Building a Web Browser Engine: Layout & Rendering

The Box Model, Block Layout, Line-Breaking & a Real Software Rasterizer — From Scratch

Topics covered:

The layout tree & the box model · block layout & constraint-based width

Inline layout, line boxes & real font metrics

Painting, a software rasterizer & stacking/paint order

Capstone: Course 1's own real capstone document, rendered end to end into a real, pixel-checked image

Exercises: 27 hands-on exercises with worked, verified solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 10

From Style Tree to Layout Tree

Building a Web Browser Engine: Layout & Rendering

Chapter 1 · From Style Tree to Layout Tree

Course 1 ended with a fully-resolved `StyledNode` tree — every DOM node paired with its own final computed style. That tree still doesn't know how big anything is, or where anything sits. This course's own job is to answer that, and it starts here: converting the style tree into a second, related-but-genuinely-different tree — the **layout tree** — shaped specifically for the algorithms the rest of this course builds.

The LayoutBox: Block, Inline, or Anonymous

```
class LayoutBox:
    def __init__(self, box_type, styled_node=None):
        self.box_type = box_type          # 'block' | 'inline' | 'anonymous'
        self.styled_node = styled_node    # None for anonymous boxes -- they came from
        self.children = []                # no single DOM node at all

def display_type(styled_node):
    """A text node is always inline -- it can never itself be a block
    container. An element uses its own computed 'display' value."""
    if styled_node.node.kind == 'text':
        return 'inline'
    return styled_node.style.get('display', 'inline')
```

A Naive First Attempt — and the Rule It Silently Breaks

```
def build_layout_tree_naive(styled_node):
    box = LayoutBox(display_type(styled_node), styled_node)
    for child in styled_node.children:
        box.children.append(build_layout_tree_naive(child))
    return box
```

A plain 1:1 mapping — every style-tree node becomes exactly one `LayoutBox`, in exactly the same shape. Every later chapter in this course, though, is going to need one specific guarantee: **a block box's own direct children never include a raw, unwrapped inline box.** Block layout (Chapter 3) stacks its children vertically — that only works if every direct child genuinely behaves like a block. Real, ordinary HTML routinely violates this on the surface.

VERIFIED DIRECTLY — PLAIN TEXT SITTING NEXT TO A REAL BLOCK ELEMENT BREAKS THE INVARIANT IMMEDIATELY

A `<div>` whose content is `Hello <b>world</b>, welcome!\n<p>Second paragraph.</p>\ntail text`. Its style-tree children are, in order: a text node, `<b>`, another text node, `<p>`, and a final text node. Run through `build_layout_tree_naive`, the div's own `LayoutBox` children come out as `['inline', 'inline', 'inline', 'block', 'inline']` — three separate inline boxes and one block box, sitting as direct siblings of each other. A real block-stacking algorithm handed this tree has no consistent rule to apply: some children are individually-sized blocks, some are inline runs that should flow together on a shared line — and nothing in the tree says which is which without re-deriving it every time.

The Fix: Group Consecutive Inline Runs Into One Shared Anonymous Box

```
def build_layout_tree(styled_node):
    box_type = display_type(styled_node)
    box = LayoutBox(box_type, styled_node)

    if box_type != 'block':
        # an inline box's own children are handled by a later chapter's own
        # line-box logic -- just recurse plainly, nothing to group here
        for child in styled_node.children:
            box.children.append(build_layout_tree(child))
        return box

    pending_inline = []
    def flush():
        if pending_inline:
            anon = LayoutBox('anonymous', None)
            anon.children = pending_inline[:]
            box.children.append(anon)
            pending_inline.clear()

    for child_styled in styled_node.children:
        child_type = display_type(child_styled)
        if child_type == 'block':
            flush()
            box.children.append(build_layout_tree(child_styled))
        else:
            pending_inline.append(build_layout_tree(child_styled))
    flush()
    return box
```

VERIFIED DIRECTLY — THREE LEADING INLINE SIBLINGS MERGE INTO ONE SHARED ANONYMOUS BOX, NOT THREE SEPARATE ONES

Same div. The fixed tree's own children come out as `['anonymous', 'block', 'anonymous']` — the text/`<b>`/text run collapses into a single anonymous box containing all three, `<p>` stays a real block box, and the trailing tail text after `<p>` gets its own separate anonymous box, since a real block sibling came between the two runs. Grouping is per contiguous run, not a single blanket wrapper for the whole container.

VERIFIED DIRECTLY — AN INLINE BOX'S OWN CHILDREN ARE NEVER GROUPED, AND A PURELY-BLOCK CONTAINER GETS ZERO ANONYMOUS BOXES

`<b>`'s own text child stays a plain inline child, untouched — grouping only ever applies to a *block* box's own direct children. And a `<div>` containing only `<h1>` and `<p>`, with no stray inline content at all, produces exactly `['block', 'block']` — no anonymous boxes appear when there's genuinely nothing inline to wrap.

The Real Document, Revisited

Course 1's own capstone document had ordinary line breaks in its source between `<h1>`, `<p>`, and `<ul>` — and that chapter's own closing finding noted these survive as real, if invisible, whitespace-only text nodes in the style tree. Run through *this* chapter's own machinery, that finding turns out to be exactly the problem this chapter exists to solve.

VERIFIED DIRECTLY — THE REAL CAPSTONE DOCUMENT HAS THIS EXACT PROBLEM, NOT JUST THE SYNTHETIC EXAMPLE

`#page`'s own real style-tree children (built by Course 1's actual, unmodified pipeline) come out as `['inline', 'block', 'inline', 'block', 'inline', 'block', 'inline']` — four whitespace-only text nodes sitting directly between three real block elements. The naive layout tree built from this real document fails the same invariant check the synthetic example failed. The fixed `build_layout_tree` resolves it identically: every child becomes `block` or `anonymous`, with each isolated whitespace run wrapped in its own single-child anonymous box.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A block box's own direct children must never mix raw block and inline types	Chapter 3's own block-layout algorithm, which will simply iterate over a block box's own children and stack each one vertically — a guarantee this chapter's own grouping step is what makes that safe to do unconditionally
Anonymous boxes exist purely to satisfy that invariant, with no DOM node of their own	Chapter 5's own inline layout and line-box building — an anonymous box is exactly where a single inline formatting context (one shared line-flowing region) will actually get built
Whitespace-only text nodes from Course 1's own capstone turning out to be a real instance of this problem	Course 1 Chapter 10's own closing bonus finding, explicitly forward-referenced there as "Course 2's own inline-layout whitespace handling" — this chapter is the first half of that promise, grouping the whitespace; a later chapter decides what to actually do with it during real line-breaking

Hands-On Exercises

EXERCISE 1

Build a single `<span>` style-tree node with one text child and no block descendants anywhere. Run it through `build_layout_tree()` and confirm the resulting box's own `box_type` is `'inline'` and its own child is also `'inline'`, un-grouped — then explain why this case never even reaches the grouping logic at all.

View solution

EXERCISE 2

Build a block container whose children are two separate inline runs, split by one real block sibling in between: text + `<b>` (run one), a `<p>`, then a lone trailing text node (run two). Run it through `build_layout_tree()` and confirm you get exactly two separate anonymous boxes — not one combined box spanning the whole container — and that the first anonymous box's own children count is 2 while the second's is 1.

View solution

EXERCISE 3

Run Course 1's own real capstone document (`#page` , containing `<h1>` , `<p class="intro">` , and `<ul>` , with real whitespace between each) through both `build_layout_tree_naive()` and `build_layout_tree()` . Confirm `check_block_container_invariant()` returns `False` for the naive tree and `True` for the fixed one, and count how many separate anonymous boxes the fixed version produces for `#page` 's own direct children.

View solution

CHAPTER 1 QUICK REFERENCE

- **LayoutBox:** `box_type` (`'block'` / `'inline'` / `'anonymous'`), an optional `styled_node` (`None` for anonymous boxes), and a list of child `LayoutBox` es
- **The invariant every later chapter relies on:** a block box's own direct children never include a raw, unwrapped inline box
- **Real bug found and fixed:** a plain 1:1 style-tree-to-layout-tree mapping lets block and inline boxes sit as direct siblings whenever stray text or an inline element appears next to a real block child — grouping consecutive inline runs into shared anonymous boxes fixes it
- **Verified:** grouping is per contiguous run (not a single blanket wrapper), inline boxes' own children are never grouped, and a purely-block container produces zero anonymous boxes
- **Verified against the real document:** Course 1's own capstone whitespace-text finding turns out to be exactly this problem, confirmed on the real pipeline output, not just a synthetic example
- **Next chapter:** The Box Model — content, padding, border, and margin, and computing each box's own real dimensions from its computed style

CHAPTER 2 OF 10

The Box Model: Content, Padding, Border, Margin

Building a Web Browser Engine: Layout & Rendering

Chapter 2 · The Box Model: Content, Padding, Border, Margin

Every real element on a page occupies four nested rectangles, not one: a content box, wrapped by padding, wrapped by a border, wrapped by margin. Every later chapter in this course computes real pixel positions by growing one of these boxes outward into the next. Getting the arithmetic and the CSS shorthand parsing genuinely right here is what everything downstream depends on.

