orphan:

Rationales for the Swift standard library designs

This document collects rationales for the Swift standard library. It is not meant to document all possible designs that we considered, but might describe some of those, when important to explain the design that was chosen.

Current designs

Some `NSString` APIs are mirrored on `String`

There was not enough time in Swift 1.0 to design a rich `String` API, so we reimplemented most of `NSString` APIs on `String` for parity. This brought the exact `NSString` semantics of those APIs, for example, treatment of Unicode or behavior in edge cases (for example, empty strings), which we might want to reconsider.

Radars: rdar://problem/19705854

`size_t` is unsigned, but it is imported as `Int`

Converging APIs to use `Int` as the default integer type allows users to write fewer explicit type conversions.

Importing `size_t` as a signed `Int` type would not be a problem for 64-bit platforms. The only concern is about 32-bit platforms, and only about operating on array-like data structures that span more than half of the address space. Even today, in 2015, there are enough 32-bit platforms that are still interesting, and x32 ABIs for 64-bit CPUs are also important. We agree that 32-bit platforms are important, but the usecase for an unsigned `size_t` on 32-bit platforms is pretty marginal, and for code that nevertheless needs to do that there is always the option of doing a bitcast to `UInt` or using C.

Type Conversions

The canonical way to convert from an instance *x* of type `T` to type `U` is `U(x)`, a precedent set by `Int(value: UInt32)`. Conversions that can fail should use failable initializers, e.g. `Int(text: String)`, yielding a `Int?`. When other forms provide added convenience, they may be provided as well. For example:

```
String.Index(s.utf16.startIndex.successor(), within: s) // canonical
s.utf16.startIndex.successor().samePosition(in: s)      // alternate
```

Converting initializers generally take one parameter. A converting initializer's first parameter should not have an argument label unless it indicates a lossy, non-typesafe, or non-standard conversion method, e.g. `Int(bitPattern: someUInt)`. When a converting initializer requires a parameter for context, it should not come first, and generally *should* use a keyword. For example, `String(33, radix: 2)`.

Rationale: First, type conversions are typical trouble spots, and we like the idea that people are explicit about the types to which they're converting. Secondly, avoiding method or property syntax provides a distinct context for code completion. Rather than appearing in completions offered after `.`, for example, the available conversions could show up whenever the user hit the "tab" key after an expression.

Protocols with restricted conformance rules

It is sometimes useful to define a public protocol that only a limited set of types can adopt. There is no language feature in Swift to disallow declaring conformances in third-party code: as long as the requirements are implemented and the protocol is accessible, the compiler allows the conformance.

The standard library adopts the following pattern: the protocol is declared as a regular public type, but it includes at least one requirement named using the underscore rule. That underscored API becomes private to the users according to the standard library convention, effectively preventing third-party code from declaring a conformance.

For example:

```
public protocol CVarArgType {
  var _cVarArgEncoding: [Word] { get }
}

// Public API that uses CVaListPointer, so CVarArgType has to be public, too.
public func withVaList<R>(
  _ args: [CVarArgType],
  @noescape invoke body: (CVaListPointer) -> R
) -> R
```

High-order functions on collections return `Array`s

We can't make `map()`, `filter()`, etc. all return `Self`:

- `map()` takes a function `(T) -> U` and therefore can't return Self literally. The required language feature for making `map()` return something like `Self` in generic code (higher-kinded types) doesn't exist in Swift. You can't write a method like `func map(_ f: (T) -> U) -> Self<U>` today.

