

Bachelorarbeit Informatik
Sommersemester 2016
Prof. Dr. Schiedermeier
Prof. Dr. Teßmann

Speicherverwaltung in Swift: Low-Level-Analyse der ARC- und Copy-on-Write-Implementierungen

Xaver Lohmüller
2427688
Studiengang Informatik

Sandstraße 5
91126 Kammerstein/Haag
0151 25310407
xaver.lohmueller@me.com
Abgegeben am 11. August 2016

Abstract

Die vorliegende Bachelorarbeit untersucht Speicherlayout und -verwaltung der Programmiersprache Swift durch eine Quellcodeanalyse. Anlass ist unter anderem die Veröffentlichung des Quellcodes der Sprache im Dezember 2015.

Im besonderen Fokus stehen dabei die Implementierung des automatischen Referenzzählens (ARC) sowie die damit verbundene Problematik von Referenzzyklen und die Copy-on-Write-Semantik von Swift-eigenen Typen wie beispielsweise Arrays. Auch die verschiedenen Arten von Variablen (let und var) werden in ihrer Wirkung untersucht. Ferner werden Aspekte von Closures und parallel ablaufendem Code im Hinblick auf die Speicherverwaltung evaluiert.

Zur Untersuchung der Themen wird der offen vorliegende Quellcode der Sprache sowie durch den Compiler generierter Zwischencode verwendet. Weiterhin werden aktuelle Quellen aus der Literatur und der offiziellen Dokumentation zum Thema herangezogen.

Als Ergebnis der Arbeit sollen die Mechanismen hinter der Speicherverwaltung näher beleuchtet und detailliert dargelegt werden. Referenzzyklen vermieden werden. Weiterhin sollen Kriterien für die Implementierung von eigenen Copy-on-Write-Datentypen festgelegt werden.

Inhaltsverzeichnis

Abstract	2
Inhaltsverzeichnis	3
Einleitung	4
Verwendete Tools und Versionen	6
Automatic Reference Counting	7
Wert- und Referenztypen, Speicherverwaltung	8
Auslesen des Speichers	12
ARC im Quellcode	14
ARC-Analyse zur Laufzeit	16
Implementierung der Referenzzähler-Modifikatoren	19
Analyse der Deallokation	21
Phasenweise Deallokation von Objekten	23
Wertsemantik mit Referenztypen	26
Werttypen mit Referenztyp-Implementierung	27
Copy-On-Write	29
Ein Beispiel: Binary Tree mit Wertsemantik	30
Let und Var	36
Einfluss des Variablentyps auf ARC	37
Closures	40
Capture-Semantik	41
Speicherlayout von Closures	45
Speicherlecks bei Closures, @noescape	46
Fazit und Ausblick	50
Anhang	51
Prüfungsrechtliche Erklärung	52
Erklärung zur Veröffentlichung der nachstehend bezeichneten Abschlussarbeit	53
Legende	54
Verzeichnisse	55
Quellenverzeichnis	58
Code	60

Einleitung

Generell gibt es zwei Möglichkeiten, Daten in einem Programm strukturiert zu speichern: Wert- und Referenztypen. Bei Werttypen werden Daten direkt in einer Variable abgelegt, bei Referenztypen besteht die Variable aus einem Zeiger, der auf die Daten zeigt. In Swift sind Werttypen durch Strukturen und Enums repräsentiert während Klassen und Closures Referenztypen sind.

Referenztypen können von vielen Stellen aus benutzt werden, ohne dass der Speicher kopiert werden muss: Instanzen von Klassen können mehrere "Besitzer" haben, indem mehrere Variablen das selbe Objekt referenzieren. Bei Werttypen ist das nicht möglich: Eine Variable speichert eine Kopie eines Wertes. Aufgrund dieser Tatsache ist auch die Speicherverwaltung verschieden: Bei Werttypen kann der Speicher sofort freigegeben werden, wenn der Besitzer die Variable nicht mehr benötigt. Bei Referenztypen hingegen kann es sein, dass der Speicher noch von einem anderen Besitzer benötigt wird: Eine einfache Aussage über die Möglichkeit der Speicherfreigabe ist nicht so leicht möglich.

