

# Fundamentals of Web Development

Third Edition by Randy Connolly and Ricardo Hoar



## Chapter 7

CSS 2: Layout

# Normal Flow

To understand CSS positioning and layout, it is essential that we understand this distinction as well as the idea of **normal flow**:

how the browser will normally display block-level elements and inline elements from left to right and from top to bottom

# Normal Flow

- **Block-level elements** such as `<p>`, `<div>`, `<h2>`, `<ul>`, and `<table>` are each contained on their own line.
- **Inline elements** do not form their own blocks but instead are displayed within lines.

# Normal Flow

Block-Level Elements



Each block exists on its own line and is displayed in normal flow from the browser window's top to its bottom.

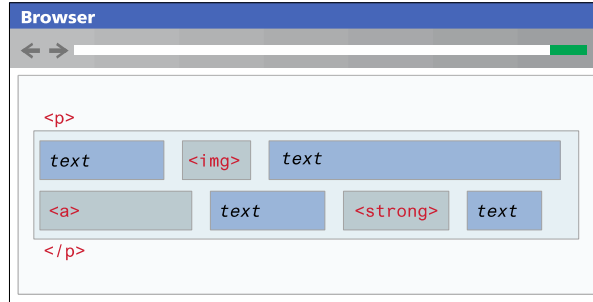
By default each block-level element fills up the entire width of its parent (in this case, it is the `<body>`, which is equivalent to the width of the browser window).

You can use CSS box model properties to customize, for instance, the width of the box and the margin space between other block-level elements.

# Normal Flow

Inline Elements

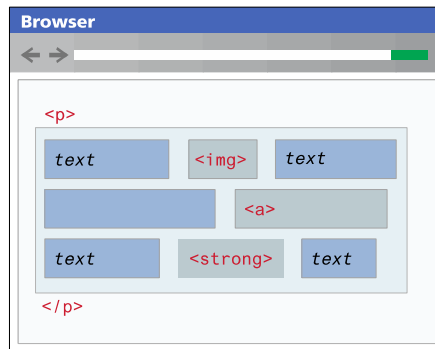
```
<p>  
This photo  of Conservatory Pond in  
<a href="http://www.centralpark.com/">Central Park</a> New York City  
was taken on October 22, 2015 with a <strong>Canon EOS 30D</strong>  
camera.  
</p>
```



Inline content is laid out horizontally left to right within its container.

Once a line is filled with content, the next line will receive the remaining content, and so on.

Here the content of this <p> element is displayed on two lines.

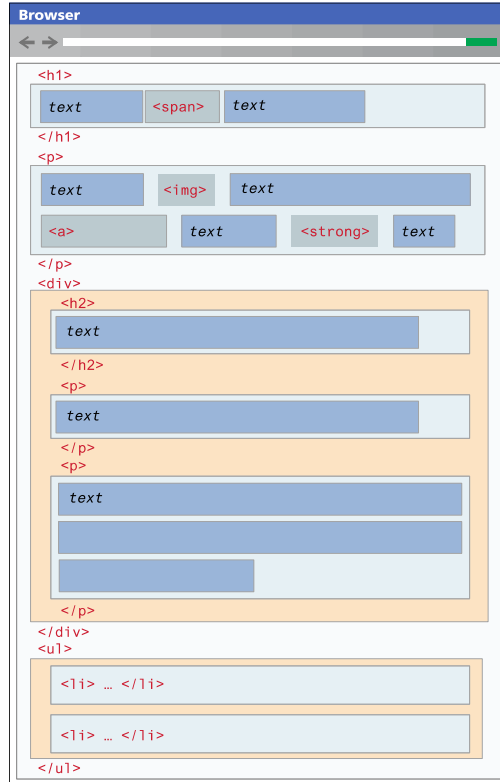


If the browser window resizes, then inline content will be "reflowed" based on the new width.

Here the content of this <p> element is now displayed on three lines.

# Normal Flow

Block and inline elements



A document consists of block-level elements stacked from top to bottom.

Within a block, inline content is horizontally placed left to right.

Some block-level elements can contain other block-level elements (in this example, a <div> can contain other blocks).

In such a case, the block-level content inside the parent is stacked from top to bottom within the container (<div>).

# Floating Elements

It is possible to displace an element out of its position in the normal flow via the CSS **float** property.

An element can be floated to the **left** or floated to the **right** and content is “reflowed” around the floated element

A floated block-level element should have a width specified



```
<h1>Float example</h1>
<p>A wonderful serenity has taken ...</p>
<figure>
  
  <figcaption>British Museum</figcaption>
</figure>
<p>When, while the lovely valley...
```

```
figure {
  border: 1px solid #A8A8A8;
  background-color: #EDEDDE;
  margin: 0;
  padding: 5px;
  width: 150px;
}
```

Notice that a floated block-level element should have a width specified.

```
figure {
  ...
  width: 150px;
  float: left;
}
```

```
figure {
  ...
  width: 150px;
  float: right;
  margin: 10px;
}
```

# Positioning Elements

The **position** property is used to specify the type of positioning. The **left**, **right**, **top**, and **bottom** properties are used to indicate the distance the element will move.