SCOPE NOTE

This course only implements `box-sizing: content-box` — the default, where an element's own `width` / `height` describe the *content* box specifically, and padding/border/margin all add on top of that. `box-sizing: border-box` (where `width` instead describes the outer, padding-and-border-inclusive box) is a real, common CSS feature — deliberately out of this course's own stated scope.

Four Boxes, Grown Outward One at a Time

```
class Rect:
    def __init__(self, x, y, width, height):
        self.x = x; self.y = y; self.width = width; self.height = height

class EdgeSizes:
    def __init__(self, top, right, bottom, left):
        self.top = top; self.right = right; self.bottom = bottom; self.left = left

def expand_rect_by(rect, edge):
    """Grows a rect OUTWARD by an EdgeSizes -- the one mechanism behind
    every step from content -> padding -> border -> margin box."""
    return Rect(
        rect.x - edge.left,
        rect.y - edge.top,
        rect.width + edge.left + edge.right,
        rect.height + edge.top + edge.bottom,
    )

class Dimensions:
    def __init__(self, content, padding, border, margin):
        self.content = content    # a Rect -- the CONTENT box only
        self.padding = padding    # EdgeSizes
        self.border = border      # EdgeSizes
        self.margin = margin      # EdgeSizes

    def padding_box(self):
        return expand_rect_by(self.content, self.padding)
    def border_box(self):
        return expand_rect_by(self.padding_box(), self.border)
    def margin_box(self):
        return expand_rect_by(self.border_box(), self.margin)
```

Every one of the three outer boxes is defined purely in terms of the box directly inside it — the same one `expand_rect_by` function, called three times in sequence, is the entire box model. This is exactly what a real browser's own DevTools "box model" panel visualizes: content, padding, border, and margin as four nested rectangles, each one measurably bigger than the last.

Bug 1: A Length Parser That Crashes on CSS's Own Valid Zero

```
def parse_length_naive(s):
    return float(s[:-2])   # assumes every length always ends in 'px'
```

VERIFIED DIRECTLY — AND IT'S NOT HYPOTHETICAL, IT'S ALREADY IN THIS COURSE'S OWN PUBLISHED UA STYLESHEET

`parse_length_naive('16px')` works fine — `16.0`. But `parse_length_naive('0')` raises `ValueError: could not convert string to float: ''` — `'0'[:-2]` on a one-character string is the empty string, and `float('')` fails. CSS explicitly allows a bare, unitless `0` (every other length needs a unit, but zero never does) — and Chapter 9's own published `UA_STYLESHEET` already contains exactly this shape: `Rule(['p'], [Declaration('margin', '16px 0')])`. Parsing that real, already-existing rule's own value with the naive parser crashes immediately.

```
def parse_length(s):
    s = s.strip()
    if s == '0':
        return 0.0
    if s.endswith('px'):
        return float(s[:-2])
    raise ValueError(f"unsupported length: {s!r}")
```

VERIFIED DIRECTLY — THE FIX RESOLVES THE EXACT REAL VALUE THAT BROKE

`parse_length('16px 0'.split())` — parsing each token of Chapter 9's own real margin value separately — gives `[16.0, 0.0]`, correctly.

Bug 2: Shorthand Expansion Getting the 3-Value Form Wrong

CSS's own `margin` / `padding` / `border-width` shorthand accepts 1, 2, 3, or 4 space-separated values, each shape meaning something genuinely different.

```
def expand_shorthand(value_str):
    lengths = [parse_length(p) for p in value_str.split()]
    if len(lengths) == 1:
        t = r = b = l = lengths[0]
    elif len(lengths) == 2:
        t = b = lengths[0]; r = l = lengths[1]
    elif len(lengths) == 3:
        t, r, b = lengths
        l = r                       # left REUSES the horizontal value
    elif len(lengths) == 4:
        t, r, b, l = lengths        # clockwise from the top
    return EdgeSizes(t, r, b, l)
```

VERIFIED DIRECTLY — A PLAUSIBLE NAIVE VERSION REUSES THE WRONG EARLIER VALUE FOR THE 3-VALUE CASE

A naive first attempt at the 3-value branch — `t, r, b = lengths; l = t` — reaches for the top value as `left`'s own fallback, since it's the first one already in scope. `expand_shorthand_naive("10px 20px 30px")` gives `EdgeSizes(top=10, right=20, bottom=30, left=10)`. Real CSS's 3-value form means *(top, horizontal, bottom)* — `left` is supposed to reuse the **horizontal** (second) value, `20`, matching `right`, not the top value. The fixed version's `l = r` gets this right: `EdgeSizes(top=10, right=20, bottom=30, left=20)`.

VERIFIED DIRECTLY — ALL FOUR SHORTHAND FORMS BEHAVE CORRECTLY

`expand_shorthand("5px")` → all four sides equal, `5` each. `expand_shorthand("10px 20px")` → vertical `10`, horizontal `20`. `expand_shorthand("1px 2px 3px 4px")` → top/right/bottom/left exactly as written, clockwise from the top.

A Real Dimensions Computation, Box by Box

A content box `100×50` at the origin, `padding: 10px`, `border-width: 2px`, `margin: 16px 0` — the exact margin shape from Chapter 9's own real UA rule.

VERIFIED DIRECTLY — EVERY BOX GROWS EXACTLY AS MUCH AS THE ARITHMETIC PREDICTS

Content: `Rect(0, 0, 100, 50)`. **Padding box:** `10px` on all sides → `Rect(-10, -10, 120, 70)` (width/height each grow by 20 — 10 on both sides). **Border box:** a further `2px` all sides → `Rect(-12, -12, 124, 74)`. **Margin box:** `16px 0` — a 2-value shorthand, vertical `16`, horizontal `0` — adds `16` to top and bottom only → `Rect(-12, -28, 124, 106)`, width unchanged from the border box, height up by `32`.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
One `expand_rect_by` function driving all three outer boxes	Chapter 3's own block layout, which will use `margin_box()` / `border_box()` directly to decide how much vertical space a block actually occupies and where the next sibling starts
A genuinely real bug (bare unitless zero) traced directly to already-published Chapter 9 code	Chapter 10's own capstone integration bug, from Course 1 — another case where a value produced correctly by an earlier chapter exposed a genuine gap only once a later chapter actually tried to consume it
CSS shorthand's own real, specific value-count rules	A recurring theme across this whole site's CSS-adjacent material: shorthand properties look simple but encode real, easy-to-misremember rules — getting the 3-value case backwards is a mistake real CSS authors make too, not just an engine bug

Hands-On Exercises

EXERCISE 1

Call `expand_shorthand()` on `"4px 8px 12px 16px"` (a full 4-value form), `"3px 6px"` (2-value), and `"20px"` (1-value). Confirm each produces the exact `EdgeSizes` real CSS's own shorthand rules predict, and state which one of the three forms is the only one where all four sides can end up genuinely different from each other.

View solution

EXERCISE 2

Build a `Dimensions` for a content box `200×100` at the origin, with `padding: '4px 8px 12px 16px'`, `border-width: '3px 6px'`, and `margin: '20px'`. Compute `padding_box()` and `border_box()` and confirm both against hand-worked arithmetic, showing your work for each edge separately rather than only checking the final numbers.

View solution

EXERCISE 3

Reproduce this chapter's own two bugs directly: call `parse_length_naive('0')` and confirm it raises `ValueError`, then call `expand_shorthand_naive("10px 20px 30px")` and confirm its own `left` value is wrong. For each bug, identify the exact line responsible and explain, in your own words, why the mistake is easy to make even though it's wrong.

View solution

CHAPTER 2 QUICK REFERENCE

- **Four boxes:** content → padding → border → margin, each one the box before it grown outward by a real `EdgeSizes`
- **Scope:** `box-sizing: content-box` only — `width` / `height` describe the content box; `border-box` is out of scope
- **Real bug found and fixed:** a naive length parser assuming every value ends in `'px'` crashes on CSS's own valid bare zero — already present in Chapter 9's own published UA stylesheet (`margin: 16px 0`)
- **Real bug found and fixed:** naive 3-value shorthand expansion reuses the *top* value for `left` instead of the correct *horizontal* (second) value
- **Verified:** all four shorthand forms (1/2/3/4 values) match real CSS's own defined expansion rules exactly
- **Next chapter:** Block Layout — using these box dimensions to actually stack a block box's own children vertically and compute real widths and heights

CHAPTER 3 OF 10

Block Layout: Width, Height & Position

Building a Web Browser Engine: Layout & Rendering

Chapter 3 · Block Layout: Width, Height & Position

Chapter 2 built the machinery for one box's own dimensions. This chapter puts real numbers into it — a real, recursive algorithm that computes an actual width, an actual x/y position, and an actual height for every block box in a tree, in the right order, each one depending on the boxes around it exactly the way real CSS specifies.

The Four-Step Algorithm

```
def layout_block(layout_box, containing_block):
    calculate_block_width(layout_box, containing_block)
    calculate_block_position(layout_box, containing_block)
    layout_block_children(layout_box)
    calculate_block_height(layout_box)
```

Width has to be known before position (position depends on this box's own margin/border/padding, computed alongside width). Position has to be known before children are laid out (children need this box's own content-box coordinates as their *own* containing block). And height, for a box with no explicit `height`, can only be known *after* children — it's the sum of what they turned out to need.

