orphan:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\ABI\[swift-main][docs][ABI]TypeLayout.rst`, line 5)

Unknown directive type "highlight".

```
.. highlight:: none
```

Type Layout

Hard Constraints on Resilience

The root of a class hierarchy must remain stable, at pain of invalidating the metaclass hierarchy. Note that a Swift class without an explicit base class is implicitly rooted in the SwiftObject Objective-C class.

Fragile Struct and Tuple Layout

Structs and tuples currently share the same layout algorithm, noted as the "Universal" layout algorithm in the compiler implementation. The algorithm is as follows:

- Start with a **size** of **0** and an **alignment** of **1**.
- Iterate through the fields, in element order for tuples, or in `var` declaration order for structs. For each field:
  - Update **size** by rounding up to the **alignment of the field**, that is, increasing it to the least value greater or equal to **size** and evenly divisible by the **alignment of the field**.
  - Assign the **offset of the field** to the current value of **size**.
  - Update **size** by adding the **size of the field**.
  - Update **alignment** to the max of **alignment** and the **alignment of the field**.
- The final **size** and **alignment** are the size and alignment of the aggregate. The **stride** of the type is the final **size** rounded up to **alignment**.

Note that this differs from C or LLVM's normal layout rules in that *size* and *stride* are distinct; whereas C layout requires that an embedded struct's size be padded out to its alignment and that nothing be laid out there, Swift layout allows an outer struct to lay out fields in the inner struct's tail padding, alignment permitting. Unlike C, zero-sized structs and tuples are also allowed, and take up no storage in enclosing aggregates. The Swift compiler emits LLVM packed struct types with manual padding to get the necessary control over the binary layout. Some examples:

```
// LLVM <{ i64, i8 }>
struct S {
  var x: Int
  var y: UInt8
}

// LLVM <{ i8, [7 x i8], <{ i64, i8 }>, i8 }>
struct S2 {
  var x: UInt8
  var s: S
  var y: UInt8
}

// LLVM <{}>
struct Empty {}

// LLVM <{ i64, i64 }>
struct ContainsEmpty {
  var x: Int
  var y: Empty
  var z: Int
}
```

Class Layout

Swift relies on the following assumptions about the Objective-C runtime, which are therefore now part of the Objective-C ABI:

- 32-bit platforms never have tagged pointers. ObjC pointer types are either nil or an object pointer.
- On x86-64, a tagged pointer either sets the lowest bit of the pointer or the highest bit of the pointer. Therefore, both of these bits are zero if and only if the value is not a tagged pointer.
- On ARM64, a tagged pointer always sets the highest bit of the pointer.
- 32-bit platforms never perform any isa masking. `object_getClass` is always equivalent to `*(Class*)object`.
- 64-bit platforms perform isa masking only if the runtime exports a symbol `uintptr_t objc_debug_isa_class_mask;`. If this symbol is exported, `object_getClass` on a non-tagged pointer is always equivalent to `(Class)(objc_debug_isa_class_mask & *(uintptr_t*)object)`.
- The superclass field of a class object is always stored immediately after the isa field. Its value is either nil or a pointer to the

class object for the superclass; it never has other bits set.

The following assumptions are part of the Swift ABI:

- Swift class pointers are never tagged pointers.

TODO

Fragile Enum Layout

In laying out enum types, the ABI attempts to avoid requiring additional storage to store the tag for the enum case. The ABI chooses one of five strategies based on the layout of the enum:

Empty Enums

In the degenerate case of an enum with no cases, the enum is an empty type.

```
enum Empty {} // => empty type
```

Single-Case Enums

In the degenerate case of an enum with a single case, there is no discriminator needed, and the enum type has the exact same layout as its case's data type, or is empty if the case has no data type.

```
enum EmptyCase { case X }             // => empty type
enum DataCase { case Y(Int, Double) } // => LLVM <{ i64, double }>
```

C-Like Enums

If none of the cases has a data type (a "C-like" enum), then the enum is laid out as an integer tag with the minimal number of bits to contain all of the cases. The machine-level layout of the type then follows LLVM's data layout rules for integer types on the target platform. The cases are assigned tag values in declaration order.

```
enum EnumLike2 { // => LLVM i1
  case A         // => i1 0
  case B         // => i1 1
}

enum EnumLike8 { // => LLVM i3
  case A         // => i3 0
  case B         // => i3 1
  case C         // => i3 2
  case D         // etc.
  case E
  case F
  case G
  case H
}
```

Discriminator values after the one used for the last case become *extra inhabitants* of the enum type (see Single-Payload Enums).