- There are lots of sequences that don't have an appropriate form for the result. What happens when you filter the only element out of a `SequenceOfOne<T>`, which is defined to have exactly one element?
- A `map()` that returns `Self<U>` hews most closely to the signature required by Functor (mathematical purity of signature), but if you make map on `Set` or `Dictionary` return `Self`, it violates the semantic laws required by Functor, so it's a false purity. We'd rather preserve the semantics of functional `map()` than its signature.
- The behavior is surprising (and error-prone) in generic code:

```
func countFlattenedElements<
  S : SequenceType where S.Generator.Element == Set<Double>
>(_ sequence: S) -> Int {
  return sequence.map { $0.count }.reduce(0) { $0 + $1 }
}
```

The function behaves as expected when given an `[Set<Double>]`, but the results are wrong for `Set<Set<Double>>`. The `sequence.map()` operation would return a `Set<Int>`, and all non-unique counts would disappear.

- Even if we throw semantics under the bus, maintaining mathematical purity of signature prevents us from providing useful variants of these algorithms that are the same in spirit, like the `flatMap()` that selects the non-nil elements of the result sequence.

The *remove*()* method family on collections

Protocol extensions for `RangeReplaceableCollectionType` define `removeFirst(n: Int)` and `removeLast(n: Int)`. These functions remove exactly `n` elements; they don't clamp `n` to `count` or they could be masking bugs.

Since the standard library tries to preserve information, it also defines special overloads that return just one element, `removeFirst() -> Element` and `removeLast() -> Element`, that return the removed element. These overloads have a precondition that the collection is not empty. Another possible design would be that they don't have preconditions and return `Element?`. Doing so would make the overload set inconsistent: semantics of different overloads would be significantly different. It would be surprising that `myData.removeFirst()` and `myData.removeFirst(1)` are not equivalent.

Lazy functions that operate on sequences and collections

In many cases functions that operate on sequences can be implemented either lazily or eagerly without compromising performance. To decide between a lazy and an eager implementation, the standard library uses the following rule. When there is a choice, and not explicitly required by the API semantics, functions don't return lazy collection wrappers that refer to users' closures. The consequence is that all users' closures are `@noescape`, except in an explicitly lazy context.

Based on this rule, we conclude that `enumerate()`, `zip()` and `reverse()` return lazy wrappers, but `filter()` and `map()` don't. For the first three functions being lazy is the right default, since usually the result is immediately consumed by for-in, so we don't want to allocate memory for it.

Note that neither of the two `sorted()` methods (neither one that accepts a custom comparator closure, nor one that uses the `Comparable` conformance) can't be lazy, because the lazy version would be less efficient than the eager one.

A different design that was rejected is to preserve consistency with other strict functions by making these methods strict, but then client code needs to call an API with a different name, say `lazyEnumerate()` to opt into laziness. The problem is that the eager API, which would have a shorter and less obscure name, would be less efficient for the common case.

Possible future directions

This section describes some of the possible future designs that we have discussed. Some might get dismissed, others might become full proposals and get implemented.

Mixed-type fixed-point arithmetic

Radars: rdar://problem/18812545 rdar://problem/18812365

Standard library only defines arithmetic operators for LHS and RHS that have matching types. It might be useful to allow users to mix types.

There are multiple approaches:

- AIR model,
- overloads in the standard library for operations that are always safe and can't trap (e.g., comparisons),
- overloads in the standard library for all operations.

TODO: describe advantages

The arguments towards not doing any of these, at least in the short term:

- demand might be lower than we think: seems like users have converged towards using `Int` as the default integer type.
- mitigation: import good C APIs that use appropriate typedefs for unsigned integers (`size_t` for example) as `Int`.

Swift: Power operator

Radars: rdar://problem/17283778

It would be very useful to have a power operator in Swift. We want to make code look as close as possible to the domain notation, the two-dimensional formula in this case. In the two-dimensional representation exponentiation is represented by a change in formatting. With `pow()`, once you see the comma, you have to scan to the left and count parentheses to even understand that there is a `pow()` there.

The biggest concern is that adding an operator has a high barrier. Nevertheless, we agree `**` is the right way to spell it, if we were to have it. Also there was some agreement that if we did not put this operator in the core library (so that you won't get it by default), it would become much more compelling.

We will revisit the discussion when we have submodules for the standard library, in one form or the other.