Width: A Real Bug in the "Auto" Case

```
def calculate_block_width_naive(layout_box, containing_block):
    style = box_style(layout_box)
    padding = expand_shorthand(style.get('padding', '0'))
    border = expand_shorthand(style.get('border-width', '0'))
    margin = expand_shorthand(style.get('margin', '0'))

    width_str = style.get('width', 'auto')
    if width_str == 'auto':
        content_width = containing_block.content.width   # BUG
    else:
        content_width = parse_length(width_str)

    layout_box.dimensions = Dimensions(Rect(0, 0, content_width, 0), padding, border, margin)
```

Reasonable-looking: when nothing sets an explicit width, just fill the container. But "fill the container" has to mean something specific — *which* box fills it?

VERIFIED DIRECTLY — A PADDED BOX OVERFLOWS ITS OWN CONTAINER BY EXACTLY ITS OWN PADDING

A 300px-wide containing block. A box with `padding: 10px` and no explicit `width`. The naive function sets its own *content* width to `300` — the same as the container. But that box's real outer edge is its **border box**, which adds the padding back on top: `300 + 10 + 10 = 320`. The box's own visible edge ends up 20px wider than the container it's supposedly filling — a genuine, real overflow, verified directly by computing `border_box()` and comparing its width against the container's own.

```
def calculate_block_width(layout_box, containing_block):
    style = box_style(layout_box)
    padding = expand_shorthand(style.get('padding', '0'))
    border = expand_shorthand(style.get('border-width', '0'))
    margin = expand_shorthand(style.get('margin', '0'))

    width_str = style.get('width', 'auto')
    if width_str == 'auto':
        content_width = (containing_block.content.width
                         - margin.left - margin.right
                         - border.left - border.right
                         - padding.left - padding.right)
    else:
        content_width = parse_length(width_str)

    layout_box.dimensions = Dimensions(Rect(0, 0, content_width, 0), padding, border, margin)
```

VERIFIED DIRECTLY — THE FIX MAKES THE OUTER EDGE FILL THE CONTAINER, NOT THE CONTENT BOX

Same box, same 300px container: content width comes out to `280` (`300 - 10 - 10`), and `border_box().width` is exactly `300` — flush with the container, no overflow. It's the box's real, visible edge that fills the available space; the content box shrinks to make room for whatever padding, border, and margin the box itself carries.

Position and Height: One Box's Own Content Area Becomes Its Children's Container

```
def calculate_block_position(layout_box, containing_block):
    d = layout_box.dimensions
    cb = containing_block.content
    d.content.x = cb.x + d.margin.left + d.border.left + d.padding.left
    d.content.y = cb.y + cb.height + d.margin.top + d.border.top + d.padding.top

def layout_block_children(layout_box):
    d = layout_box.dimensions
    for child in layout_box.children:
        layout_block(child, d)               # THIS box's own dims -> child's containing block
        d.content.height += child.dimensions.margin_box().height

def calculate_block_height(layout_box):
    style = box_style(layout_box)
    height_str = style.get('height', 'auto')
    if height_str != 'auto':
        layout_box.dimensions.content.height = parse_length(height_str)
    # else: leave it as whatever layout_block_children already accumulated
```

`cb.y + cb.height` is the whole stacking mechanism in one expression: a box's own `y` starts at its container's own top, *plus* however much vertical space the container has already accumulated from earlier siblings. `layout_block_children` grows that accumulated height by exactly one child's own `margin_box().height` every time a child finishes — so the next sibling's own position automatically lands below it.

VERIFIED DIRECTLY — A REAL, THREE-LEVEL NESTED LAYOUT, EVERY NUMBER CHECKED BY HAND

A 300px containing block holds box `A` (`padding: 10px`), which holds `B` (`width: 100px; margin: 5px;`) and `C` (`height: 20px`). Final results: `A.content = Rect(10, 10, 280, 30)`, `B.content = Rect(15, 15, 100, 0)`, `C.content = Rect(10, 20, 280, 20)`, and `A`'s own `margin_box() = Rect(0, 0, 300, 50)` — exactly as wide as the original container. `B`'s content height comes out to `0` honestly — it has no children and no explicit height, so with no inline-content sizing built yet (a later chapter's own job), an empty block genuinely collapses to zero height, matching what a real browser does with a truly empty `<div></div>` too.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A box's own dimensions become its children's containing block	Chapter 1's own layout tree — the recursive shape this chapter's algorithm walks is exactly the block/anonymous tree Chapter 1 built, with anonymous boxes handled identically to real block boxes via `box_style()`'s own `INITIAL_VALUES` fallback
"Auto" width filling the container's OUTER edge, not its content edge	Chapter 4's own constraint-based width — this chapter only handles the single common case (width unset, margins not auto); the full puzzle of `auto` margins and over-constrained widths is that chapter's own entire subject
An empty block honestly collapsing to zero height	Chapter 5's own inline layout — real height for boxes containing actual text will come from measuring that text, not from this chapter's own block-only stacking

Hands-On Exercises

EXERCISE 1

Build a container with three plain block children, each with only an explicit `height` set (15px, 25px, 5px) and everything else default. Lay it out inside a 300px-wide root containing block and confirm each child's own `content.y` — showing that the third child's own position depends on the running *sum* of both earlier children's heights, not just the immediately preceding one.

View solution

EXERCISE 2

Build a container with a single child that has an explicit `width: 50px`. Lay it out inside a 300px containing block and confirm the child's own content width is exactly `50`, completely independent of the container's own width — then explain in your own words why this is fundamentally different from the `auto` case, where the container's width genuinely does matter.

View solution

EXERCISE 3

Reproduce this chapter's own overflow bug directly: lay out a box with `padding: 10px` and no explicit width inside a 300px containing block, using both `calculate_block_width_naive()` and `calculate_block_width()`. Compute `border_box().width` for both results and confirm the naive version overflows its own container while the fixed version doesn't — then identify exactly which line differs between the two functions.

View solution

CHAPTER 3 QUICK REFERENCE

- **Order matters:** width → position → children (recursively) → height — each step depends on the one(s) before it
- **Real bug found and fixed:** naive "auto" width sets content width equal to the container's own width directly, ignoring this box's own padding/border/margin — the box's real outer edge (border box) then overflows the container by exactly that padding/border/margin amount
- **The fix:** for "auto" width, subtract this box's own horizontal margin/border/padding from the container's width first, so the box's own *outer* edge — not its content edge — fills the available space
- **Position:** `cb.y + cb.height` is the whole vertical-stacking mechanism — a box starts where its container's own already-accumulated height leaves off
- **Height:** for an explicit `height`, use it directly; otherwise, sum every child's own `margin_box().height` — verified against a real three-level nested layout, checked field by field
- **Next chapter:** Constraint-Based Width — the full "auto" puzzle when margins themselves can also be `auto`

CHAPTER 4 OF 10

Constraint-Based Width: The "Auto" Value Puzzle

Building a Web Browser Engine: Layout & Rendering

Chapter 4 · Constraint-Based Width: The "Auto" Value Puzzle

Chapter 3 handled exactly one case: `width: auto`, with margins that were never anything but a plain number. Real CSS allows *three* independent things to be `auto` at once — `width`, `margin-left`, and `margin-right` — and which one (or two) actually end up `auto` changes which equation the browser has to solve. This chapter builds the real algorithm.

SCOPE NOTE

This chapter implements the algorithm for left-to-right (LTR) content only, matching this course's own stated scope — right-to-left layout swaps which margin absorbs the over-constrained case, and isn't covered here.

A Bug Inherited Directly From Chapter 3's Own Approach

Chapter 3 computed margin the same way it computed padding and border-width — `expand_shorthand()`, straight from Chapter 2. That function calls `parse_length()` on every token in a shorthand value, unconditionally.

VERIFIED DIRECTLY — THE SINGLE MOST FAMOUS CSS IDIOM IN EXISTENCE CRASHES THIS ENGINE OUTRIGHT

`margin: 0 auto` — the classic "center this block horizontally" trick, quite possibly the most-copy-pasted line of CSS ever written — fed through Chapter 3's own margin handling raises `ValueError: unsupported length: 'auto'`. `expand_shorthand()` was built for `padding` and `border-width`, two properties that genuinely never allow `auto` at all — `parse_length('auto')` has no case for that string, and never needed one, until margin (which *does* allow `auto`) started reusing the same function.

```
class MarginEdges:
    """Like EdgeSizes, but each field may hold the STRING 'auto' instead
    of a number -- margin is the only one of the three that allows it."""
    def __init__(self, top, right, bottom, left):
        self.top = top; self.right = right; self.bottom = bottom; self.left = left

def parse_margin_component(s):
    s = s.strip()
    if s == 'auto':
        return 'auto'
    return parse_length(s)

def expand_margin_shorthand(value_str):
    parts = [parse_margin_component(p) for p in value_str.split()]
    # same 1/2/3/4-value expansion as Chapter 2's own expand_shorthand,
    # just using parse_margin_component instead of parse_length
    ...
    return MarginEdges(t, r, b, l)
```

VERIFIED DIRECTLY — 'AUTO' IS NOW A REAL, DISTINCT VALUE, NOT A CRASH

`expand_margin_shorthand('0 auto')` returns `top=0.0, bottom=0.0, left='auto', right='auto'` — the 2-value shorthand rule (vertical, horizontal) applying exactly as before, just with the horizontal component preserved as the literal string `'auto'` rather than an attempted (and failing) number parse.

