Initial value: none

Applies to: Block-level elements

Inherited: No

The `clear` property may be applied only to block elements. It is best explained with a simple example. The `left` value starts the element below any elements that have been floated to the left edge of the containing block. The rule in this example ensures that all first-level headings in the document start below left-floated elements, as shown in Figure 21-7.

```
img {float: left; margin-right: 10px; }
h1 {clear: left; top-margin: 2em;}
```

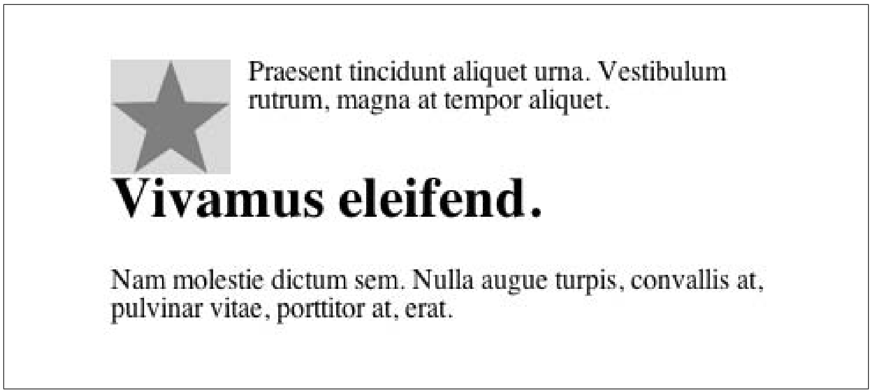
Praesent tincidunt aliquet urna. Vestibulum rutrum, magna at tempor aliquet.
Vivamus eleifend.
Nam molestie dictum sem. Nulla augue turpis, convallis at, pulvinar vitae, porttitor at, erat.

Figure 21-7. Clearing a left-floated element

As you might guess, the `right` value works in a similar manner and prevents an element from appearing next to an element that has been floated to the right. The value `both` moves the element down until it is clear of floated elements on both sides. User agents are instructed by CSS 2.1 to add an amount of *clearance* space above the margins of block elements until the top edge of the content fits below the float.

Notice in Figure 21-7, that although there is a top margin applied to the `h1` element, the text is touching the bottom of the floated image. That is a result of collapsing vertical margins on the `h1` block element. If you want to be sure that there is space below a floated element, add a bottom margin to the float itself, because margins on floated elements never collapse. This remains true when a floated element is set to clear other floated elements on the same side of the page. In that case, adjacent margins of the floated elements add up and don't collapse.

Positioning Basics

It is obvious by how readily web designers co-opted HTML tables that there was a need for page-like layout on web pages. Cascading Style Sheets provides several methods for positioning elements on the page relative to where they would normally appear in the document flow.

If you thought tables were tricky to manage, get ready for CSS positioning! While the positioning properties are fairly simple at face value, inconsistent and buggy browser implementation can make it challenging to achieve the results you're after on all browsers. If fact, positioning can be complicated even when the CSS Recommendation is followed to the letter. It's a recipe for frustration unless you get to know how positioning *should* behave and then know which browsers are likely to give you trouble (some notorious browser bugs are listed in Chapter 25). This section introduces the positioning-related properties as they are defined in CSS 2.1 as well as some key concepts.

Types of Positioning

To get the ball rolling, we'll look at the various options for positioning elements and how they differ. There are four types of positioning, specified by the `position` property.

position

Values:	`static	relative	absolute	fixed	inherit`
Initial value:	`static`
Applies to:	All elements
Inherited:	No

The `position` property identifies that an element is to be positioned and selects one of four positioning methods (each will be discussed in detail in upcoming sections in this chapter):

`static`
: This is the normal positioning scheme in which element boxes are rendered in order as they appear in the document flow.

`relative`
: Relative positioning moves the element box, but its original space in the document flow is preserved.

`absolute`
: Absolutely positioned objects are completely removed from the document flow and are positioned relative to their containing block (discussed in the next section). Because they are removed from the document flow, they no longer influence the layout of surrounding elements, and the space they once occupied is closed up. Absolutely positioned elements always take on block behaviors.

`fixed`
: Fixed positioning is like absolute positioning (the element is removed from the document flow), but instead of a containing element, it is positioned relative to the viewport (in most cases, the browser window).

