If the cross size property of the flex item ~~is~~ computes to ‘auto’, its used value is …

> **Determine the used cross size of each flex item.** If a flex item has ‘align-self: stretch’, its computed cross size property is ‘auto’, and …

- Clarify that the rules of the formatting context are used for determining the flex container’s main size.

> **Determine the main size of the flex container** using the rules of the formatting context in which it participates ~~its main size property~~ .

- Clarified that ‘order’-modified document order is used instead of raw document order when painting. (This was already stated in the ‘order’ section, but not in the section explicitly about painting order.)
- Clarified line-breaking to precisely handle negatively-sized flex items and zero-size items at the end of a line. (Issue 1)

Otherwise, starting from the first uncollected item, collect consecutive items one by one until the first time that the *next* collected item would not fit into the flex container's inner main size, or until a forced break is encountered. If the very first uncollected item wouldn't fit, collect just it into the line ~~as many consecutive flex items as will fit or until a forced break is encountered (but collect at least one) into the flex container's inner main size into a flex line~~ .

Note that ~~items with zero main size will never start a line unless they're the very first items in the flex container, or they're preceded by a forced break.~~ The "collect as many" line will collect ~~them~~ zero-sized flex items onto the end of the previous line even if the last non-zero item exactly "filled up" the line.

¶ • Clarified that flex container cross sizes are still clamped by the flex container's min/max properties. (Issue 24)

- If the cross size property is a definite size, use that, clamped by the min and max cross size properties of the flex container .
- Otherwise, use the sum of the flex lines' cross sizes, clamped by the min and max cross size properties of the flex container .

§ 11. Privacy and Security Considerations

Flexbox 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 17

This is an example of an informative example.

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

Note, this is an informative note.

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

UAs MUST provide an accessible alternative.

§ 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.

§ CR exit criteria

For this specification to be advanced to Proposed Recommendation, there must be at least two independent, interoperable implementations of each feature. Each feature may be implemented by a different set of products, there is no requirement that all features be implemented by a single product. For the purposes of this criterion, we define the following terms:

independent
: each implementation must be developed by a different party and cannot share, reuse, or derive from code used by another qualifying implementation. Sections of code that have no bearing on the implementation of this specification are exempt from this requirement.

interoperable
: passing the respective test case(s) in the official CSS test suite, or, if the implementation is not a Web browser, an equivalent test. Every relevant test in the test suite should have an equivalent test created if such a user agent (UA) is to be used to claim interoperability. In addition if such a UA is to be used to claim interoperability, then there must one or more additional UAs which can also pass those equivalent tests in the same way for the purpose of interoperability. The equivalent tests must be made publicly available for the purposes of peer review.

implementation
: a user agent which:

1. implements the specification.
2. is available to the general public. The implementation may be a shipping product or other publicly available version (i.e., beta version, preview release, or "nightly build"). Non-shipping product releases must have implemented the feature(s) for a period of at least one month in order to demonstrate stability.
3. is not experimental (i.e., a version specifically designed to pass the test suite and is not intended for normal usage going forward).

The specification will remain Candidate Recommendation for at least six months.

§ Index

§ Terms defined by this specification

strut size, in §9.4

target main size, in §9.7

transferred size suggestion, in §4.5

wrap, in §5.2

wrap-reverse, in §5.2

§ Terms defined by reference

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

- alignment baseline
- alignment container
- alignment context
- baseline set
- generate baselines
- justify-self
- last-baseline alignment
- synthesize baseline
- synthesized baseline

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

- computed value
- initial
- used value

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

- vertical-align

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

- overflow
- scroll container
- 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
- first formatted line

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

auto

available space

behave as auto

box-sizing

definite

indefinite

inner size

intrinsic sizing

max-content (for width)

max-content constraint

max-content contribution

max-content size

min-content (for width)

min-content constraint

min-content contribution

min-content size

minimum size

outer size

preferred size

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

white-space

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

text-decoration

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

<integer>

<number>

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

<length-percentage>

?

|

||

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

block axis

block-end

block-start

horizontal-tb

inline axis

inline size

inline-end

inline-start

ltr

rtl

vertical-rl

writing mode

writing-mode

[CSS21] defines the following terms:

clear

float

height

margin

max-height

max-width

min-height

min-width

page-break-after

page-break-before

visibility

width

[CSS3-BREAK] defines the following terms:

break-after

break-before

break-inside

fragmentation container

fragmentation context

[CSS3-DISPLAY] defines the following terms:

- anonymous
- block box
- block container
- block-level
- blockify
- containing block
- display
- establishes an independent formatting context
- flow layout
- inline-level
- text node
- text run

[css3-images] defines the following terms:

- specified size

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

- direction
- end
- start

[cssom-1] defines the following terms:

- resolved value

§ References

§ Normative References

[CSS-ALIGN-3]
Elika Etemad; Tab Atkins Jr.. CSS Box Alignment Module Level 3. 30 August 2018. WD. URL: https://www.w3.org/TR/css-align-3/

[CSS-CASCADE-4]
Elika Etemad; Tab Atkins Jr.. CSS Cascading and Inheritance Level 4. 28 August 2018. CR. URL: https://www.w3.org/TR/css-cascade-4/

[CSS-INLINE-3]
Dave Cramer; Elika Etemad; Steve Zilles. CSS Inline Layout Module Level 3. 8 August 2018. WD. URL: https://www.w3.org/TR/css-inline-3/

[CSS-OVERFLOW-3]
David Baron; Elika Etemad; Florian Rivoal. CSS Overflow Module Level 3. 31 July 2018. 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-SIZING-3]
Tab Atkins Jr.; Elika Etemad. CSS Intrinsic & Extrinsic Sizing Module Level 3. 4 March 2018. WD. URL: https://www.w3.org/TR/css-sizing-3/

[CSS-TEXT-3]
Elika Etemad; Koji Ishii. CSS Text Module Level 3. 20 September 2018. WD. URL: https://www.w3.org/TR/css-text-3/

[CSS-VALUES-3]
Tab Atkins Jr.; Elika Etemad. CSS Values and Units Module Level 3. 14 August 2018. CR. URL: https://www.w3.org/TR/css-values-3/

[CSS-VALUES-4]
Tab Atkins Jr.; Elika Etemad. CSS Values and Units Module Level 4. 10 October 2018. WD. URL: https://www.w3.org/TR/css-values-4/

[CSS-WRITING-MODES-4]
Elika Etemad; Koji Ishii. CSS Writing Modes Level 4. 24 May 2018. CR. URL: https://www.w3.org/TR/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/

[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]
Tab Atkins Jr.; Elika Etemad. CSS Display Module Level 3. 28 August 2018. CR. URL: https://www.w3.org/TR/css-display-3/

[CSS3-IMAGES]
Elika Etemad; Tab Atkins Jr.. CSS Image Values and Replaced Content Module Level 3. 17 April 2012. CR. URL: https://www.w3.org/TR/css3-images/

[CSS3-WRITING-MODES]
Elika Etemad; Koji Ishii. CSS Writing Modes Level 3. 24 May 2018. CR. URL: https://www.w3.org/TR/css-writing-modes-3/

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

[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-TEXT-DECOR-3]

Elika Etemad; Koji Ishii. CSS Text Decoration Module Level 3. 3 July 2018. CR. URL: https://www.w3.org/TR/css-text-decor-3/

[CSS3UI]

Tantek Çelik; Florian Rivoal. CSS Basic User Interface Module Level 3 (CSS3 UI). 21 June 2018. REC. URL: https://www.w3.org/TR/css-ui-3/

[HTML]

Anne van Kesteren; et al. HTML Standard. Living Standard. URL: https://html.spec.whatwg.org/multipage/

§ Property Index

Name	Value	Initial	Applies to	Inh.	%ages	Anim-ation type	Canonical order	Computed value
'align-content'	flex-start | flex-end | center | space-between | space-around | stretch	stretch	multi-line flex containers	no	n/a	discrete	per grammar	specified keyword
'align-items'	flex-start | flex-end | center | baseline | stretch	stretch	flex containers	no	n/a	discrete	per grammar	specified keyword
'align-self'	auto | flex-start | flex-end | center | baseline | stretch	auto	flex items	no	n/a	discrete	per grammar	specified keyword
'flex'	none | [<'flex-grow'> <'flex-shrink'>? || <'flex-basis'>]	0 1 auto	flex items	no	see individual properties	by computed value type	per grammar	see individual properties
'flex-basis'	content | <'width'>	auto	flex items	no	relative to the flex container's inner main size	by computed value type	per grammar	specified keyword or a computed <length-percentage> value
'flex-direction'	row | row-reverse | column | column-reverse	row	flex containers	no	n/a	discrete	per grammar	specified keyword

Name	Value	Initial	Applies to	Inh.	%ages	Anim-ation type	Canonical order	Computed value
'flex-flow'	<'flex-direction'> || <'flex-wrap'>	see individual properties	see individual properties	see individual properties	see individual properties	see individual properties	per grammar	see individual properties
'flex-grow'	<number>	0	flex items	no	n/a	by computed value type	per grammar	specified number
'flex-shrink'	<number>	1	flex items	no	n/a	number	per grammar	specified value
'flex-wrap'	nowrap | wrap | wrap-reverse	nowrap	flex containers	no	n/a	discrete	per grammar	specified keyword
'justify-content'	flex-start | flex-end | center | space-between | space-around	flex-start	flex containers	no	n/a	discrete	per grammar	specified keyword
'order'	<integer>	0	flex items	no	n/a	by computed value type	per grammar	specified integer