The Real Algorithm: Three Cases

`margin-left + border-left + padding-left + width + padding-right + border-right + margin-right` has to equal the containing block's own width. Solve for whichever piece(s) are `auto`.

VERIFIED DIRECTLY — WIDTH:AUTO STILL MATCHES CHAPTER 3'S OWN ALREADY-VERIFIED RESULT EXACTLY

A 300px container, `padding: 10px`, no margin set (defaults to a plain, non-auto `0`). Content width: `280.0` — identical to Chapter 3's own result. This chapter's algorithm is a strict generalization, not a replacement.

VERIFIED DIRECTLY — A REAL, GENUINELY SURPRISING CSS RULE: AN EXPLICIT MARGIN-RIGHT CAN BE SILENTLY OVERRIDDEN

`width: 100px; margin: 20px;` (both sides, explicitly 20px each) in that same 300px container. Final `margin.left`: `20.0`, exactly as declared. Final `margin.right`: `180.0` — *not* the declared 20px. This isn't a bug in this engine; it's a real, specified rule: when `width` and both margins are all non-`auto`, the total is **over-constrained**, and (in LTR) the browser silently recomputes `margin-right` to make everything fit exactly, discarding whatever value was actually written for it. Real CSS authors get caught out by this regularly.

VERIFIED DIRECTLY — MARGIN:0 AUTO GENUINELY CENTERS THE BOX, NUMERICALLY

Same container, `width: 100px; margin: 0 auto;`. Final `margin.left`: `100.0`. Final `margin.right`: `100.0`. `100 + 100 + 100 = 300` — the box sits exactly centered, because both margins being `auto` means the leftover space (`containing block width - everything else`) is split **evenly** between them.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A dedicated `MarginEdges` / `expand_margin_shorthand`, distinct from Chapter 2's own `EdgeSizes` / `expand_shorthand`	Chapter 2's own established pattern of one shared mechanism (`expand_rect_by`) driving all three outer boxes — this chapter shows that pattern has a real limit: margin's own `auto` behavior is genuinely different from padding/border-width, and reusing identical code for all three was a mistake worth catching explicitly
The over-constrained case silently discarding a declared margin-right value	A real, well-known CSS gotcha independent of this course entirely — worth knowing when debugging a real page where an author's own margin value seems to be "ignored" by the browser for no apparent reason
Both-auto margins splitting the leftover space evenly	Chapter 3's own `calculate_block_position`, unchanged — once `dimensions.margin.left` holds a real, resolved number (never the string `'auto'`), position calculation works exactly as already built, with no changes needed there at all

Hands-On Exercises

EXERCISE 1

Build a box with `width: 100px; margin: 0 20px 0 auto;` — a 4-value shorthand where only `left` is `auto` — inside a 300px containing block. Compute `margin.left` and `margin.right` and confirm the explicitly-set `margin-right` (20px) is left completely untouched while `margin-left` alone absorbs the entire remaining space.

View solution

EXERCISE 2

Build a box with `width: 250px; margin: 10px auto;` inside a 400px containing block. Confirm the box centers correctly (both horizontal margins equal, and their sum plus the width equals the container's own width), and confirm the shorthand's own vertical component (10px) is completely unaffected by the auto-margin solving.

View solution

EXERCISE 3

Reproduce this chapter's own crash directly: call `calculate_block_width_naive()` on a box with `margin: 0 auto` and confirm it raises `ValueError`. Identify the exact function call inside `calculate_block_width_naive` responsible, and explain specifically why `padding` and `border-width` never trigger this same crash even though they go through structurally identical shorthand-parsing code.

View solution

CHAPTER 4 QUICK REFERENCE

- **Real bug found and fixed:** reusing Chapter 2's own general-purpose shorthand expander for margin crashes on `auto` — margin needs its own `MarginEdges` / `expand_margin_shorthand`, since it's the only one of the three (margin/border/padding) that CSS allows `auto` on at all
- **width:auto** (margins fixed) — width absorbs all remaining space, exactly matching Chapter 3's own already-verified behavior
- **Neither margin auto, explicit width** — over-constrained (or an exact fit): margin-right is silently recomputed to fill the leftover space, its own declared value discarded (a real, specified CSS rule, not a bug)
- **One margin auto** — that margin alone absorbs all remaining space
- **Both margins auto** — the classic centering idiom: leftover space is split evenly between left and right, verified numerically centered
- **Next chapter:** Inline Layout & Line Boxes — giving real width and height to the text and inline content this course has deferred since Chapter 1

CHAPTER 5 OF 10

Inline Layout & Line Boxes

Building a Web Browser Engine: Layout & Rendering

Chapter 5 · Inline Layout & Line Boxes

Every anonymous box built back in Chapter 1 has sat empty ever since — a real, correctly-shaped wrapper with genuine inline content inside it, but nothing that ever gave that content an actual size. This chapter is where text and inline elements finally get laid out: flattened into a flat sequence, then broken into real **line boxes** that fit an available width.

SCOPE NOTE

Real character widths depend on the actual font, its size, and even which specific letters are involved — genuine font-metric measurement is Chapter 6's own entire subject. This chapter uses an honestly-flagged placeholder: every character (and every inter-word space) counts as a fixed `8px`, regardless of what it actually is. The line-*breaking* algorithm this chapter builds is real and correct; only the width numbers it's fed are a stand-in.

Flattening Nested Inline Content Into a Flat Word List

```
def collect_words(layout_box):
    """Flattens an inline/anonymous box's own text content into a flat
    list of words, walking through nested inline elements (like <b>)
    in order."""
    words = []
    if (layout_box.box_type == 'inline' and layout_box.styled_node is not None
            and layout_box.styled_node.node.kind == 'text'):
        words.extend(layout_box.styled_node.node.text.split())
    for child in layout_box.children:
        words.extend(collect_words(child))
    return words
```

VERIFIED DIRECTLY — NESTED INLINE ELEMENTS CONTRIBUTE THEIR OWN TEXT, IN READING ORDER

An anonymous box built from Chapter 1's own inline-run shape — `"This is " + <b>"bold"</b> + " and normal text."` — flattens via `collect_words()` to exactly `['This', 'is', 'bold', 'and', 'normal', 'text.']`. A text box contributes its own words directly; an inline *element* box like `<b>` contributes nothing of its own, but the recursion still walks into *its* children, correctly picking up the text nested one level deeper.

Line-Breaking: A Real Bug in the "Does It Fit" Check

```
def break_into_lines_naive(words, available_width):
    lines = []
    current_line = []
    current_width = 0
    for word in words:
        w = measure_word(word)
        if current_line and current_width + w > available_width:
            lines.append(current_line)
            current_line = [word]
            current_width = w
        else:
            current_line.append(word)
            current_width += w   # BUG: never accounts for the space before this word
    if current_line:
        lines.append(current_line)
    return lines
```

VERIFIED DIRECTLY — THREE WORDS THAT "FIT" BY THE NAIVE CHECK OVERFLOW BY 12PX ONCE ACTUALLY RENDERED

The real sentence `"The quick brown fox jumps over the lazy dog"`, broken at a 100px available width. The naive algorithm places `"jumps"`, `"over"`, and `"the"` on one line — their word widths alone sum to `96`, which passes the naive `96 <= 100` check. But the **real** rendered width of that line — three words plus the two real spaces that have to sit between them — is `96 + 16 = 112`. A genuine 12px overflow the naive check never even measured, because it only ever tracked word widths, never the spaces the words are actually going to need once they sit next to each other on a real line.

```
def break_into_lines(words, available_width):
    lines = []
    current_line = []
    current_width = 0
    for word in words:
        w = measure_word(word)
        extra = w if not current_line else (w + SPACE_WIDTH_PX)
        if current_line and current_width + extra > available_width:
            lines.append(current_line)
            current_line = [word]
            current_width = w
        else:
            current_line.append(word)
            current_width += extra
    if current_line:
        lines.append(current_line)
    return lines
```

VERIFIED DIRECTLY — EVERY REAL LINE'S OWN RENDERED WIDTH NOW GENUINELY FITS

The exact same sentence, fixed algorithm, same 100px width, breaks into `[['The','quick'], ['brown','fox'], ['jumps','over'], ['the','lazy','dog']]` — four real lines, each one's true rendered width (words plus every real inter-word space) individually checked and confirmed at or under 100px: `72`, `72`, `80`, and `96`.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
`collect_words()` walking through nested inline elements	Chapter 1's own anonymous-box grouping — this chapter is the first to actually consume the content those anonymous boxes were built to hold, confirming the grouping mechanism produces something genuinely usable
A real, honest placeholder character-width model	Chapter 6's own real font-metric measurement — the exact same `break_into_lines()` algorithm this chapter built stays unchanged; only `measure_word()`'s own implementation gets replaced with something accurate
Space-width accounted for BEFORE the overflow check, not after	Chapter 3's own real "auto" width overflow bug — the same shape of mistake (checking against a quantity that was measured too narrowly) showing up a second time, in a genuinely different part of this course's own layout engine

Hands-On Exercises

EXERCISE 1

Run this chapter's own real nine-word sentence through `break_into_lines()` with a generous 400px available width. Confirm it all fits on a single line, and confirm that single line's own real rendered width (via the chapter's own width-plus-spaces formula) is still verified within the 400px budget rather than just assumed.

