

**orphan:** SUMMARY: Option sets should be structs of Books, with a protocol to provide bitwise-ish operations.

## Option Sets

Option sets in C and ObjC are often represented using enums with bit-pattern constants, as used in Cocoa's NS\_OPTIONS idiom. For example:

```
// ObjC
typedef NS_OPTIONS(NSUInteger, NSStringCompareOptions) {
    NSCaseInsensitiveSearch = 1,
    NSLiteralSearch = 2,
    NSBackwardsSearch = 4,
    NSAnchoredSearch = 8,
    NSNumericSearch = 64,
    NSDiacriticInsensitiveSearch = 128,
    NSWidthInsensitiveSearch = 256,
    NSForcedOrderingSearch = 512,
    NSRegularExpressionSearch = 1024
};
```

This approach doesn't map well to Swift's enums, which are intended to be strict enumerations of states, or "sum types" to use the type-theory-nerd term. An option set is more like a product type, and so more naturally map to a struct of booleans:

```
// Swift
struct NSStringCompareOptions {
    var CaseInsensitiveSearch,
        LiteralSearch,
        BackwardsSearch,
        AnchoredSearch,
        NumericSearch,
        DiacriticInsensitiveSearch,
        WidthInsensitiveSearch,
        ForcedOrderingSearch,
        RegularExpressionSearch : Bool = false
}
```

There are a few reasons this doesn't fly in C:

- Boolean fields in C structs waste a byte by default. Option set enums are compact.
- Bitfield ABI has historically been weird and unstable across C implementations. Option set enums have a very concrete binary representation.
- Prior to C99 it was difficult to use struct literals in expressions.
- It's useful to apply bitwise operations to option sets, which can't be applied to C structs.
- Bitmasks also provide a natural way to express common option subsets as constants, as in the `AllEdges` constants in the following example:

```
// ObjC
typedef NS_OPTIONS(unsigned long long, NSAlignmentOptions) {

    NSAlignMinXInward = 1ULL << 0,
    NSAlignMinYInward = 1ULL << 1,
    NSAlignMaxXInward = 1ULL << 2,
    NSAlignMaxYInward = 1ULL << 3,
    NSAlignWidthInward = 1ULL << 4,
    NSAlignHeightInward = 1ULL << 5,

    NSAlignMinXOutward = 1ULL << 8,
    NSAlignMinYOutward = 1ULL << 9,
    NSAlignMaxXOutward = 1ULL << 10,
    NSAlignMaxYOutward = 1ULL << 11,
    NSAlignWidthOutward = 1ULL << 12,
    NSAlignHeightOutward = 1ULL << 13,

    NSAlignMinXNearest = 1ULL << 16,
    NSAlignMinYNearest = 1ULL << 17,
    NSAlignMaxXNearest = 1ULL << 18,
    NSAlignMaxYNearest = 1ULL << 19,
    NSAlignWidthNearest = 1ULL << 20,
    NSAlignHeightNearest = 1ULL << 21,

    NSAlignRectFlipped = 1ULL << 63, // pass this if the rect is in a flipped coordinate system. This allows 0.5 to be treated

    // convenience combinations
    NSAlignAllEdgesInward = NSAlignMinXInward|NSAlignMaxXInward|NSAlignMinYInward|NSAlignMaxYInward,
    NSAlignAllEdgesOutward = NSAlignMinXOutward|NSAlignMaxXOutward|NSAlignMinYOutward|NSAlignMaxYOutward,
    NSAlignAllEdgesNearest = NSAlignMinXNearest|NSAlignMaxXNearest|NSAlignMinYNearest|NSAlignMaxYNearest,

};
```

However, we can address all of these issues in Swift. We should make the theoretically correct struct-of-Books representation also be the natural and optimal way to express option sets.

### The 'OptionSet' Protocol

One of the key features of option set enums is that, by using the standard C bitwise operations, they provide easy and expressive intersection, union, and negation of option sets. We can encapsulate these capabilities into a protocol:

```
// Swift
protocol OptionSet : Equatable {
    // Set intersection
    @infix func &(_:Self, _:Self) -> Self
    @infix func &=(_: inout Self, _:Self)

    // Set union
    @infix func |(_:Self, _:Self) -> Self
    @infix func |=(_: inout Self, _:Self)

    // Set xor
    @infix func ^(_:Self, _:Self) -> Self
    @infix func ^=(_: inout Self, _:Self)

    // Set negation
    @prefix func ~(_:Self) -> Self

    // Are any options set?
    func any() -> Bool

    // Are all options set?
    func all() -> Bool

    // Are no options set?
    func none() -> Bool
}
```

The compiler can derive a default conformance for a struct whose instance stored properties are all `Bool`:

```
// Swift
struct NSStringCompareOptions : OptionSet {
    var CaseInsensitiveSearch,
        LiteralSearch,
```

```

        BackwardsSearch,
        AnchoredSearch,
        NumericSearch,
        DiacriticInsensitiveSearch,
        WidthInsensitiveSearch,
        ForcedOrderingSearch,
        RegularExpressionSearch : Bool = false
    }

    var a = NSStringCompareOptions(CaseInsensitiveSearch: true,
                                   BackwardsSearch: true)
    var b = NSStringCompareOptions(WidthInsensitiveSearch: true,
                                   BackwardsSearch: true)

    var c = a & b
    (a & b).any() // => true
    c == NSStringCompareOptions(BackwardsSearch: true) // => true

```

### Optimal layout of Bool fields in structs

Boolean fields should take up a single bit inside aggregates, avoiding the need to mess with bitfields to get efficient layout. When used as inout arguments, boolean fields packed into bits can go through writeback buffers.