- **absolute** The element is removed from normal flow and positioned in relation to its nearest positioned ancestor.
- **fixed** The element is fixed in a specific position in the window even when the document is scrolled.
- **relative** The element is moved relative to where it would be in the normal flow.
- **static** The element is positioned according to the normal flow. **This is the default.**
- **sticky** The element is positioned in according to the normal flow, and then offset relative to its nearest scrolling ancestor. This is used to allow an item to scroll, and then stay fixed in position once its scroll position is reached.



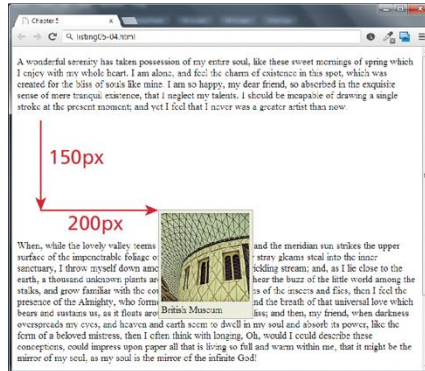
# Relative Positioning



<p>A wonderful serenity has taken ... </p>

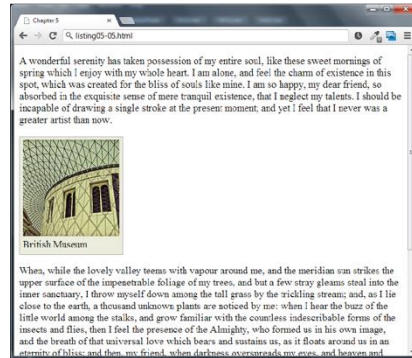
```
<figure>
  
  <figcaption>British Museum</figcaption>
</figure>
```

<p>When, while the lovely valley ...



```
figure {
  border: 1pt solid #A8A8A8;
  background-color: #EDEDDE;
  padding: 5px;
  width: 150px;
  position: relative;
  top: 150px;
  left: 200px;
}
```

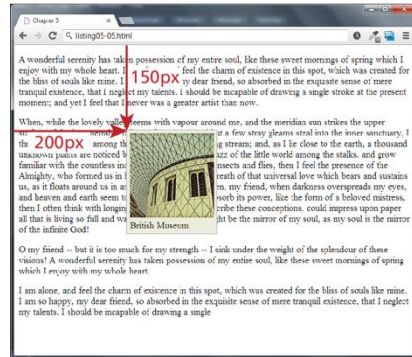
# Absolute positioning



<p>A wonderful serenity has taken possession of my ...

```
<figure>
  
  <figcaption>British Museum</figcaption>
</figure>
```

<p>When, while the lovely valley ...

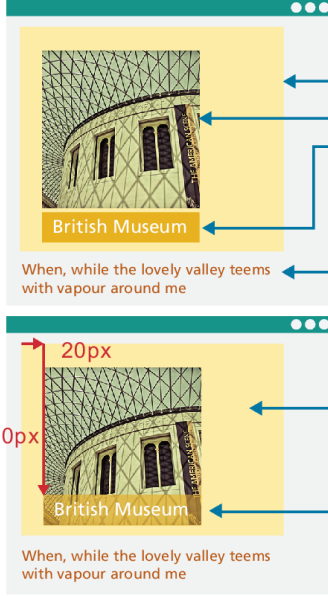


```
figure {
  margin: 0;
  border: 1pt solid #A8A8A8;
  background-color: #EDEDDE;
  padding: 5px;
  width: 150px;
  position: absolute;
  top: 150px;
  left: 200px;
}
```

# Overlapping and Hiding Elements

One of the more common design tasks with CSS is to place two elements on top of each other, or to selectively hide and display elements.

In such a case, relative positioning is used to create the **positioning context** for a subsequent absolute positioning move.



The diagram shows a browser window with a yellow background. Inside, there is a curved building image (British Museum) and a caption below it. The image is positioned relative to its container, and the caption is positioned absolutely within that context. Red arrows indicate the dimensions of the positioning context: 20px for the width and 130px for the height.

```
<figure>  
    
  <figcaption>British Museum</figcaption>  
</figure>  
<p>When, while the lovely valley teems ...
```

```
figure {  
  ...  
  position: relative;  
}  
figcaption {  
  ...  
  position: absolute;  
  top: 130px;  
  left: 20px;  
}
```

This creates the positioning context.


This does the actual move.

# Overlapping and Hiding Elements (ii)

Consider an image that is the same size as the underlying one is placed on top of the other image using absolute positioning.

