Index Invalidation Rules in the Swift Standard Library

Points to consider

(1) Collections can be implemented as value types or a reference types.

(2) Copying an instance of a value type, or copying a reference has well-defined semantics built into the language and is not controllable by the user code.

    Consequence: value-typed collections in Swift have to use copy-on-write for data stored out-of-line in reference-typed buffers.

(3) We want to be able to pass/return a Collection along with its indices in a safe manner.

    In Swift, unlike C++, indices are not sufficient to access collection data; one needs an index and a collection. Thus, merely passing a collection by value to a function should not invalidate indices.

General principles

In C++, validity of an iterator is a property of the iterator itself, since iterators can be dereferenced to access collection elements.

In Swift, in order to access a collection element designated by an index, subscript operator is applied to the collection, `C[I]`. Thus, index is valid or not only in context of a certain collection instance at a certain point of program execution. A given index can be valid for zero, one or more than one collection instance at the same time.

An index that is valid for a certain collection designates an element of that collection or represents a one-past-end index.

Operations that access collection elements require valid indexes (this includes accessing using the subscript operator, slicing, swapping elements, removing elements etc.)

Using an invalid index to access elements of a collection leads to unspecified memory-safe behavior. (Possibilities include trapping, performing the operation on an arbitrary element of this or any other collection etc.) Concrete collection types can specify behavior; implementations are advised to perform a trap.

An arbitrary index instance is not valid for an arbitrary collection instance.

The following points apply to all collections, defined in the library or by the user:

(1) Indices obtained from a collection `C` via `C.startIndex`, `C.endIndex` and other collection-specific APIs returning indices, are valid for `C`.

(2) If an index `I` is valid for a collection `C`, a copy of `I` is valid for `C`.

(3) If an index `I` is valid for a collection `C`, indices obtained from `I` via `I.successor()`, `I.predecessor()`, and other index-specific APIs, are valid for `C`. FIXME: disallow startIndex.predecessor(), endIndex.successor()

(4) **Indices of collections and slices freely interoperate.**

If an index `I` is valid for a collection `C`, it is also valid for slices of `C`, provided that `I` was in the bounds that were passed to the slicing subscript.

If an index `I` is valid for a slice obtained from a collection `C`, it is also valid for `C` itself.

(5) If an index `I` is valid for a collection `C`, it is also valid for a copy of `C`.

(6) If an index `I` is valid for a collection `C`, it continues to be valid after a call to a non-mutating method on `C`.

(7) Calling a non-mutating method on a collection instance does not invalidate any indexes.

(8) Indices behave as if they are composites of offsets in the underlying data structure. For example:

- an index into a set backed by a hash table with open addressing is the number of the bucket where the element is stored;
- an index into a collection backed by a tree is a sequence of integers that describe the path from the root of the tree to the leaf node;
- an index into a lazy flatMap collection consists of a pair of indices, an index into the base collection that is being mapped, and the index into the result of mapping the element designated by the first index.

This rule does not imply that indices should be cheap to convert to actual integers. The offsets for consecutive elements could be non-consecutive (e.g., in a hash table with open addressing), or consist of multiple offsets so that the conversion to an integer is non-trivial (e.g., in a tree).

Note that this rule, like all other rules, is an "as if" rule. As long as the resulting semantics match what the rules dictate, the actual implementation can be anything.

Rationale and discussion:

- This rule is mostly motivated by its consequences, in particular, being able to mutate an element of a collection without changing the collection's structure, and, thus, without invalidating indices.
- Replacing a collection element has runtime complexity O(1) and is not considered a structural mutation. Therefore, there seems to be no reason for a collection model would need to invalidate indices from the implementation point of view.

- Iterating over a collection and performing mutations in place is a common pattern that Swift's collection library needs to support. If replacing individual collection elements would invalidate indices, many common algorithms (like sorting) wouldn't be implementable directly with indices; the code would need to maintain its own shadow indices, for example, plain integers, that are not invalidated by mutations.

Consequences:

- The setter of `MutableCollection.subscript(_: Index)` does not invalidate any indices. Indices are composites of offsets, so replacing the value does not change the shape of the data structure and preserves offsets.
- A value type mutable linked list cannot conform to `MutableCollectionType`. An index for a linked list has to be implemented as a pointer to the list node to provide O(1) element access. Mutating an element of a non-uniquely referenced linked list will create a copy of the nodes that comprise the list. Indices obtained before the copy was made would point to the old nodes and wouldn't be valid for the copy of the list.

  It is still valid to have a value type linked list conform to `CollectionType`, or to have a reference type mutable linked list conform to `MutableCollection`.

The following points apply to all collections by default, but specific collection implementations can be less strict:

(1) A call to a mutating method on a collection instance, except the setter of `MutableCollection.subscript(_: Index)`, invalidates all indices for that collection instance.

Consequences:

- Passing a collection as an `inout` argument invalidates all indexes for that collection instance, unless the function explicitly documents stronger guarantees. (The function can call mutating methods on an `inout` argument or completely replace it.)
  - `Swift.swap()` does not invalidate any indexes.

Additional guarantees for `Swift.Array`, `Swift.ContiguousArray`, `Swift.ArraySlice`

Valid array indexes can be created without using Array APIs. Array indexes are plain integers. Integers that are dynamically in the range `0..<A.count` are valid indexes for the array or slice `A`. It does not matter if an index was obtained from the collection instance, or derived from input or unrelated data.

Traps are guaranteed. Using an invalid index to designate elements of an array or an array slice is guaranteed to perform a trap.

Additional guarantees for `Swift.Dictionary`

Insertion into a Dictionary invalidates indexes only on a rehash. If a `Dictionary` has enough free buckets (guaranteed by calling an initializer or reserving space), then inserting elements does not invalidate indexes.

Note: unlike C++'s `std::unordered_map`, removing elements from a `Dictionary` invalidates indexes.