Programmiersprachen gehen unterschiedlich mit diesem Problem um: Teilweise wird die Heap-Speicherverwaltung komplett dem Nutzer überlassen (zB in C), manche bieten eine Hilfestellung wie MRC (Manual Reference Counting, Objective-C früher). Bei moderneren Sprachen haben sich vollautomatische Speicherverwaltung-Systeme wie (Tracing) Garbage Collection und Automatic Reference Counting (ARC) durchgesetzt.

In Swift kommt ARC zum Einsatz: Objekte werden mit Referenzzählern versehen und werden automatisch freigegeben, wenn sie nicht mehr benötigt werden.

In dieser Bachelorarbeit wird untersucht, wie ARC genau implementiert ist und funktioniert. Daraufhin wird eine weitere Anwendungsmöglichkeit der Referenzzähler, nämlich Copy-On-Write, untersucht. Außerdem werden die verschiedenen Arten von Variablen (let und var) in ihrer Wirkung auf ARC genauer betrachtet. Abschließend werden Closures und deren Capture-Semantik unter die Lupe genommen: Da es sich dabei wie bei Klassen um Referenztypen handelt werden diese durch ARC verwaltet.

Verwendete Tools und Versionen

Sämtlicher Code in dieser Bachelorarbeit wurde mit Xcode 7 erstellt. Im Laufe der Arbeit kamen die Versionen 7.2.1, 7.3 und 7.3.1 zum Einsatz. Sämtlicher Code im Anhang wurde mit der neuesten Version getestet.

Die Programmiersprache Swift selbst wird in der Version 2.2 verwendet. An einigen Stellen wird ein Ausblick auf Swift 3 gegeben und, wenn vorhanden, der Feature-Vorschlag (SE) genannt. Da die Diskussion über die Features von Swift 3 erst am 27. Juli 2016 beendet wird und die Syntax sich danach noch geringfügig ändern kann, liegt der Code nur für die stabile Version 2.2 vor. Nach dem Release von Swift 3 sollte es möglich sein, den Code mit Hilfe von Migrationstools zu aktualisieren.

Sämtliche Codebeispiele, die in der Arbeit diskutiert werden, sind unabhängig von den Tools und Betriebssystemen von Apple und beziehen sich nur auf Sprachfeatures. Im Zuge der Arbeit wird eine iOS-App vorgestellt, welche natürlich diese Unabhängigkeit nicht genießt. Die App dient nur zu Visualisierungszwecken und ist nicht Teil der Bachelorarbeit. Es ist also möglich, die Arbeit ohne Wissen über die Plattformen von Apple nachzuvollziehen.

Zitate aus dem Quellcode von Swift entsprechen dem Stand vom 23. Juli 2016¹.

Die Arbeit in digitaler Form befindet sich neben sämtlichem Quellcode auf der beigelegten SD-Speicherkarte. Die Inhalte dieser Karte sind außerdem auf Github² verfügbar.

¹ Master-Branch, Commit "a9147c215ade90e4dd76b14d2b6ac020f0861afb"

² <https://github.com/xaverlohmueller/Bachelorarbeit>

Automatic Reference Counting

Wert- und Referenztypen, Speicherverwaltung

Werttypen zeichnen sich durch ihren Wert aus: unveränderlich und immer gleich. Wenn eine Variable mit einem Werttyp verändert wird, ändert sich nicht der Wert: Er wird durch einen anderen Wert ausgetauscht.

Referenztypen hingegen zeichnen sich durch ihre Identität aus: Über die Lebensdauer einer Referenz kann deren Inhalt sich verändern, es ist aber dennoch das selbe Objekt.