You can hide this image using the **display** property


```
<figure>
  
  <figcaption>British Museum</figcaption>
  
</figure>
```



new-banner.png

Transparent area

```
.overlaid {
  position: absolute;
  top: 10px;
  left: 10px;
}
```



When, while the lovely valley teems with vapour around me

British Museum

When, while the lovely valley teems with vapour around me

British Museum

```
.hide {
  display: none;
}
```

This is the preferred way to hide: by adding this additional class to the element.


```

```

# Older Approaches to CSS Layout

- The display property in CSS provides a mechanism for the developer to change an element to block, inline, or inline-block.
- For the first 20 years of CSS, designers had to “hack” together multi column layouts using **floats** and/or **positioning**. There may be times when you may have to support legacy CSS so it makes sense to learn the basics of floats and positioning.
- Newer approaches (flexbox and grid display modes) make columnar layouts much easier to implement.

# Flexbox Layout



### Fall in Calgary

Nunc nec fermentum dolor. Duis at iaculis turpis. Sed rutrum elit ac egestas dapibus. Duis nec consequat enim.

Mauris porta arcu id magna adipiscing lacinia at congue lacus. Vivamus blandit quam quis tincidunt egestas. Etiam posuere lectus sed sapien malesuada molestie.

Phasellus vel felis purus. Aliquam consequat pellentesque dui, non mollis erat dictum sit amet. Curabitur non quam dictum, consectetur arcu in, vehicula justo.


← float

margin-left  
= image size + margin-right

```
.media-image {  
  float: left;  
  margin-right: 10px;  
}  
.media-body {  
  margin-left: 160px;  
}
```

```
<div class="media">  
    
  <div class="media-body">  
    <h2>Fall in Calgary</h2>  
    <p>Nunc nec fermentum dolor...</p>  
    <p>Mauris porta arcu id...</p>  
    <p>Phasellus vel felis purus...</p>  
  </div>  
</div>
```

Prior to flexbox, one would create such a layout within a container using floats plus margins. The problem with this approach is that margins needed to be in pixels and had to exactly match image size. If image size changed (or you wanted same kind of style elsewhere), you had to modify the style.



### Fall in Calgary

Nunc nec fermentum dolor. Duis at iaculis turpis. Sed rutrum elit ac egestas dapibus. Duis nec consequat enim.

Mauris porta arcu id magna adipiscing lacinia at congue lacus. Vivamus blandit quam quis tincidunt egestas. Etiam posuere lectus sed sapien malesuada molestie.

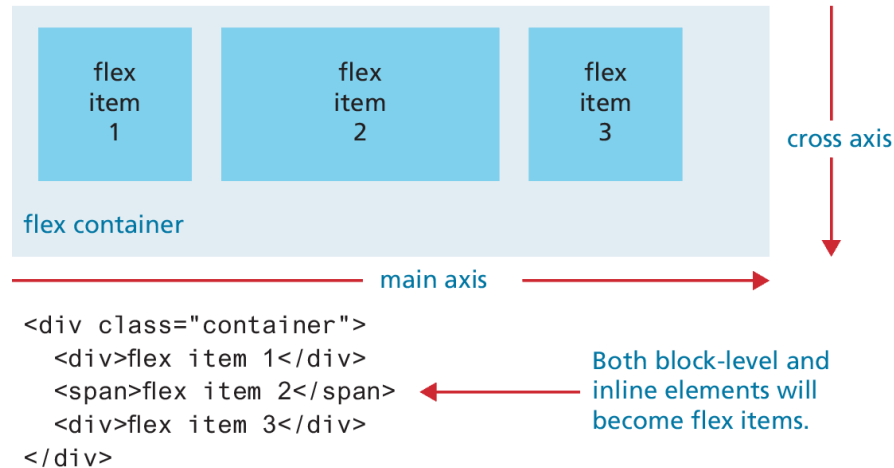
Phasellus vel felis purus. Aliquam consequat pellentesque dui, non mollis erat dictum sit amet. Curabitur non quam dictum, consectetur arcu in, vehicula justo.

```
.media {  
  display: flex;  
  align-items: flex-start;  
}  
.media-image {  
  margin-right: 1em;  
}
```

Using flexbox, we now have a much more generalized (and thus reusable) style.

# Flex containers and items

There are two places in which you will be assigning flexbox properties: the **flex container** and the **flex items** within the container.



# The flexbox container properties

display: **flex**      Necessary to use flexbox

flex-direction: **row**



Specifies the direction of the main axis.  
The default is **row**.

flex-direction: **row-reverse**

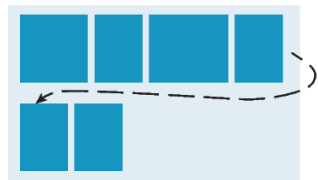


flex-direction: **column**

flex-direction: **column-reverse**



flex-wrap: **wrap**



flex-wrap: **nowrap**



Indicates whether new lines should be created if flex container can't contain all the flex items.



