Objective-C Mapping for Structures

On this page:

Basic Objective-C Mapping for Structures

A Slice structure maps to an Objective-C class.

For each Slice data member, the generated Objective-C class has a corresponding property. For example, here is our `Employee` structure once more:

Slice

```
struct Employee {
    long number;
    string firstName;
    string lastName;
};
```

The Slice-to-Objective-C compiler generates the following definition for this structure:

Objective-C

```
@interface EXEmployee : NSObject <NSCopying>
{
    @private
        ICELong number;
        NSString *firstName;
        NSString *lastName;
}

@property(nonatomic, assign) ICELong number;
@property(nonatomic, retain) NSString *firstName;
@property(nonatomic, retain) NSString *lastName;

-(id) init:(ICELong)number firstName:(NSString *)firstName
                           lastName:(NSString *)lastName;
+(id) employee:(ICELong)number firstName:(NSString *)firstName
                               lastName:(NSString *)lastName;
+(id) employee;
// This class also overrides copyWithZone,
// hash, isequal, and dealloc.
@end
```

Mapping for Data Members in Objective-C

For each data member in the Slice definition, the Objective-C class contains a corresponding private instance variable of the same name, as well as a property definition that allows you to set and get the value of the corresponding instance variable. For example, given an instance of `EXEmployee`, you can write the following:

Objective-C

```
ICELong number;
EXemployee *e = ...;
[e setNumber:99];
number = [e number];

// Or, more concisely with dot notation:

e.number = 99;
number = e.number;
```

Properties that represent data members always use the `nonatomic` property attribute. This avoids the overhead of locking each data member during access. The second property attribute is `assign` for integral and floating-point types and `retain` for all other types (such as strings, structures, and so on.)

Creating and Initializing Structures in Objective-C

Structures provide the typical (inherited) `init` method:

Objective-C

```
EXEmployee *e = [[EXEmployee alloc] init];
// ...
[e release];
```

As usual, `init` initializes the instance variables of the structure with zero-filled memory. You can also declare default values in your Slice definition, in which case this `init` method initializes each data member with
its declared value.

In addition, a structure provides a second `init` method that accepts one parameter for each data member of the structure:

Objective-C

```
-(id) init:(ICELong)number firstName:(NSString *)firstName
                           lastName:(NSString *)lastName;
```

Note that the first parameter is always unlabeled; the second and subsequent parameters have a label that is the same as the name of the corresponding Slice data member. The additional `init` method allows you to instantiate a structure and initialize its data members in a single statement:

Objective-C

```
EXEmployee *e = [[EXEmployee alloc] init:99 firstName:@"Brad" lastName:@"Cox"];
// ...
[e release];
```

`init` applies the memory management policy of the corresponding properties, that is, it calls `retain` on the `firstName` and `lastName` arguments.

Each structure also provides two convenience constructors that mirror the `init` methods: a parameter-less convenience constructor and one that has a parameter for each Slice data member:

Objective-C

```
+(id) employee;
+(id) employee:(ICELong)number firstName:(NSString *)firstName
                               lastName:(NSString *)lastName;
```

The convenience constructors have the same name as the mapped Slice structure (without the module prefix). As usual, they allocate an instance, perform the same initialization actions as the corresponding `init` methods, and call `autorelease` on the return value:

Objective-C

```
EXEmployee *e = [EXEmployee employee:99 firstName:@"Brad" lastName:@"Cox"];

// No need to call [e release] here.
```

Copying Structures in Objective-C

Structures implement the `NSCopying` protocol. Structures are copied by assigning instance variables of value type and calling `retain` on each instance variable of non-value type. In other words, the copy is shallow:

Objective-C

```
EXEmployee *e = [EXEmployee employee:99 firstName:@"Brad" lastName:@"Cox"];
EXEmployee *e2 = [e copy];
NSAssert(e.number == e2.number);
NSAssert([e.firstName == e2.firstName]); // Same instance
// ...
[e2 release];
```

Note that, if you assign an `NSMutableString` to a structure member and use the structure as a dictionary key, you must not modify the string inside the structure without copying it because doing so will corrupt the dictionary.

Deallocating Structures in Objective-C

Each structure implements a `dealloc` method that calls `release` on each instance variable with a `retain` property attribute. This means that structures take care of the memory management of their contents: releasing a structure automatically releases all its instance variables.

Structure Comparison and Hashing in Objective-C

Structures implement `isEqual`, so you can compare them for equality. Two structures are equal if all their instance variables are equal. For value types, equality is determined by the == operator; for non-value types other than classes, equality is determined by the corresponding instance variable's `isEqual` method. Classes are compared by comparing their identity: two class members are equal if they both point at the same instance.

The `hash` method returns a hash value is that is computed from the hash value of all of the structure's instance variables.

See Also

- Structures
- Dictionaries
- Objective-C Mapping for Modules
- Objective-C Mapping for Identifiers
- Objective-C Mapping for Built-In Types
- Objective-C Mapping for Enumerations
- Objective-C Mapping for Sequences
- Objective-C Mapping for Dictionaries
- Objective-C Mapping for Constants
- Objective-C Mapping for Exceptions
- Objective-C Mapping for Interfaces

- Objective-C Mapping for Classes