View solution

EXERCISE 2

Call `break_into_lines(['The'], 10)` — a single word, in an available width narrower than the word itself. Confirm the word still ends up on its own line rather than being dropped or raising an error, and explain exactly which part of the algorithm guarantees a lone word can never be rejected outright, no matter how narrow the available width is.

View solution

EXERCISE 3

Reproduce this chapter's own overflow bug directly: run the real nine-word sentence through both `break_into_lines_naive()` and `break_into_lines()` at the same 100px width. For every line in the naive result, compute its real rendered width and confirm exactly one line overflows. Identify the specific variable in `break_into_lines_naive` that never gets incremented by `SPACE_WIDTH_PX`, and where the fixed version adds it instead.

View solution

CHAPTER 5 QUICK REFERENCE

- **collect_words():** flattens an anonymous box's own inline content — plain text and nested inline elements alike — into one flat, ordered word list
- **Placeholder measurement:** a fixed 8px per character and per space, honestly flagged for replacement by Chapter 6's own real font metrics
- **Real bug found and fixed:** naive line-breaking accumulates only word widths, never the space that has to sit between them — verified overflowing a real 100px line by 12px on a real nine-word sentence
- **The fix:** account for one `SPACE_WIDTH_PX` before every word except the first on a line, as part of the overflow check itself, not as an afterthought
- **Verified:** a lone word too wide for the available space still gets placed on its own line — never dropped, never crashes, matching how real browsers render an unbreakable overflowing word
- **Next chapter:** Text Measurement & Font Metrics — replacing this chapter's own placeholder width model with something a real browser would actually trust

CHAPTER 6 OF 10

Text Measurement & Font Metrics

Building a Web Browser Engine: Layout & Rendering

Chapter 6 · Text Measurement & Font Metrics

Chapter 5 built a real line-breaking algorithm and honestly flagged its own weak point: every character, at every font-size, measured as a flat `8px`. This chapter replaces that placeholder with something a real browser would actually trust — and finds that the placeholder wasn't just imprecise, it was wrong in a way that would visibly break real pages.

A Real (Simplified) Per-Glyph Width Table, in Em Units

```
CHAR_WIDTHS_EM = {
    # narrow characters
    'i': 0.28, 'l': 0.28, 'j': 0.28, '.': 0.28, ',': 0.28,
    # normal-width lowercase
    'a': 0.50, 'e': 0.44, 'o': 0.50, ...
    # wide lowercase
    'm': 0.78, 'w': 0.72,
    # uppercase -- generally wider than their own lowercase counterpart
    'A': 0.67, 'W': 0.94, ...
}
DEFAULT_CHAR_WIDTH_EM = 0.50   # fallback for anything not in the table

def char_width_px(ch, font_size_px):
    em = CHAR_WIDTHS_EM.get(ch, DEFAULT_CHAR_WIDTH_EM)
    return em * font_size_px

def measure_word(word, font_size_px):
    return sum(char_width_px(ch, font_size_px) for ch in word)
```

Defining widths as a fraction of the font's own size — "em" units, matching how real font metrics actually work — is exactly what makes correct font-size scaling fall out for free: multiply every character's own em-width by whatever `font_size_px` actually is, and the whole word scales proportionally.

Bug 1: Chapter 5's Placeholder Had No Way to Know Font-Size Ever Mattered

VERIFIED DIRECTLY — AND THIS IS A REAL, ALREADY-PUBLISHED FACT ABOUT THIS COURSE'S OWN UA STYLESHEET

`measure_word_naive('Welcome')` — Chapter 5's own flat model — has no `font_size_px` parameter *at all*. It returns `56`, unconditionally, whether that word is rendered as tiny fine print or a giant headline. The real model gives `'Welcome'` a width of `61.12px` at the default 16px body-text size, and **exactly double**, `122.24px`, at 32px. That doubling isn't an arbitrary test case — Course 1's own published `UA_STYLESHEET` already contains `Rule(['h1'], [Declaration('font-size', '2em'), ...])`. A real `<h1>`'s own text is measurably almost twice as wide, per word, as the identical text set as plain body copy — and the old placeholder had no way to represent that at all.

Bug 2: Even at One Fixed Font-Size, a Flat Model Ignores Real Glyph Shape

VERIFIED DIRECTLY — TWO EQUAL-LENGTH STRINGS, DRAMATICALLY DIFFERENT REAL WIDTHS

`measure_word_naive('iiiii')` and `measure_word_naive('mmmmm')` both return `40` — five characters, flat 8px each, no distinction at all. The real model, at the same 16px: `'iiiii'` comes out to `22.4px`, `'mmmmm'` to `62.4px` — nearly three times wider, matching how dramatically different these two letters actually look in any real font.

The Real, Motivated Overflow: An H1's Own Text

A real `<h1>` containing `"Welcome home now"`, available width `150px`, actual font-size `32px` (per the real UA rule above).

VERIFIED DIRECTLY — THE OLD MODEL PACKS ALL THREE WORDS ONTO ONE LINE; THE REAL WIDTH OVERFLOWS BY 116PX

Fed Chapter 5's own line-breaking algorithm, using the OLD flat measurement: all three words end up on **one single line**, because their flat 8-per-character counts sum to `128`, comfortably under the 150px budget. But measured correctly, at the real 32px this text is actually rendered at, that one line's true rendered width is `266.24px` — a genuine `116px` overflow the old model had no way to see, because it never had font-size as an input in the first place.

VERIFIED DIRECTLY — THE REAL MODEL WRAPS CORRECTLY, AND BOTH RESULTING LINES GENUINELY FIT

Line-broken again with the real, font-size-aware model: `[['Welcome'], ['home', 'now']]`. `'Welcome'` alone already takes `122.24px` at this size, so it correctly gets its own line before the 150px budget is exceeded. Both resulting lines' own real rendered widths — `122.24px` and `135.04px` — are independently confirmed within the 150px budget.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Widths defined in em units, scaling with font-size	Course 1 Chapter 7's own inheritance model — `font-size` is one of the properties Chapter 7 established as inheritable, meaning a real font-size genuinely does propagate down through nested inline elements like `<b>`, exactly the value this chapter's own `measure_word` needs
Chapter 5's own line-breaking loop reused completely unchanged	Chapter 5's own explicit forward reference — "the exact same `break_into_lines()` algorithm this chapter built stays unchanged; only `measure_word()`'s own implementation gets replaced" — confirmed exactly true here, only the measurement function itself changed
A real 116px overflow on genuinely real UA-stylesheet content	Chapter 2's own overflow bug and Chapter 3's own overflow bug — a recurring theme across this entire layout course: a plausible simplification quietly produces boxes or lines that visually overflow their own intended space, caught only once real numbers are actually run through it

Hands-On Exercises

EXERCISE 1

Call `char_width_px('#', 16.0)` — a character with no explicit entry in `CHAR_WIDTHS_EM`. Confirm it falls back to `DEFAULT_CHAR_WIDTH_EM` rather than raising an error, and compute the exact pixel value it should return at 16px.

View solution

EXERCISE 2

Compare `char_width_px('a', 16.0)` against `char_width_px('A', 16.0)`. Confirm the uppercase letter measures wider than its own lowercase counterpart at the identical font-size, and explain why this matches real typography rather than being an arbitrary choice in the width table.

View solution

EXERCISE 3

Compute `measure_word('Welcome', 24.0)` and compare it against `measure_word('Welcome', 16.0)`. Confirm the relationship between the two is exactly what the em-based scaling model predicts, using a font-size that isn't a clean doubling of the 16px reference, and explain why this confirms the scaling is genuinely linear rather than a special case that only happens to work for 2x.

View solution

CHAPTER 6 QUICK REFERENCE

- **Real per-glyph widths, in em units:** each character's own width is a fraction of the font's own size — multiplying by `font_size_px` gives correct scaling for free

- **Real bug found and fixed:** Chapter 5's own flat model had no font-size input at all — verified measuring a real `<h1>`'s own 32px text identically to 16px body text, causing a genuine 116px line overflow once run through real line-breaking
- **A second, independent bug:** even at a single fixed font-size, a flat per-character model can't distinguish a narrow glyph (`'i'`) from a wide one (`'m'`) — fixed by the real per-character table
- **Verified:** the scaling relationship is genuinely linear — doubling font-size exactly doubles measured width, and this holds at any font-size, not just a clean 2x
- **Chapter 5's own line-breaking algorithm needed zero changes** — only the measurement function feeding it was replaced
- **Next chapter:** Painting — turning a finished layout tree into an ordered list of paint commands, the real bridge to actual pixels

CHAPTER 7 OF 10

Painting: From Layout Boxes to a Display List

Building a Web Browser Engine: Layout & Rendering

Chapter 7 · Painting: From Layout Boxes to a Display List

Every chapter so far has computed real numbers — widths, positions, line breaks — but none of it has said anything about what actually gets drawn. This chapter builds the bridge: walking a finished layout tree and producing a real, flat, ordered **display list** of simple paint commands — the exact intermediate representation real browsers build before ever touching a pixel.