# The flexbox container properties (ii)

`justify-content: flex-start`

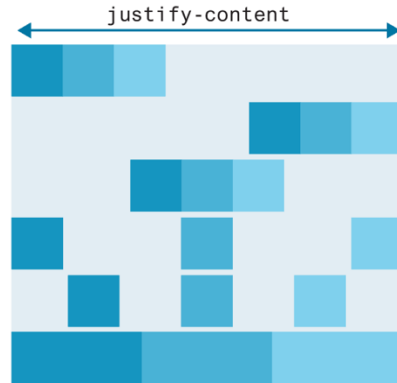
`justify-content: flex-end`

`justify-content: center`

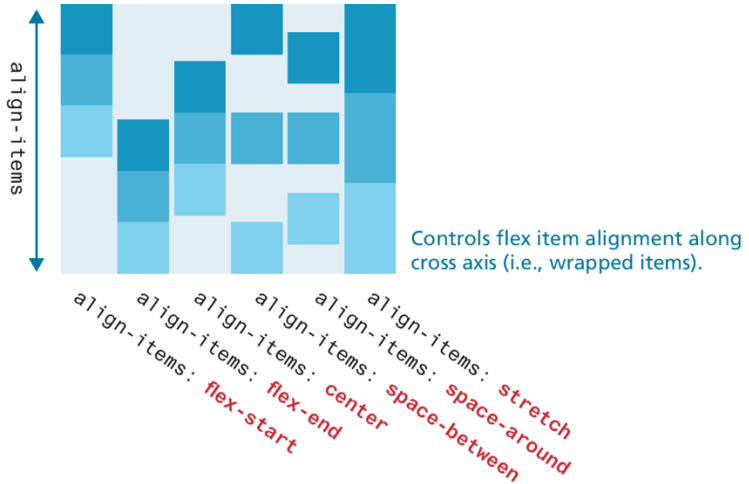
`justify-content: space-between`

`justify-content: space-around`

`justify-content: stretch`



Controls flex item alignment along main axis.



Controls flex item alignment along cross axis (i.e., wrapped items).

# The flexbox item (child) properties

flex-basis: **auto**



The flex-basis property determines the initial size of the flex item before the remaining space is distributed.

The default auto value means that the size is determined by the width and height.

flex-basis: **200px**



You can specify a width using px, %, or other measurement units.

flex-grow: **2**



$\text{width} = n$

$\text{width} = n \times 2$

Defines the growth factor of an element relative to the other items.

flex-shrink: **2**



Defines how much an item will shrink when not enough space in container.

# Flexbox Cards



```
<div class="card">
  <div class="content">
    
    <h3>Albert Hall</h3>
  </div>
  <footer>
    <a href="#">View</a>
  </footer>
</div>
```



```
.card {
  ...
  display: flex;
  flex-direction: column;
}

.card .content {
  flex: 1 1 auto;
}
```

# Grid Layout

**Grid layout** is adjustable, powerful, and, compared to floats, positioning, and even flexbox, is relatively easy to learn and use!

- Each blocklevel child in a parent container whose **display** property is set to **grid** will be automatically placed into a grid cell

By default, a grid container will behave like any container in that each block element will be on its own line (or row).

10px gap

grid cell

```
<div class="container">
  <div>A</div>
  <div>B</div>
  <div>C</div>
  ...
</div>
```

Grid container

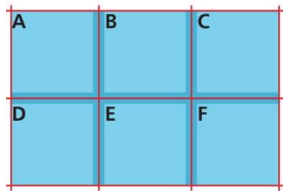
The container's block-level children will become the grid items.

```
.container {
  display: grid;
  gap: 10px;
}
```

To make each cell more visually distinguishable, we have specified a gap, which adds space around each cell.

# Specifying Grid Structure

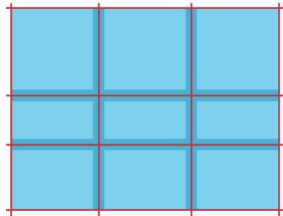
**grid-template-columns** is used for adding columns by specifying each column's width using the **fr** unit.



```
.container {  
  display: grid;  
  gap: 10px;  
  grid-template-columns: 1fr 1fr 1fr;  
}
```

This makes each column equal to an equal fraction of the available space.

You can specify the number of columns per row/line via the grid-template-column property.



```
.container {  
  ...  
  height: 300px;  
  grid-template-columns: 1fr 1fr 1fr;  
  grid-template-rows: 80px 30px 50px;  
}
```

While you often do not need to set a row height, you can make rows taller than their actual content.

# Specifying column widths

Column widths can be specified

The CSS **repeat()** function provides a way to specify repeating patterns of columns.

```
grid-template-columns: 70px 70px 45px;
```



Each column can have its own unique width value, in px, %, em, etc.

```
grid-template-columns: 50px auto 50px;
```



An auto value indicates the width will fill the remaining space.

```
grid-template-columns: repeat(3, 1fr);
```



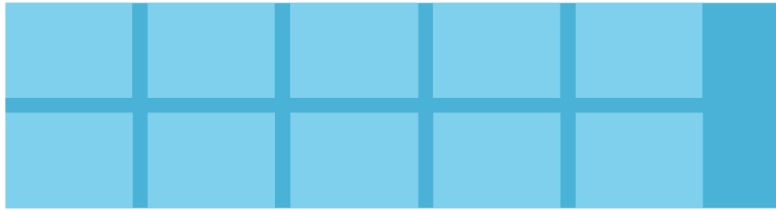
The repeat function can be used to defining a repeating pattern.

```
grid-template-columns: repeat(2, 30px 50px) 70px;
```



30px 50px 30px 50px 70px

# Specifying column widths (ii)



```
grid-template-columns: repeat(auto-fill, minmax(100px, 1fr));
```

Fills the row with as many columns as can fit into the container space.