### Option Subsets

Option subsets can be expressed as static functions of the type. (Ideally these would be static constants, if we had those.) For example:

```

// Swift
struct NSAlignmentOptions : OptionSet {
    var AlignMinXInward,
        AlignMinYInward,
        AlignMaxXInward,
        AlignMaxYInward,
        AlignWidthInward,
        AlignHeightInward : Bool = false

    // convenience combinations
    static func NSAlignAllEdgesInward() {
        return NSAlignmentOptions(AlignMinXInward: true,
                                   AlignMaxXInward: true,
                                   AlignMinYInward: true,
                                   AlignMaxYInward: true)
    }
}

```

### Importing option sets from Cocoa

When importing an NS\_OPTIONS declaration from Cocoa, we import it as an OptionSet-conforming struct, with each single-bit member of the Cocoa enum mapping to a Bool field of the struct with a default value of false. Their IR-level layout places the fields at the correct bits to be ABI-compatible with the C type. Multiple-bit constants are imported as [option subsets](#), mapping to static functions.

*OPEN QUESTION:* What to do with bits that only appear as parts of option subsets, as in:

```

// ObjC
typedef NS_OPTIONS(unsigned, MyOptions) {
    Foo = 0x01,
    Bar = 0x03, // 0x02 | 0x01
    Bas = 0x05, // 0x04 | 0x01
};

```

### Areas for potential syntactic refinement

There are some things that are a bit awkward under this proposal which I think are worthy of some examination. I don't have great solutions to any of these issues off the top of my head.

#### Type and default value of option fields

It's a bit boilerplate-ish to have to spell out the : Bool = true for the set of fields:

```

// Swift
struct MyOptions : OptionSet {
    var Foo,
        Bar,
        Bas : Bool = false
}

```

(though by comparison with C, it's still a net win, since the bitshifted constants don't need to be manually spelled out and maintained. Is this a big deal?)

#### Construction of option sets

The implicit elementwise keyworded constructor for structs works naturally for option set structs, except that it requires a bulky and repetitive : true (or : false) after each keyword:

```

// Swift
var myOptions = MyOptions(Foo: true, Bar: true)

```

Some sort of shorthand for keyword: true/keyword: false would be nice and would be generally useful beyond option sets, though I don't have any awesome ideas of how that should look right now.

#### Nonuniformity of single options and option subsets

Treating individual options and [option subsets](#) differently disrupts some of the elegance of the bitmask idiom. As static functions, option subsets can't be combined freely in constructor calls like they can with | in C. As instance stored properties, individual options must be first constructed before bitwise operations can be applied to them.

```

// ObjC
typedef NS_OPTIONS(unsigned, MyOptions) {
    Foo = 0x01,
    Bar = 0x02,
    Bas = 0x04,

    Foobar = 0x03,
};

MyOptions x = Foobar | Bas;

// Swift, under this proposal
struct MyOptions : OptionSet {
    var Foo, Bar, Bas : Bool = false

    static func Foobar() -> MyOptions {
        return MyOptions(Foo: true, Bar: true)
    }
}

var x: MyOptions = .Foobar() | MyOptions(Bas: true)

```

This nonuniformity could potentially be addressed by introducing additional implicit decs, such as adding implicit static properties

corresponding to each individual option:

```
// Swift
struct MyOptions : OptionSet {
    // Stored properties of instances
    var Foo, Bar, Bas : Bool = false

    static func Foobar() -> MyOptions {
        return MyOptions(Foo: true, Bar: true)
    }

    // Implicitly-generated static properties?
    static func Foo() -> MyOptions { return MyOptions(Foo: true) }
    static func Bar() -> MyOptions { return MyOptions(Bar: true) }
    static func Bas() -> MyOptions { return MyOptions(Bas: true) }
}

var x: MyOptions = .Foobar() | .Bas()
```

This is getting outside of strict protocol conformance derivation, though.

### Lack of static properties

Static constant properties seem to me like a necessity to make option subsets really acceptable to declare and use. This would be a much nicer form of the above:

```
// Swift
struct MyOptions : OptionSet {
    // Stored properties of instances
    var Foo, Bar, Bas : Bool = false

    static val Foobar = MyOptions(Foo: true, Bar: true)

    // Implicitly-generated static properties
    static val Foo = MyOptions(Foo: true)
    static val Bar = MyOptions(Bar: true)
    static val Bas = MyOptions(Bas: true)
}

var x: MyOptions = .Foobar | .Bas
```