A Paint Command, and the Simplest Real Thing to Paint

```
class PaintCommand:
    def __init__(self, kind, **kwargs):
        self.kind = kind
        self.__dict__.update(kwargs)

def paint_background(layout_box, display_list):
    style = box_style(layout_box)
    bg = style.get('background-color', 'transparent')
    if bg != 'transparent' and layout_box.dimensions is not None:
        rect = layout_box.dimensions.border_box()   # background extends to the BORDER edge
        display_list.append(PaintCommand('rect', rect=rect, color=bg))
```

A background genuinely paints out to the *border* box, not the content box — real CSS behavior, and exactly why `border_box()` (Chapter 2) is what gets used here, not `content` directly.

A Naive Walk Order — and the Real Visual Bug It Causes

```
def build_display_list_naive(layout_box, display_list=None):
    if display_list is None:
        display_list = []
    for child in layout_box.children:
        build_display_list_naive(child, display_list)   # children FIRST
    paint_background(layout_box, display_list)           # then this box's own background
    return display_list
```

Looks harmless — walk the tree, collect commands. But a display list is painted *in order*: each command draws on top of everything already painted before it, exactly the way a real rasterizer works.

VERIFIED DIRECTLY — A CHILD'S BACKGROUND GETS COMPLETELY PAINTED OVER BY ITS OWN PARENT

A real, laid-out nested box (via the actual Chapter 3/4 `layout_block` pipeline, not hand-faked rectangles): box `A` (`background: red`, filling a 300×50 area) containing box `B` (`background: blue`, 100×50, sitting fully inside `A`). The naive walk produces the display list `['blue', 'red']` — `B`'s command comes *first*, `A`'s comes *after*. Simulating what's actually visible at a point sitting inside both boxes, `(50, 25)`: the answer comes out **red**. `A`'s own background, painted last, completely covers `B`'s — a real browser would show `B`'s blue box sitting visibly on top of `A`'s red one, not swallowed by it.

```
def build_display_list(layout_box, display_list=None):
    if display_list is None:
        display_list = []
    paint_background(layout_box, display_list)           # THIS box's own background first
    for child in layout_box.children:
        build_display_list(child, display_list)           # then recurse into children
    return display_list
```

VERIFIED DIRECTLY — SWAPPING THE ORDER FIXES IT, AND A SANITY CHECK CONFIRMS THE FIX IS SCOPED CORRECTLY

Same two boxes, fixed order: display list `['red', 'blue']`. The same overlap point now correctly resolves to **blue** — `B` visibly sits on top of `A`, exactly matching real CSS stacking (a child paints in front of its own parent). A point inside `A` but genuinely outside `B`'s own bounds, `(250, 25)`, resolves to red in *both* the naive and fixed versions — confirming the bug specifically affects the overlap region, not areas where only the parent's background was ever going to be visible anyway.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A flat, ordered display list, painted strictly in sequence	Chapter 8's own software rasterizer — this exact display list is the direct input; the rasterizer's own job is simply to execute each command in order onto a real pixel buffer
Background painting out to the border edge, using `border_box()`	Chapter 2's own box-model machinery, reused unchanged — this chapter needed no new geometry code at all, only a new way of walking the tree that was already fully built
Paint order matching nesting depth, verified at three levels deep	Chapter 9's own stacking and paint-order chapter, which will need to handle real CSS cases (like `z-index`) where paint order and DOM nesting order genuinely diverge — this chapter's own "parent first, then children" rule is the honest, simpler default that real CSS uses whenever nothing overrides it

Hands-On Exercises

EXERCISE 1

Build a container with a single child whose own `background-color` is left at the default `transparent`. Run it through `build_display_list()` and confirm the resulting list is completely empty — not a command with a transparent color, genuinely no command at all — then explain which specific line in `paint_background()` is responsible.

View solution

EXERCISE 2

Build a genuinely three-level-deep nested layout — an outer box, a middle box inside it, and an inner box inside the middle one — each with its own distinct background color. Run it through `build_display_list()` and confirm the resulting paint order matches the nesting depth exactly: outermost first, innermost last.

View solution

EXERCISE 3

Reproduce this chapter's own two-box bug directly: lay out box A (background:red) containing box B (background:blue) using the real layout pipeline, then build display lists with both `build_display_list_naive()` and `build_display_list()`. Compute `color_at_point()` for a point inside both boxes and confirm the two versions disagree. Identify the exact two-line reordering responsible for the difference.

View solution

CHAPTER 7 QUICK REFERENCE

- **Display list:** a flat, ordered sequence of simple paint commands, built by walking the finished layout tree

- **Background painting:** paints to the *border* box, not the content box — reuses Chapter 2's own `border_box()` unchanged
- **Real bug found and fixed:** recursing into children before appending a box's own background command puts the parent's background LATER (painted on top) — verified completely hiding a real child's background wherever the two overlap
- **The fix:** paint a box's own background first, then recurse — matching real CSS paint order (parent, then children, outermost to innermost)
- **Verified:** the fix holds at three levels of nesting, not just two, and a transparent background genuinely produces no command at all
- **Next chapter:** A Software Rasterizer — turning this display list into an actual pixel buffer, verified pixel by pixel

CHAPTER 8 OF 10

A Software Rasterizer: Display List to Pixels

Building a Web Browser Engine: Layout & Rendering

Chapter 8 · A Software Rasterizer: Display List to Pixels

Chapter 7 produced a real, ordered list of paint commands. This chapter executes it — a real pixel buffer, filled one command at a time, that can be checked pixel by pixel and written out as an actual, viewable image file. This is the literal end of the pipeline this entire two-course project has been building toward.

A Canvas, a Color Table, and the Naive First Attempt

```
class Canvas:
    def __init__(self, width, height, background=(255, 255, 255)):
        self.width = width
        self.height = height
        self.pixels = [[background for _ in range(width)] for _ in range(height)]

NAMED_COLORS = {
    'red': (255,0,0), 'blue': (0,0,255), 'navy': (0,0,128), ...
}
DEFAULT_COLOR = (0, 0, 0)

def parse_color(name):
    return NAMED_COLORS.get(name, DEFAULT_COLOR)

def fill_rect_naive(canvas, rect, color):
    x0, y0 = int(rect.x), int(rect.y)
    x1, y1 = int(rect.x + rect.width), int(rect.y + rect.height)
    for y in range(y0, y1):
        for x in range(x0, x1):
            canvas.pixels[y][x] = color   # no bounds checking at all
```

A real, legitimate concern going in: rects computed by earlier chapters routinely have negative `x` / `y` — a margin box's own `expand_rect_by` (Chapter 2) subtracts `edge.left` / `edge.top`, and that can go negative. What actually happens when `fill_rect_naive` meets one turns out to be worse than a simple crash.

Two Different Failure Modes, One Root Cause

VERIFIED DIRECTLY — A RECT EXTENDING OFF THE LEFT EDGE SILENTLY CORRUPTS THE RIGHT EDGE INSTEAD

A 10×10 canvas, white. `fill_rect_naive(canvas, Rect(-2, 0, 4, 1), red)`. Row 0 afterward: columns `0` and `1` are correctly red — the part of the rect genuinely on-canvas. But columns `8` **and** `9` **are ALSO red**. `canvas.pixels[0][-2]` and `[0][-1]` are completely valid Python indexing — negative indices wrap to count from the *end* of the list — so the naive rasterizer silently painted the far right edge of the canvas, which the intended rect never touched at all. Not a crash: a real, silent, visually confusing corruption of pixels that have nothing to do with the rect that caused it.

VERIFIED DIRECTLY — A RECT EXTENDING PAST THE BOTTOM OR RIGHT EDGE GENUINELY CRASHES INSTEAD

The same 10×10 canvas, `fill_rect_naive(canvas, Rect(8, 8, 5, 5), blue)`. This one raises a real `IndexError: list assignment index out of range` the moment `y` reaches `10`. The identical root cause — no bounds checking — produces two genuinely *different* failure modes depending on which direction the rect overruns: silent corruption on the negative side, a hard crash on the positive side.

```
def fill_rect(canvas, rect, color):
    x0 = max(0, int(rect.x))
    y0 = max(0, int(rect.y))
    x1 = min(canvas.width, int(rect.x + rect.width))
    y1 = min(canvas.height, int(rect.y + rect.height))
    for y in range(y0, y1):
        for x in range(x0, x1):
            canvas.pixels[y][x] = color
```

VERIFIED DIRECTLY — ONE FIX RESOLVES BOTH FAILURE MODES AT ONCE

Clipping `x0` / `y0` / `x1` / `y1` against the canvas's own bounds *before* the fill loop ever starts means `canvas.pixels` is never indexed with a negative or out-of-range coordinate at all. The same `Rect(-2, 0, 4, 1)` now leaves columns 8 and 9 correctly untouched. The same `Rect(8, 8, 5, 5)` no longer crashes, and correctly paints only the pixels that genuinely sit on-canvas.

Rasterizing Chapter 7's Own Real Example, and Writing a Real Image File

VERIFIED DIRECTLY — THE ACTUAL PIXEL BUFFER MATCHES CHAPTER 7'S OWN COLOR_AT_POINT() SIMULATION EXACTLY

Chapter 7's own real red-A-containing-blue-B layout, rasterized onto a real 300×50 canvas: pixel `(50, 25)` — inside both boxes — comes out `(0, 0, 255)`, genuine blue. Pixel `(250, 25)` — inside A only — comes out `(255, 0, 0)`, genuine red. The real, physically-filled pixel buffer agrees exactly with the abstract simulation Chapter 7 used to verify paint order.