The min column size is 100px and the max is whatever size is necessary to allow each column to be equal sized.



```
grid-template-columns: repeat(auto-fit, minmax(100px, 1fr));
```

Fits the columns into space by expanding the column width into the available container space.

# Contrasting grid approaches

## **<!-- Bootstrap 4 Approach -->**

```
<div class="container">
  <div class="row">
    <div class="col"><img src=1.gif /></div>
    <div class="col"><img src=2.gif /></div>
    <div class="col"><img src=3.gif /></div>
  </div>
  <div class="row">
    <div class="col"><img src=4.gif /></div>
    <div class="col"><img src=5.gif /></div>
    <div class="col"><img src=6.gif /></div>
  </div>
</div>
```

## **<!-- CSS Grid Approach -->**

```
<div class="container">
  <img src=1.gif />
  <img src=2.gif />
  <img src=3.gif />
  <img src=4.gif />
  <img src=5.gif />
  <img src=6.gif />
</div>
```

## **<!-- CSS for grid approach -->**

```
.container {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(100px, 1fr));
}
.container img { display: block; }
```

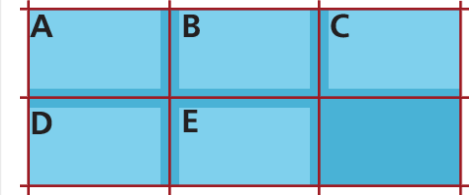
**LISTING 7.1** Comparing Bootstrap grid with CSS Grid



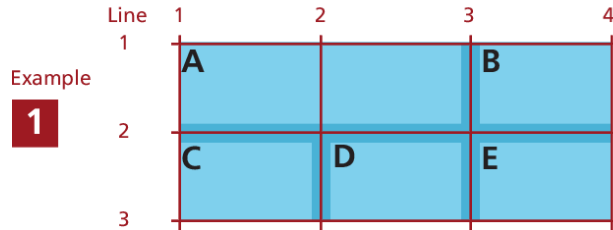
# Explicit Grid Placement Example 1

```
<div class="container">
  <div class="a">A</div>
  <div class="b">B</div>
  <div class="c">C</div>
  <div class="d">D</div>
  <div class="e">E</div>
</div>
```

```
.container {
  display: grid;
  gap: 10px;
  grid-template-columns: repeat(3,1fr);
  grid-template-rows: repeat(2,200px);
}
```



With implicit layout, grid items are placed automatically.



```
.a {
  grid-column-start: 1;
  grid-column-end: 3;
}
```

The start and end numbers refer to the line number not the column number.

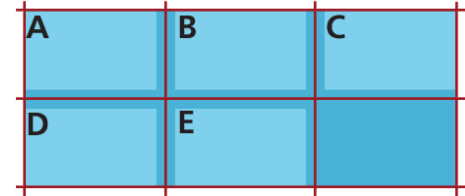
The same effect also possible using either of the following:

```
grid-column: 1 / 3;
grid-column: 1 / span 2;
```

# Explicit Grid Placement Example 2

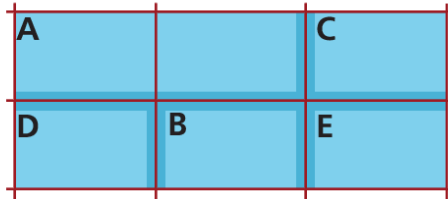
```
<div class="container">
  <div class="a">A</div>
  <div class="b">B</div>
  <div class="c">C</div>
  <div class="d">D</div>
  <div class="e">E</div>
</div>
```

```
.container {
  display: grid;
  gap: 10px;
  grid-template-columns: repeat(3,1fr);
  grid-template-rows: repeat(2,200px);
}
```



With implicit layout, grid items are placed automatically.

