Protocol

Layout

A type that defines the geometry of a collection of views.

iOS 16.0+ | iPadOS 16.0+ | Mac Catalyst 16.0+ | macOS 13.0+ | tvOS 16.0+ | visionOS 1.0+ | watchOS 9.0+

```
@preconcurrency
protocol Layout : Sendable, Animatable
```

Overview

You traditionally arrange views in your app's user interface using built-in layout containers like `HStack` and `Grid`. If you need more complex layout behavior, you can define a custom layout container by creating a type that conforms to the `Layout` protocol and implementing its required methods:

- `sizeThatFits(proposal:subviews:cache:)` reports the size of the composite layout view.
- `placeSubviews(in:proposal:subviews:cache:)` assigns positions to the container's subviews.

You can define a basic layout type with only these two methods:

```
struct BasicVStack: Layout {
    func sizeThatFits(
        proposal: ProposedViewSize,
        subviews: Subviews,
        cache: inout ()
    ) -> CGSize {
        // Calculate and return the size of the layout container.
    }
```

```
    func placeSubviews(
        in bounds: CGRect,
        proposal: ProposedViewSize,
        subviews: Subviews,
        cache: inout ()
    ) {
        // Tell each subview where to appear.
    }
}
```

Use your layout the same way you use a built-in layout container, by providing a `ViewBuilder` with the list of subviews to arrange:

```
BasicVStack {
    Text("A Subview")
    Text("Another Subview")
}
```

Support additional behaviors

You can optionally implement other protocol methods and properties to provide more layout container features:

- Define explicit horizontal and vertical layout guides for the container by implementing `explicitAlignment(of:in:proposal:subviews:cache:)` for each dimension.
- Establish the preferred spacing around the container by implementing `spacing(subviews:cache:)`.
- Indicate the axis of orientation for a container that has characteristics of a stack by implementing the `layoutProperties` static property.
- Create and manage a cache to store computed values across different layout protocol calls by implementing `makeCache(subviews:)`.

The protocol provides default implementations for these symbols if you don't implement them. See each method or property for details.

Add input parameters

You can define parameters as inputs to the layout, like you might for a `View`:

```
struct BasicVStack: Layout {
    var alignment: HorizontalAlignment

    // ...
}
```

Set the parameters at the point where you instantiate the layout:

```
BasicVStack(alignment: .leading) {
    // ...
}
```

If the layout provides default values for its parameters, you can omit the parameters at the call site, but you might need to keep the parentheses after the name of the layout, depending on how you specify the defaults. For example, suppose you set a default alignment for the basic stack in the parameter declaration:

```
struct BasicVStack: Layout {
    var alignment: HorizontalAlignment = .center

    // ...
}
```

To instantiate this layout using the default center alignment, you don't have to specify the alignment value, but you do need to add empty parentheses:

```
BasicVStack() {
    // ...
}
```

The Swift compiler requires the parentheses in this case because of how the layout protocol implements this call site syntax. Specifically, the layout's `callAsFunction(_:)` method looks for an initializer with exactly zero input arguments when you omit the parentheses from the call site. You can enable the simpler call site for a layout that doesn't have an implicit initializer of this type by explicitly defining one:

```
init() {
    self.alignment = .center
}
```

For information about Swift initializers, see Initialization in *The Swift Programming Language*.

Interact with subviews through their proxies

To perform layout, you need information about all of its subviews, which are the views that your container arranges. While your layout can't interact directly with its subviews, it can access a set of subview proxies through the `Layout.Subviews` collection that each protocol method receives as an input parameter. That type is an alias for the `LayoutSubviews` collection type, which in turn contains `LayoutSubview` instances that are the subview proxies.

You can get information about each subview from its proxy, like its dimensions and spacing preferences. This enables you to measure subviews before you commit to placing them. You also assign a position to each subview by calling its proxy's `place(at:anchor:proposal:)` method. Call the method on each subview from within your implementation of the layout's `placeSubviews(in:proposal:subviews:cache:)` method.

Access layout values

Views have layout values that you set with view modifiers. Layout containers can choose to condition their behavior accordingly. For example, a built-in `HStack` allocates space to its subviews based in part on the priorities that you set with the `layoutPriority(_:)` view modifier. Your layout container accesses this value for a subview by reading the proxy's `priority` property.

You can also create custom layout values by creating a layout key. Set a value on a view with the `layoutValue(key:value:)` view modifier. Read the corresponding value from the subview's proxy using the key as an index on the subview. For more information about creating, setting, and accessing custom layout values, see `LayoutValueKey`.

Topics

Sizing the container and placing subviews

```
func sizeThatFits(proposal: ProposedViewSize, subviews: Self.Subviews, cache: inout Self.Cache) -> CGSize
```

Returns the size of the composite view, given a proposed size and the view's subviews.

Required

```
func placeSubviews(in: CGRect, proposal: ProposedViewSize, subviews: Self.Subviews, cache: inout Self.Cache)
```

Assigns positions to each of the layout's subviews.

Required

`typealias Subviews`

A collection of proxies for the subviews of a layout view.

Reporting layout container characteristics

`func explicitAlignment(of:in:proposal:subviews:cache:)`

Returns the position of the specified horizontal alignment guide along the x axis.

Required Default implementations provided.

`func spacing(subviews: Self.Subviews, cache: inout Self.Cache) -> ViewSpacing`

Returns the preferred spacing values of the composite view.

Required Default implementation provided.

`static var layoutProperties: LayoutProperties`

Properties of a layout container.

Required Default implementation provided.

Managing a cache

`func makeCache(subviews: Self.Subviews) -> Self.Cache`

Creates and initializes a cache for a layout instance.

Required Default implementation provided.

`func updateCache(inout Self.Cache, subviews: Self.Subviews)`

Updates the layout's cache when something changes.

Required Default implementation provided.

`associatedtype Cache = Void`

Cached values associated with the layout instance.

Required

Supporting types

`func callAsFunction<V>(() -> V) -> some View`

Combines the specified views into a single composite view using the layout algorithms of the custom layout container.

Instance Methods

`func depthAlignment(DepthAlignment) -> some Layout`

Sets the depth alignment for this layout.

`func depthAlignment<Content>(DepthAlignment, content: () -> Content) -> some View`

Creates a layout view with the specified depth alignment.

Relationships

Inherits From

`Animatable`, `Sendable`, `SendableMetatype`

Conforming Types

`AnyLayout`
`GridLayout`
`HStackLayout`
`SpatialContainer`
`VStackLayout`
`ZStackLayout`

See Also

Creating a custom layout container

{} Composing custom layouts with SwiftUI

Arrange views in your app's interface using layout tools that SwiftUI provides.

`struct LayoutSubview`

A proxy that represents one subview of a layout.

```
struct LayoutSubviews
```

A collection of proxy values that represent the subviews of a layout view.