Containing Blocks

The CSS 2.1 Recommendation states that "The position and size of an element's box(es) are sometimes calculated relative to a certain rectangle, called the *containing block* of the element." It is critical to have an awareness of the containing block for the element you want to position.

Unfortunately, it's not entirely straightforward and depends on the context of the element. CSS 2.1 lays out a number of rules for determining the containing block.

- The containing block created by the root element (`html`) is called the *initial containing block*. The rectangle of the initial containing block fills the dimensions of the viewport. The initial containing block is used if there is no other containing block present. Note that some browsers base the initial containing block on the `body` element; the net result is the same in that it fills the browser window.
- For elements (other than the root) that are set to `static` or `relative`, the containing block is the *content edge* of the nearest block-level, table cell, or inline-block ancestor.
- For absolutely placed elements, the containing block is the nearest ancestor element that has a position other than `static`. In other words, the ancestor element must be set to `relative`, `absolute`, or `fixed` to act as a containing block for its children. Once an ancestor element is established as the containing block, its boundaries differ based on whether it is a block-level or inline element.
- For block-level elements, the containing block extends to the element's *padding edge* (just inside the border).
- For inline-elements, the containing block is set to the *content edge*. Its boundaries are calculated based on the direction of the text. For left-to-right languages, it begins in the top-left corner of the first line generated by the element and ends in the bottom-right corner of the last line generated by the element. For right-to-left languages, it goes from top-left corner of the first line to bottom-left corner of the last line.
- If there are no ancestor elements, then the initial containing block is used.

Specifying Position

Once the positioning value has been established, the actual positioning is done with the four offset properties.

top, right, bottom, left

Values: `<length> | <percentage> | auto | inherit`

Initial value: `auto`

Applies to: Positioned elements (where position value is `relative`, `absolute`, or `fixed`)

Inherited: No

The values provided for each of the offset properties defines the distance that the element should be offset from that edge. For instance, the value of `top` defines the distance from the outer edge of the positioned element to the top edge of its containing block. Positive values move the element down (toward the center of the block); negative values move the element up (and out of the containing block). Similarly, the value provided for the `left` property specifies a distance from the left edge of the containing block to the left outer edge of the positioned element. Again, positive values push the element in toward the center of the containing block while negative values move the box outward.

CSS 2 positioned elements from their content edges, not their margin edges, but this was changed in 2.1.

This rather verbose explanation should be made clearer with a few examples of absolutely positioned elements. In this example, the positioned element is placed in the bottom-left corner of the containing block using percentage values (Figure 21-8).

```
div {position: absolute; height: 120px; width: 300px; border: 1px solid
#000;}
img {position: absolute; top: 100%; left: 0%;}
```

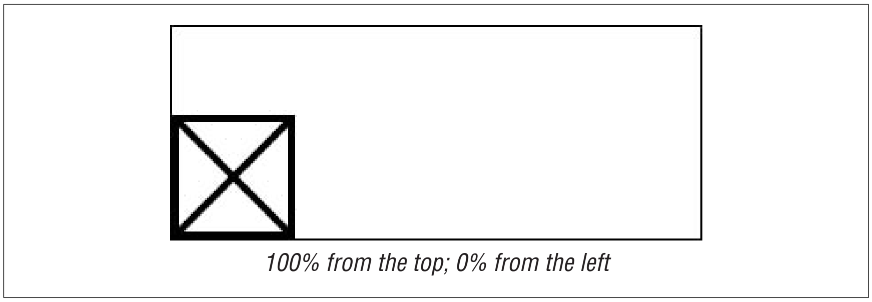

Figure 21-8. Positioning with percentage values

In this example, pixel lengths are provided to place the positioned element at a particular spot in the containing element (Figure 21-9).

```
div.a {position: absolute; height: 120px; width: 300px; border: 1px solid
#000; background-color:#CCC}

div.b {position: absolute; top: 20px; right: 30px; bottom: 40px; left: 50px;
border: 1px solid #000; background-color:#666}

<div class="a">
```

```
    <div class="b"></div>
</div>
```