2

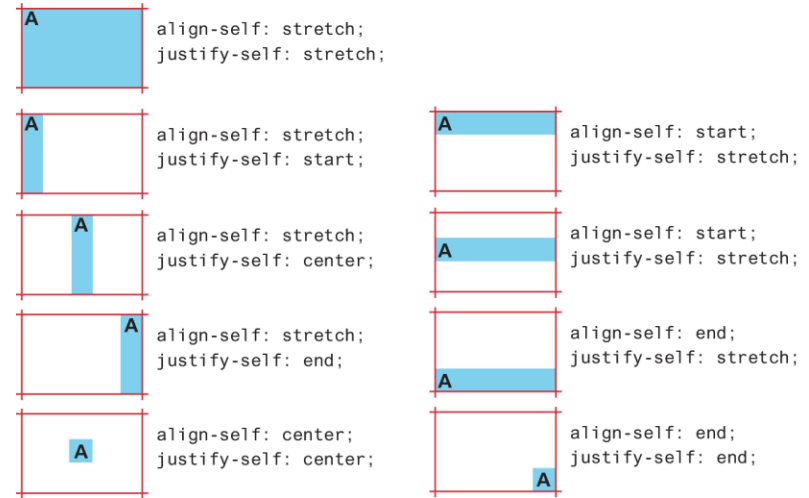


```
.b {
  grid-row: 2;
  grid-column: 2;
}
```

Grid cells can be placed into any row and column.

# Cell properties

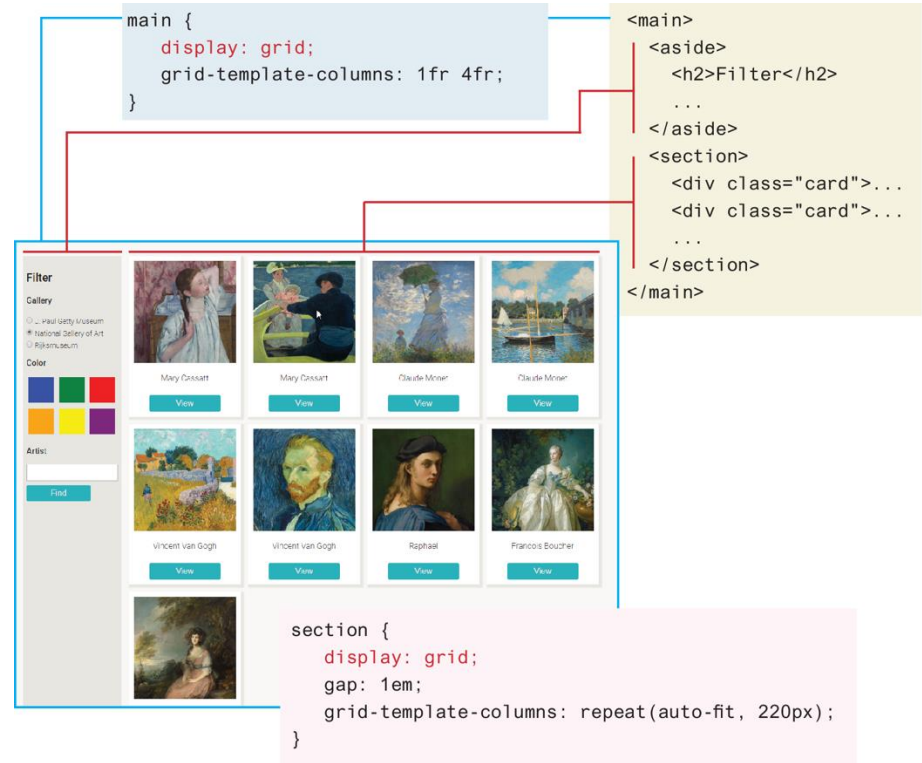
- **align-self** and **justify-self** control the cell content's horizontal and vertical alignment within its grid container.



- You can similarly control cell alignment within a grid container using **align-items** and **justify-items**

# Nested Grid

- **align-self** and **justify-self** control the cell content's horizontal and vertical alignment within its grid container.