`save_ppm()` writes the canvas out as a real, valid, dependency-free image — the plain-text PPM (P3) format: a three-line header, then one `R G B` line per pixel. No image library needed, and the result is a genuinely viewable file.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Negative rect coordinates being a real, expected input, not an edge case	Chapter 2's own `expand_rect_by` — margin boxes routinely produce negative `x` / `y`, meaning this chapter's own bug wasn't hypothetical; it was guaranteed to be hit by real output from earlier chapters
Two different failure modes from one root cause	Chapter 3's own overflow bug and Chapter 5's own line-overflow bug — a recurring pattern across this whole course: skipping a bounds check doesn't fail loudly and uniformly, it fails differently depending on which direction the bound is violated
A real pixel buffer matching Chapter 7's own abstract simulation	The literal completion of this course's own stated four-stage pipeline from Chapter 1 — parse (Course 1) → style (Course 1) → layout → paint → *this chapter* — an actual image, not just a data structure claiming to represent one

Hands-On Exercises

EXERCISE 1

Call `parse_color('cornflowerblue')` — a real CSS color keyword, but one not present in this chapter's own simplified `NAMED_COLORS` table. Confirm it falls back to a sensible default rather than raising an error, and explain why a hand-built color table can never realistically cover every one of CSS's 140+ named colors.

View solution

EXERCISE 2

Fill a 10×10 canvas with a rect that sits entirely outside the canvas's own bounds on all sides, e.g. `Rect(20, 20, 5, 5)`. Confirm every single pixel on the canvas remains at its original background color, and explain why this doesn't require a special "rect is fully off-canvas" check anywhere in `fill_rect`'s own code.

View solution

EXERCISE 3

Reproduce this chapter's own two failure modes directly: call `fill_rect_naive` with `Rect(-2, 0, 4, 1)` on a 10×10 canvas and inspect row 0's own pixel values at columns 8 and 9; separately, call `fill_rect_naive` with `Rect(8, 8, 5, 5)` and confirm it raises `IndexError`. For each case, explain in your own words why Python's own indexing rules produce that SPECIFIC outcome (silent corruption vs. a crash) rather than the other way around.

View solution

CHAPTER 8 QUICK REFERENCE

- **Canvas:** a real 2D pixel buffer, plus a simplified named-color table with a black fallback for unrecognized names
- **Real bug found and fixed:** no bounds clipping produces TWO different failure modes from the same root cause — negative coordinates silently wrap and corrupt the opposite edge of the canvas; positive out-of-range coordinates genuinely crash with `IndexError`
- **The fix:** clip a rect's own bounds against the canvas's own bounds before the fill loop ever runs — one change resolves both failure modes
- **Verified:** the real, physically-rasterized pixel buffer matches Chapter 7's own abstract paint-order simulation exactly, on the same real example
- **A real, dependency-free image file:** the PPM (P3) format needs no library at all — a genuinely viewable image, written directly to disk
- **Next chapter:** Stacking, Paint Order & Overlapping Elements — the cases where DOM nesting order and real visual paint order genuinely diverge

CHAPTER 9 OF 10

Stacking, Paint Order & Overlapping Elements

Building a Web Browser Engine: Layout & Rendering

Chapter 9 · Stacking, Paint Order & Overlapping Elements

This engine has no `position: absolute`, no floats — normal-flow block layout is all it knows. So how can two boxes ever genuinely *overlap* at all? There's exactly one way already fully supported: a **negative margin**. This chapter uses that as the real, concrete overlap case, then builds a real (but deliberately narrow) `z-index` feature on top of it.

SCOPE NOTE

Real CSS stacking contexts are a genuinely deep subject — `position`, `opacity`, `transform`, and `z-index` together determine when a *new* stacking context even exists, and z-index values only ever compare *within* one. This chapter builds one honest, narrow slice: `z-index` re-sorting a single box's own direct children, nothing more.

Negative-Margin Overlap: Verified Working, With Zero New Code

Two siblings under one container: `top_box` (100×40, red), `bottom_box` (100×40, blue, `margin: -20px 0 0 0`).

VERIFIED DIRECTLY — THE OVERLAP IS REAL, AND REQUIRED NO NEW LAYOUT CODE AT ALL

`top_box.border_box() = Rect(0, 0, 100, 40)`. `bottom_box.border_box() = Rect(0, 20, 100, 40)` — genuinely overlapping `top_box`'s own bottom 20 pixels. Chapter 2's `expand_rect_by()` and Chapter 3's own position/height-accumulation formulas are pure addition and subtraction — nothing in them ever assumed a margin had to be positive. A negative value simply flows through the same math correctly, by construction.

VERIFIED DIRECTLY — THE EXISTING PAINT-ORDER RULE ALREADY HANDLES THIS OVERLAP CORRECTLY

Checking the actually-visible color at a point inside the real overlap band, `(50, 30)`: **blue**. `bottom_box`, later in source order, correctly paints on top of `top_box` — Chapter 7's own source-order rule, unmodified, already gets this right. No z-index needed for the default case at all.

A Real z-index — Scoped to Siblings, Never Across Parent/Child

```
def get_z_index(layout_box):
    style = box_style(layout_box)
    z = style.get('z-index', '0')
    try:
        return int(float(z))
    except (ValueError, TypeError):
        return 0

def build_display_list_zindex(layout_box, display_list=None):
    if display_list is None:
        display_list = []
    paint_background(layout_box, display_list)                      # THIS box's own bg, uncondition
    ordered_children = sorted(layout_box.children, key=get_z_index)   # SIBLINGS ONLY, stable sort
    for child in ordered_children:
        build_display_list_zindex(child, display_list)
    return display_list
```

VERIFIED DIRECTLY — HIGHER Z-INDEX PAINTS ON TOP, EVEN AGAINST SOURCE ORDER

Two siblings, source order `[x(z-index:1), y(z-index:0)]`. Paint order after the sort: `['yellow', 'green']` — `y` paints first despite coming second in the source, `x` paints last (on top), matching real CSS: z-index reorders paint, independent of DOM order.

VERIFIED DIRECTLY — EQUAL Z-INDEX SIBLINGS KEEP THEIR ORIGINAL SOURCE ORDER

Two siblings, both left at the default `z-index: 0`: paint order comes out `['orange', 'purple']` — exactly matching source order, with no special tie-breaking code needed at all. Python's own `sorted()` is a *stable* sort: equal keys never change relative order.

A Real Bug: Sorting Globally Reintroduces Chapter 7's Own Bug

```
def build_display_list_naive_global_zindex(root):
    all_boxes = collect_all_boxes(root)              # flattens the WHOLE tree, ignoring structure
    all_boxes_sorted = sorted(all_boxes, key=get_z_index)
    display_list = []
    for box in all_boxes_sorted:
        paint_background(box, display_list)
    return display_list
```

A plausible-looking shortcut: just collect every box in the whole tree and sort them *all* by z-index, once. Real CSS z-index never works this way — it only ever compares elements *within the same stacking context*, never an arbitrary parent against its own child.

VERIFIED DIRECTLY — A PARENT'S OWN HIGH Z-INDEX MAKES IT PAINT OVER ITS OWN CHILD, HIDING IT COMPLETELY

Parent `P` (`background: red`, `z-index: 5`) containing child `C` (`background: blue`, default `z-index: 0`). The global sort puts `P`'s own entry *after* `C`'s, purely because `5 > 0` — the display list comes out `['blue', 'red']`. Checking a point inside both boxes: **red**. `P` paints completely over its own child, hiding it — this is **the exact same visual bug Chapter 7 found and fixed** (a parent's background covering its own content), reintroduced here through a totally different mechanism: not "children before self," but "global z-index sort that doesn't know or care about tree structure at all."

VERIFIED DIRECTLY — SIBLING-SCOPED SORTING FIXES IT, REGARDLESS OF EITHER BOX'S OWN Z-INDEX VALUE

Same two boxes, through `build_display_list_zindex()`: `['red', 'blue']`. The same point now correctly resolves to **blue**. `paint_background(layout_box, display_list)` runs *unconditionally*, before `P`'s own children are ever sorted or recursed into — `P` and `C` are never compared to each other by z-index at all, because they're never siblings of one another.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Negative margins working correctly with zero new code	Chapter 4's own auto-margin algorithm — `parse_length` has always happily returned negative floats for a value like `"-20px"`; nothing anywhere in this course's own layout math ever assumed non-negative margins, so this "just worked" the moment it was tested
z-index reintroducing Chapter 7's own exact bug via a new mechanism	Chapter 6's own font-size bug and Chapter 8's own bounds-clipping bug — a recurring pattern across this whole course: the same class of visual mistake (something painting over, or measuring past, what it should) keeps resurfacing through genuinely different code paths, not because earlier fixes were wrong, but because each new feature is a new opportunity to reintroduce an old category of mistake
z-index scoped strictly to direct siblings	The honest boundary of this whole course's own stacking model — real CSS stacking contexts, `position`-triggered new contexts, and isolation rules are named directly as out of scope, not silently ignored

Hands-On Exercises

EXERCISE 1

Build three siblings with z-index values 2, 0, and 1 respectively (in that source order), each with a distinct background color. Run them through `build_display_list_zindex()` and confirm the resulting paint order is sorted purely by z-

index (lowest painted first), independent of the original source order.