Single-Payload Enums

If an enum has a single case with a data type and one or more no-data cases (a "single-payload" enum), then the case with data type is represented using the data type's binary representation, with added zero bits for tag if necessary. If the data type's binary representation has **extra inhabitants**, that is, bit patterns with the size and alignment of the type but which do not form valid values of that type, they are used to represent the no-data cases, with extra inhabitants in order of ascending numeric value matching no-data cases in declaration order. If the type has *spare bits* (see Multi-Payload Enums), they are used to form extra inhabitants. The enum value is then represented as an integer with the storage size in bits of the data type. Extra inhabitants of the payload type not used by the enum type become extra inhabitants of the enum type itself.

```
enum CharOrSectionMarker { => LLVM i32
  case Paragraph            => i32 0x0020_0000
  case Char(UnicodeScalar)  => i32 (zext i21 %Char to i32)
  case Chapter              => i32 0x0020_0001
}

CharOrSectionMarker.Char('\x00') => i32 0x0000_0000
CharOrSectionMarker.Char('\u10FFFF') => i32 0x0010_FFFF

enum CharOrSectionMarkerOrFootnoteMarker { => LLVM i32
  case CharOrSectionMarker(CharOrSectionMarker) => i32 %CharOrSectionMarker
  case Asterisk                                 => i32 0x0020_0002
  case Dagger                                   => i32 0x0020_0003
  case DoubleDagger                             => i32 0x0020_0004
}
```

If the data type has no extra inhabitants, or there are not enough extra inhabitants to represent all of the no-data cases, then a tag bit is added to the enum's representation. The tag bit is set for the no-data cases, which are then assigned values in the data area of the enum in declaration order.

```
enum IntOrInfinity { => LLVM <{ i64, i1 }>
  case NegInfinity    => <{ i64, i1 }> {    0, 1 }
  case Int(Int)       => <{ i64, i1 }> { %Int, 0 }
  case PosInfinity    => <{ i64, i1 }> {    1, 1 }
}

IntOrInfinity.Int(    0) => <{ i64, i1 }> {     0, 0 }
IntOrInfinity.Int(20721) => <{ i64, i1 }> { 20721, 0 }
```

Multi-Payload Enums

If an enum has more than one case with data type, then a tag is necessary to discriminate the data types. The ABI will first try to find common **spare bits**, that is, bits in the data types' binary representations which are either fixed-zero or ignored by valid values of all of the data types. The tag will be scattered into these spare bits as much as possible. Currently only spare bits of primitive integer types, such as the high bits of an `i21` type, are considered. The enum data is represented as an integer with the storage size in bits of the largest data type.

```
enum TerminalChar {              => LLVM i32
  case Plain(UnicodeScalar)      => i32     (zext i21 %Plain     to i32)
  case Bold(UnicodeScalar)       => i32 (or (zext i21 %Bold      to i32), 0x0020_0000)
  case Underline(UnicodeScalar)  => i32 (or (zext i21 %Underline to i32), 0x0040_0000)
  case Blink(UnicodeScalar)      => i32 (or (zext i21 %Blink     to i32), 0x0060_0000)
  case Empty                     => i32 0x0080_0000
  case Cursor                    => i32 0x0080_0001
}
```

If there are not enough spare bits to contain the tag, then additional bits are added to the representation to contain the tag. Tag values are assigned to data cases in declaration order. If there are no-data cases, they are collected under a common tag, and assigned values in the data area of the enum in declaration order.

```
class Bignum {}

enum IntDoubleOrBignum { => LLVM <{ i64, i2 }>
  case Int(Int)           => <{ i64, i2 }> {           %Int,            0 }
  case Double(Double)     => <{ i64, i2 }> { (bitcast  %Double to i64), 1 }
  case Bignum(Bignum)     => <{ i64, i2 }> { (ptrtoint %Bignum to i64), 2 }
}
```

Existential Container Layout

Values of protocol type, protocol composition type, or `Any` type are laid out using **existential containers** (so-called because these types are "existential types" in type theory).

Opaque Existential Containers

If there is no class constraint on a protocol or protocol composition type, the existential container has to accommodate a value of arbitrary size and alignment. It does this using a **fixed-size buffer**, which is three pointers in size and pointer-aligned. This either directly contains the value, if its size and alignment are both less than or equal to the fixed-size buffer's, or contains a pointer to a side allocation owned by the existential container. The type of the contained value is identified by its *type metadata* record, and witness tables for all of the required protocol conformances are included. The layout is as if declared in the following C struct:

```
struct OpaqueExistentialContainer {
  void *fixedSizeBuffer[3];
  Metadata *type;
  WitnessTable *witnessTables[NUM_WITNESS_TABLES];
};
```

Class Existential Containers

If one or more of the protocols in a protocol or protocol composition type have a class constraint, then only class values can be stored in the existential container, and a more efficient representation is used. Class instances are always a single pointer in size, so a fixed-size buffer and potential side allocation is not needed, and class instances always have a reference to their own type metadata, so the separate metadata record is not needed. The layout is thus as if declared in the following C struct:

```
struct ClassExistentialContainer {
  HeapObject *value;
  WitnessTable *witnessTables[NUM_WITNESS_TABLES];
};
```

Note that if no witness tables are needed, such as for the "any class" type `protocol<class>` or an Objective-C protocol type, then the only element of the layout is the heap object pointer. This is ABI-compatible with `id` and `id <Protocol>` types in Objective-C.