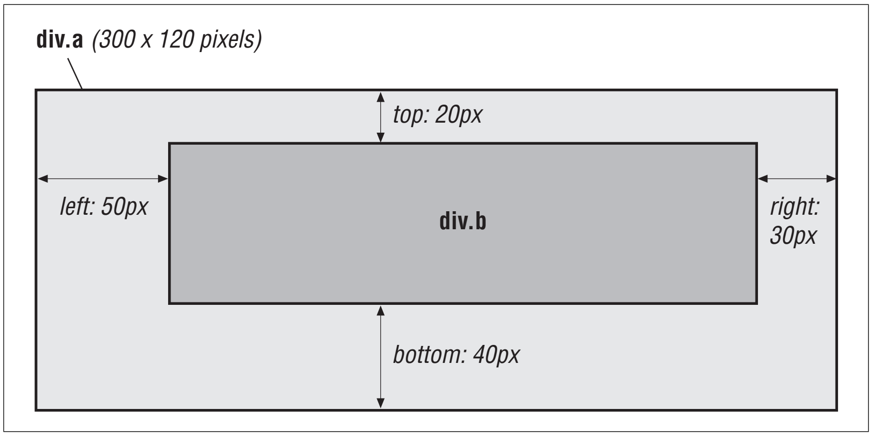

Figure 21-9. Positioning with pixel values

Notice that it is possible to set the dimensions of an element indirectly by defining the positions of its four sides relative to the containing block. The space that is leftover becomes the width and height of the element. If the positioned element also has specified `width` and `height` properties that conflict with that space, a set of CSS rules kicks in for settling the difference (these are addressed in the upcoming "Calculating Position" section).

Setting the `width` and `height` of elements is covered in Chapter 19.

This final example demonstrates that when negative values are provided for offset properties, the element can break out of the confines of the containing box (Figure 21-10).

```
div.a {position: absolute; height: 120px; width: 300px; border: 1px solid
#000; background-color:#CCC}

div.b {position: absolute; top: -20px; right: -30px; bottom: 40px; left:
50px; border: 1px solid #000; background-color:#666}

<div class="a">
    <div class="b"></div>
</div>
```

Handling Overflow

When an element is set to a size that is too small to contain all of its contents, it is possible to specify what to do with the content that doesn't fit using the `overflow` property.

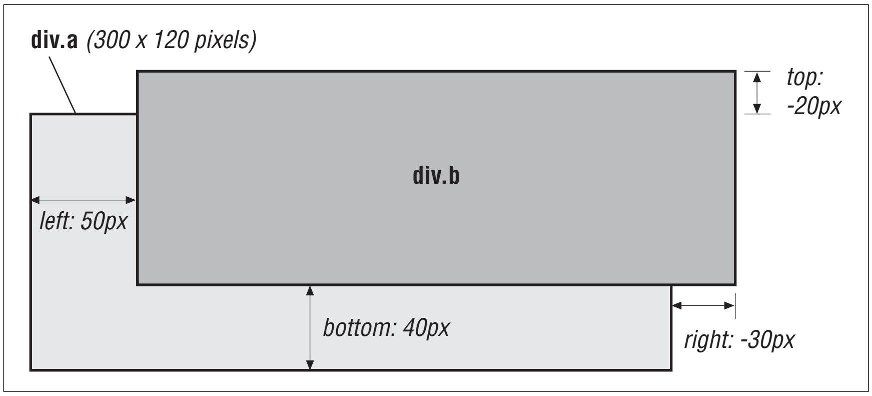

Figure 21-10. Negative offset values

overflow

Values:	`visible | hidden | scroll | auto | inherit`
Initial value:	`visible`
Applies to:	Block-level and replaced elements
Inherited:	No

There are four values for the `overflow` property:

`visible`
: The default value is `visible`, which allows the content to display outside its element box.

`hidden`
: When `overflow` is set to `hidden`, the content that does not fit in the element box gets clipped and does not appear beyond its edges.

`scroll`
: When `scroll` is specified, scrollbars (or an alternate scrolling mechanism) are added to the element box to allow scrolling through the content while keeping the content visible in the box area only. Be aware that the `scroll` value causes scrollbars to be rendered even if the content fits comfortably in the content box.

`auto`
: The `auto` value allows the user agent to decide how to handle overflow. In most cases, scrollbars are added only when the content doesn't fit and they are needed.

Figure 21-11 shows examples of each of the overflow values as applied to an element that is 150 pixels square. The gray background color makes the edges of the content area clear.