View solution

EXERCISE 2

Build a box with an invalid `z-index` value, e.g. `'auto'` (a real CSS keyword this course's own simplified model doesn't parse as a number). Call `get_z_index()` on it directly and confirm it falls back to `0` rather than raising an error, and explain which part of the function is responsible.

View solution

EXERCISE 3

Reproduce this chapter's own global-z-index bug directly: build a parent with `z-index: 5` containing a child with default `z-index: 0`, lay it out for real, and build display lists with both `build_display_list_naive_global_zindex()` and `build_display_list_zindex()`. Confirm the two disagree at a point inside both boxes, and explain specifically why `collect_all_boxes()` is the root of the problem, even though `get_z_index()` itself is computing perfectly correct values.

View solution

CHAPTER 9 QUICK REFERENCE

- **The only real overlap this engine supports:** negative margins — verified working correctly with zero new layout code, since the existing formulas never assumed margins were positive
- **Default stacking:** source order — a later sibling paints on top of an earlier one, already correct via Chapter 7's own rule, with no z-index needed
- **z-index, correctly scoped:** re-sorts a single box's own direct children only — never compares a parent against its own descendants
- **Real bug found and fixed:** sorting the ENTIRE tree by z-index globally reintroduces Chapter 7's own exact "parent paints over child" bug, since a parent's own z-index can outrank its child's in a flat, structure-blind sort
- **Verified:** higher z-index correctly overrides source order among siblings; equal z-index siblings keep source order via Python's own stable sort
- **Next chapter:** Capstone — rendering a real web page end to end, all the way to a pixel-checked image

CHAPTER 10 OF 10

Capstone — Rendering a Real Web Page End to End

Building a Web Browser Engine: Layout & Rendering

Chapter 10 · Capstone — Rendering a Real Web Page End to End

This is the moment every chapter across both courses has been building toward: one real HTML document, one real stylesheet, run through the *entire* pipeline — tokenize, parse, build a style tree (Course 1) — then layout tree, block layout, a display list, and a real, physically rasterized pixel buffer (Course 2) — with nothing hand-faked at any stage.

The Document: Course 1's Own Capstone, With Its CSS Honestly Extended

The HTML is exactly Course 1's own real capstone document, unchanged. Its original CSS, though, never set a single `background-color` — nothing for Course 2's own rasterizer to actually paint. The stylesheet below keeps every one of Course 1's own original rules and **openly adds** a few real background colors and one genuine negative-margin overlap, specifically to give this final chapter something worth rendering.

```
/* Course 1's own original rules, unchanged: */
#page { color: navy; }
.intro { font-size: 20px; }
.hidden { display: none; }
p { margin: 24px 0; }

/* added for this capstone, to exercise Course 2's own real capabilities: */
#page { background-color: yellow; padding: 20px; width: 360px; }
p     { background-color: blue; height: 40px; }
h1    { background-color: orange; height: 30px; }
ul    { background-color: green; margin: -34px 0 0 0; height: 30px; }
```

A Real, Honest Finding — Discovered Live, Building This Capstone

VERIFIED DIRECTLY — TEXT-ONLY CONTENT CONTRIBUTES ZERO HEIGHT WITHOUT AN EXPLICIT HEIGHT SET

The very first run of this capstone, before `h1 { height: 30px; }` was added, gave `h1`'s own border box a height of exactly `0` — despite `h1` containing the real word "Welcome." Chapter 5 built real line-breaking, and Chapter 6 built real font-metric measurement — but neither was ever wired into `layout_block`'s own height computation

(Chapter 3), which only ever sums a block box's own *block-level children's* margin-box heights. An anonymous box holding pure text was never given a real content height anywhere in this course. This is a genuine, honest scope boundary of this specific two-course project — not a bug to silently patch here, but a real limitation worth stating plainly: connecting Chapter 5/6's own text metrics to a block box's own height is exactly the kind of integration work a genuinely complete engine would still need.

Verifying the Full Pipeline, Stage by Stage

VERIFIED DIRECTLY — COURSE 1'S REAL STYLE TREE CORRECTLY RESOLVES THE NEW BACKGROUND-COLOR

`styled_root.style['background-color']` for `#page`: `'yellow'` — Course 1's own cascade, unmodified, correctly picks up a property this document's original CSS never used at all.

VERIFIED DIRECTLY — REAL BLOCK LAYOUT, WIDTH CONSTRAINED BY THE EXPLICIT 360PX, HEIGHT BY EXPLICIT VALUES

`#page` border box: `Rect(0, 0, 400, 154)`. `h1`: `Rect(20, 20, 360, 30)`. `p`: `Rect(20, 74, 360, 40)`. `ul`: `Rect(20, 104, 360, 30)`. Every number here is Course 2's own real width/position/height algorithm (Chapters 2–4), computed from real cascaded style values, not asserted.

VERIFIED DIRECTLY — A GENUINE NEGATIVE-MARGIN OVERLAP, CORRECTLY PAINTED, ON A REAL FULL-PAGE LAYOUT

`ul`'s own `-34px` top margin pulls its border box up to `y=104` — genuinely overlapping `p`'s own border box, which ends at `y=114`. Checking the actual rasterized pixel color inside that real overlap band: **green** — `ul`, later in source order, correctly paints on top of `p`, exactly matching Chapter 9's own established rule, now confirmed on a real, full document rather than an isolated two-box example.

VERIFIED DIRECTLY — LI.HIDDEN'S ABSENCE HOLDS ALL THE WAY TO THE FINAL PIXEL BUFFER

Counting `ul`'s own block-level children in the *final layout tree*: exactly **1**. `li.hidden`, excluded back at Course 1's own style-tree stage (`display: none`), never reaches the layout tree, the display list, or the rasterizer — confirmed at the very last stage of the entire two-course pipeline, not just where it was originally removed.

A real, valid PPM image file was written to disk from the finished pixel buffer — a genuine, viewable picture of a real web page, produced entirely by code built chapter by chapter across two full courses.

An Honest Comparison Against a Real Browser

A live browser wasn't available while writing this chapter, so this comparison is grounded in well-documented, uncontroversial CSS specification behavior rather than a literal pixel diff — and a pixel diff wouldn't be a fair comparison anyway, for a reason stated back in Course 1 Chapter 1: **real font rendering was always out of this project's own stated scope.** This engine never draws a single letter; a real browser draws real glyphs,

with real kerning, hinting, and anti-aliasing this engine's own 8px-per-character placeholder was never meant to visually match.

CLAIM THIS ENGINE MAKES	REAL, DOCUMENTED CSS BEHAVIOR
`#page`, `p`, `h1`, `ul`, `li` default to block; `b` defaults to inline	Matches every real browser's own default (user-agent) stylesheet exactly — this is standard, specified behavior, not engine-specific
Background paints to the border edge, not just the content box	Matches the CSS box model specification exactly — `background-clip`'s own real initial value is `border-box`
A later sibling paints on top of an earlier one in the same stacking context	Matches the real CSS painting-order specification's own default (no z-index / positioning override) exactly
`display: none` removes an element and its descendants from layout entirely	Matches real browsers exactly — a real browser also gives such an element zero `getBoundingClientRect()` size and no box at all

What This Project Doesn't Cover

Restating the scope drawn back in Course 1 Chapter 1, now that the whole thing has actually been built: no JavaScript, no networking beyond an in-memory string, no images, no Flexbox or Grid, no forms, no real font rendering, and no real GPU/OS rendering surface (a hand-written software rasterizer instead). Beyond that original list, building this project surfaced several more, more specific boundaries worth naming honestly: no `position: absolute/relative/fixed` (only negative-margin overlap and normal flow); no real CSS stacking contexts (z-index is scoped to direct siblings only); no `box-sizing: border-box`; and — found live, in this very chapter — no integration between real text measurement (Chapters 5–6) and a block box's own computed height.

BUILDING A WEB BROWSER ENGINE — BOTH COURSES, COMPLETE

- **Course 1, Parsing & the DOM (10 chapters):** a real HTML tokenizer and DOM parser, a real CSS tokenizer/parser and selector matcher, the cascade, inheritance, the style tree, and a real user-agent stylesheet — five genuine bugs found and fixed along the way, closing with a capstone that found and fixed a sixth (a cross-chapter `Declaration.property` / `.name` mismatch) by wiring every piece together for the first time
- **Course 2, Layout & Rendering (10 chapters):** the layout tree, the box model, real block layout with the full constraint-based width algorithm, inline line-breaking, real font metrics, painting, a software rasterizer, and a scoped stacking model — six more genuine bugs found and fixed, closing with this chapter's own honest, live-discovered scope boundary
- **Twelve real bugs, found and fixed, across twenty chapters** — every one verified with real, hand-run Python before being written up, matching this site's own established rigor for every course in this "ambitious learning projects" tier

- **The end result:** a real HTML document and a real stylesheet, producing a real, pixel-checked image, written to a real file on disk — a genuinely complete, if deliberately scoped-down, browser engine, built from nothing but a first-principles understanding of how the real thing works