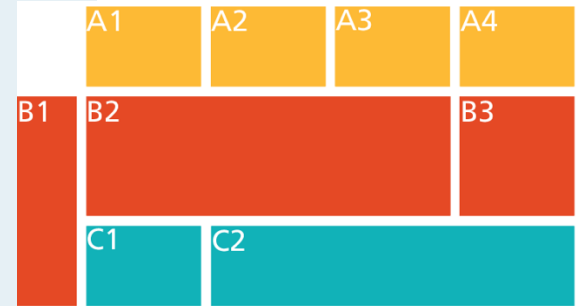


# Grid Areas

```
<style>
.container {
  grid-gap: 10px;
  display: grid;
  grid-template-rows: 100px 150px 100px;
  grid-template-columns: 75px 1fr 1fr 1fr 1fr;
  grid-template-areas: ". a1 a2 a3 a4"
                      "b1 b2 b2 b2 b3"
                      "b1 c1 c2 c2 c2";
}
.a1 { grid-area: a1; }
.a2 { grid-area: a2; }
.a3 { grid-area: a3; }
.a4 { grid-area: a4; }
.b1 { grid-area: b1; }
.b2 { grid-area: b2; }
.b3 { grid-area: b3; }

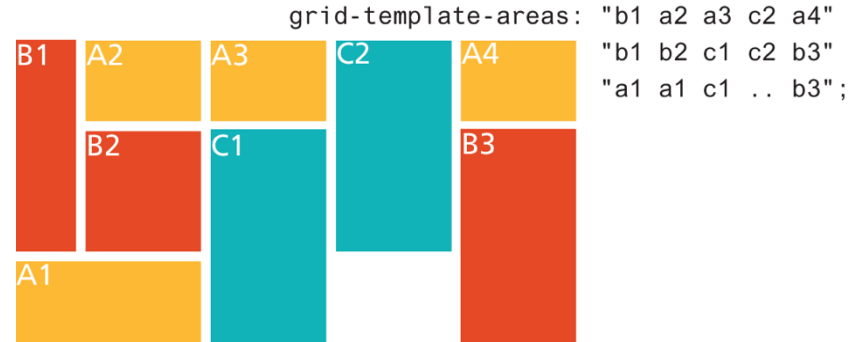
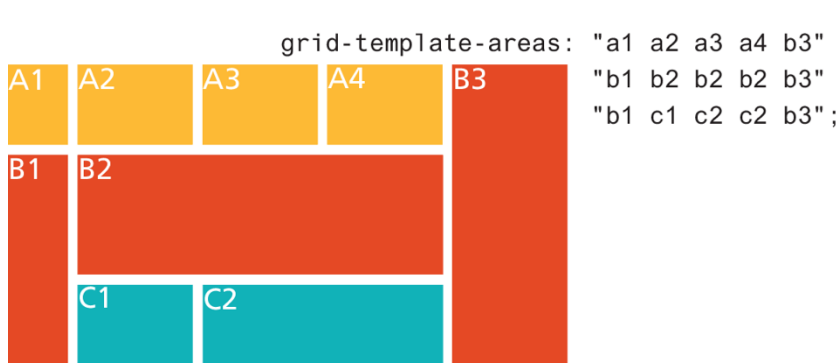
.c1 { grid-area: c1; }
.c2 { grid-area: c2; }
</style>

<section class="container">
  <div class="yellow a1">A1</div>
  <div class="yellow a2">A2</div>
  <div class="yellow a3">A3</div>
  <div class="yellow a4">A4</div>
  <div class="orange b1">B1</div>
  <div class="orange b2">B2</div>
  <div class="orange b3">B3</div>
  <div class="cyan c1">C1</div>
  <div class="cyan c2">C2</div>
</section>
```



**LISTING 7.2** Using grid areas

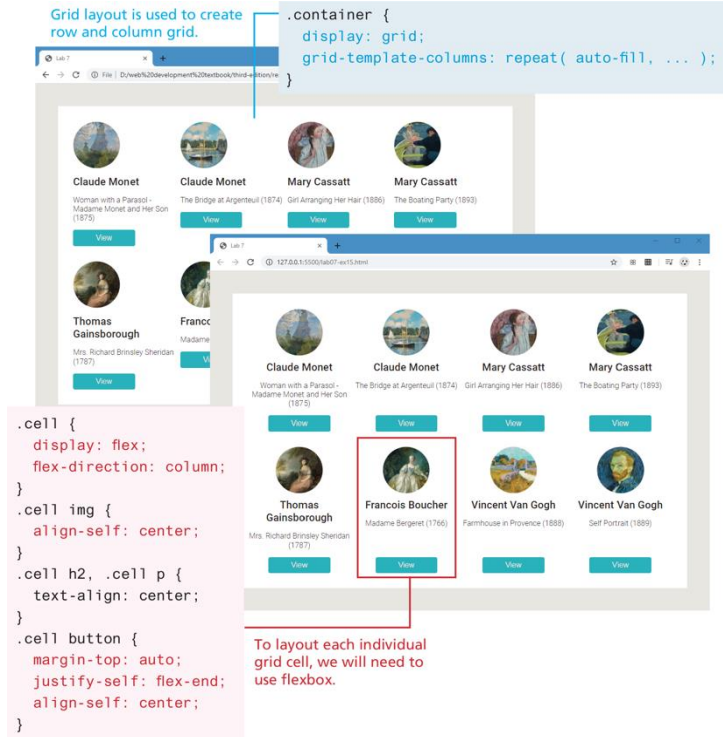
# Grid Areas (ii)



**LISTING 7.2** Using grid areas

# Grid and Flexbox Together

- **grid** and **flexbox** each have their strengths and these strengths can be combined
- **grid** layout is ideal for constructing the layout structure of your page
- **flexbox** is ideal for laying out the contents of a grid cell.



# CSS Effects

Four CSS3 modules have become broadly popular amongst designers:

- transformations,
- filters,
- transitions, and



# Transforms

CSS **transforms** provide additional ways to change the size, position, and even the shape of HTML elements.

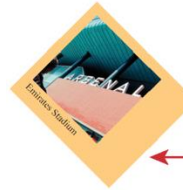
CSS transforms allow you to **rotate**, **skew**, **transform** (move), and **scale** an element.

It is also possible to transform an element in 3D space using the `perspective()`, `rotate3d()`, `scale3d()`, and `translate3d()` functions

```
<figure>
  
  <figcaption>Emirates Stadium</figcaption>
</figure>
```



```
figure {
  padding: 1em;
  background: #FFCC80;
  width: 200px;
}
```



```
figure {
  transform: rotate(45deg);
}
```

Notice that the transform affects all the content within the transformed container.

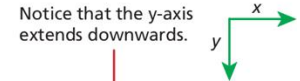


```
figure {
  transform: skew(-20deg);
}
```



```
figure img {
  transform: translateX(100px) translateY(-30px);
}
```

You can combine transforms.



```
figure {
  transform: rotate(15deg);
}
figure img {
  transform: rotate(45deg) scale(0.5);
}
```

# Filters

**Filters** provide a way to modify how an image appears in the browser. Filters are specified by using the filter property and then one or more filter functions

```
#someImage {  
  filter: grayscale(100%);  
}  
#anotherImage {  
  /* multiple filters are space separated */  
  filter: blur(5px) hue-rotate(60deg) saturate(2);  
}
```

**LISTING 7.4** Using a Filter



original



saturate(3)



grayscale(100%)



contrast(200%)



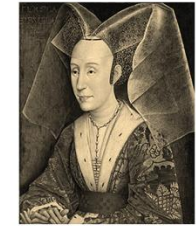
brightness(30%)



blur(3px)



invert(100%)



sepia(100%)



huerotate(90deg)



opacity(50%)



brightness(1.5)  
contrast(3)  
grayscale(0.6)  
invert(0.23)  
sepia(0.2)



brightness(1.3)  
contrast(1.1)  
hue-rotate(180deg)  
saturate(2)

# Transitions

Transitions are a powerful new feature of CSS3. **Transitions** will take effect across a length of time. provide a way to indicate that a property change

- **transition-delay** The delay time in seconds before the animation begins.
- **transition-duration** How long in seconds for the transition to complete.
- **transition-property** The name of the CSS property to which the transition is applied.
- **Transition-timing-function** The function that defines how the intermediate steps in the transition are calculated. CSS defines a variety of different easing functions which define the rate of the transition.

# Transition with example

1. The CSS property which will be transitioned.
2. The duration of the transition.
3. The easing function to use, which changes the speed and style of the transition(optional).
4. How long to delay before starting the transition (optional).

The button as it normally appears.

The button as it appears during transition between two states.

The button as it appears when hovered over.



```
button {
```

```
  background-color: #146d37;
```

Which CSS property of the button is going to be transitioned across time?

```
1 transition-property: background-color;
```

We will transition the background color of the button across time.

```
2 How long is the transition?
```

```
  transition-duration: 0.5s;
```

The transition will last half a second.

```
3 What will be the rate of transition change?
```

```
  transition-timing-function: ease-out;
```

The transition will slow down towards the end.

```
4 Do we delay the start of the transition?
```

```
  transition-delay: 0s;
```

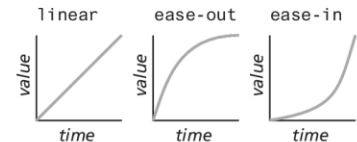
No delay (transition will start immediately)

```
}
```

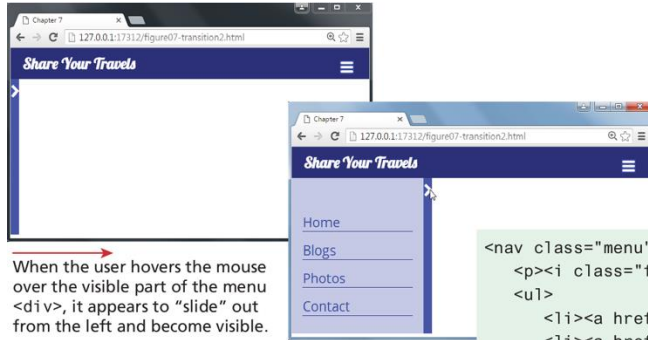
```
button:hover {
```

```
  background-color: #60b946;
```

This light green color will be displayed when the mouse hovers over the button.



# A sliding menu transition example



When the user hovers the mouse over the visible part of the menu <div>, it appears to “slide” out from the left and become visible.

Menu is initially hidden by being positioned outside the visible area.

```
.menu {  
  position: absolute;  
  left: -210px;  
}
```

```
<nav class="menu">  
  <p><i class="fa fa-chevron-right"></i></p>  
  <ul>  
    <li><a href="#">Home</a></li>  
    <li><a href="#">Blogs</a></li>  
    <li><a href="#">Photos</a></li>  
    <li><a href="#">Contact</a></li>  
  </ul>  
</nav>
```

```
.menu:hover {  
  left: 0;
```

When the user hovers over the menu, move the left edge of the element to left edge of the browser (i.e., it will now be visible).

```
  transition: left .6s ease-out;
```

```
}
```

Using the transition shorthand property. Transition the left property across 0.6 seconds and use the ease-out function (i.e., slow down transition at end).

```
.menu {  
  transition: left .6s ease-out;
```

We want the same transition when the mouse is no longer hovering over the menu. This creates illusion of menu sliding back out of sight.

```
}
```