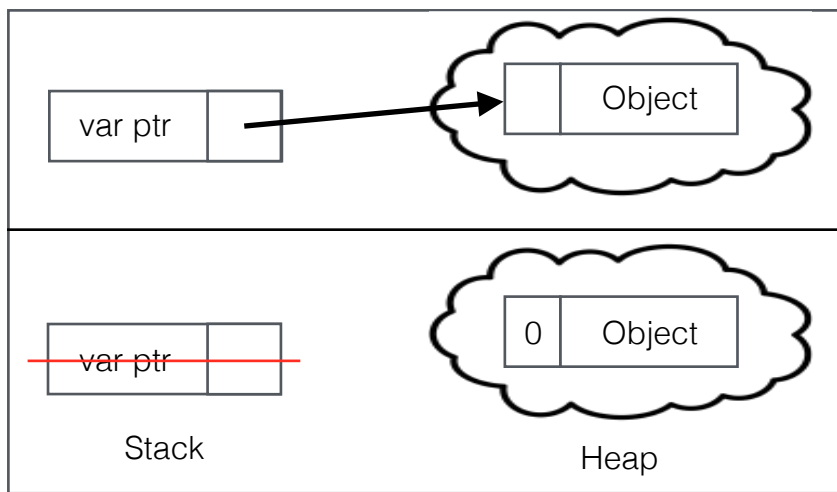
Weiterhin sind die Konzepte von Gleichheit unterschiedlich: Variablen von Werttypen sind dann gleich, wenn ihre Werte übereinstimmen. Bei Referenztypen spricht man von Gleichheit, wenn zwei Variablen auf das selbe Objekt zeigen, ihr Objekt also identisch ist.

Speicher ist eine begrenzte Ressource in Computern. Jede Variable, die im Programm angelegt wird, verbraucht während ihrer Lebenszeit ein Stück von diesem Speicher. Bei Variablen, die auf dem Stack (generell Werttypen) gespeichert werden sind, wird der Speicher beim Abbau des Stacks automatisch freigegeben. Im Gegensatz dazu stehen Variablen, deren Speicher im Heap liegt (generell Referenztypen): Der Speicher kann von mehreren Stellen aus referenziert sein und kann dadurch nicht sofort freigegeben werden, wenn eine referenzierende Variable entfernt wird.

Über die Jahre haben Programmiersprachen verschiedene Lösungen für diese Problematik angeboten. Im wesentlichen kann der Entwickler des Programms den Speicher selbst verwalten oder die Programmiersprache bietet ein System zur automatisierten Speicherverwaltung an. Dabei wiederum unterscheidet man zwischen Tracing Garbage Collection und Automatic Reference Counting.

Verfahren	Beispiele
Manuelle Speicherverwaltung	C, C++
Tracing Garbage Collection (GC)	Java, JavaScript, C#
Automatic Reference Counting (ARC)	Swift, Objective-C, Python, C++11/14

Die automatischen Verfahren geben Speicherbereiche im Heap frei, sobald diese nicht mehr durch Referenzen aus dem Programm erreicht werden können. Ein Unterschied zwischen den genannten Verfahren ist, wann der Speicher freigegeben wird: Bei der Tracing Garbage Collection wird das Programm kurzzeitig angehalten und die Speicherverwaltung verfolgt alle Referenzen in den Heap, um festzustellen, welche Objekte nicht mehr erreicht werden können, und gibt diese frei. Automatic Reference Counting hingegen merkt sich jede Referenz auf ein Objekt, und gibt den Speicher direkt frei, wenn die Anzahl der Referenzen auf ein Objekt auf Null sinkt:



Ein Heap-Objekt wird durch eine Variable referenziert. Der Referenzzähler steht auf 1.

Die Variable wird gelöscht, der Zähler sinkt auf 0. Das Objekt wird freigegeben.

Bild 1: ARC: Objekte werden freigegeben, sobald der Referenzzähler auf 0 fällt

Ein wesentliches Problem bei ARC sind Speicherzyklen: Wenn sich beispielsweise zwei Objekte gegenseitig referenzieren können deren Referenzzähler niemals auf Null absinken:

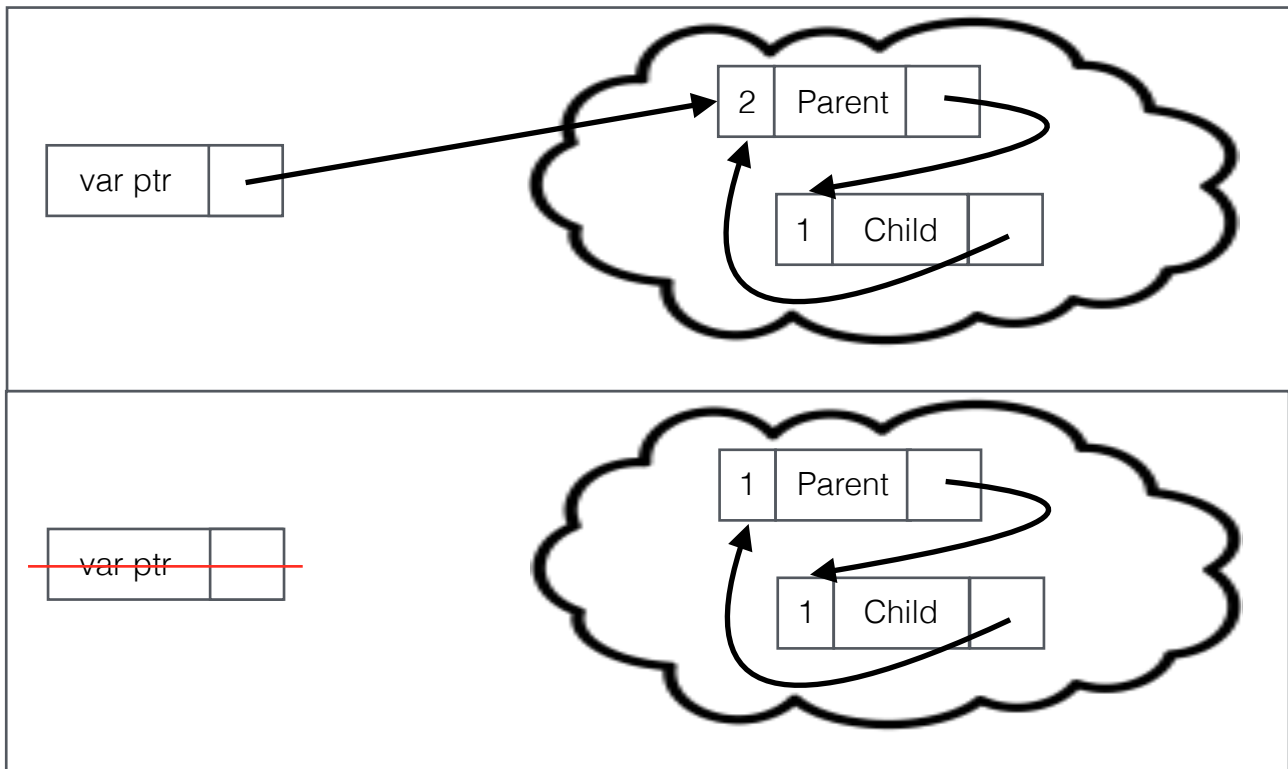


Bild 2: Obwohl die Variable gelöscht wurde, können die Objekte nicht freigegeben werden

Die Objekte können weder erreicht noch freigegeben werden und belegen so für die Laufzeit des Programms dauerhaft den Speicher. Man spricht dabei auch von einem Speicherleck.

Bei GC kommt dieses Problem nicht vor: Während einer Aufräumphase würden die sich gegenseitig referenzierenden Objekte nicht aus dem Heap erreicht werden und können somit freigegeben werden. Python verwendet deshalb neben ARC auch einen GC, um Zyklen zu beheben^{Q1}.

In Swift und C++11 gibt es keine Tracing Garbage Collection. Solche Zyklen sind somit möglich und müssen durch den Programmierer vermieden werden. Das passende Werkzeug sind dabei Referenzen, die zwar auf ein Objekt zeigen, den Referenzzähler aber nicht erhöhen. In Swift gibt es zwei Arten dieser besonderen Referenzen: **weak** und **unowned**. Das Wort 'unowned' umschreibt die Art dieser Referenz am treffendsten: Die Variable kennt zwar das Heap-Objekt, besitzt es aber nicht. Semantisch gesehen läuft die Speicherverwaltung ab, ohne solche Referenzen zu berücksichtigen.

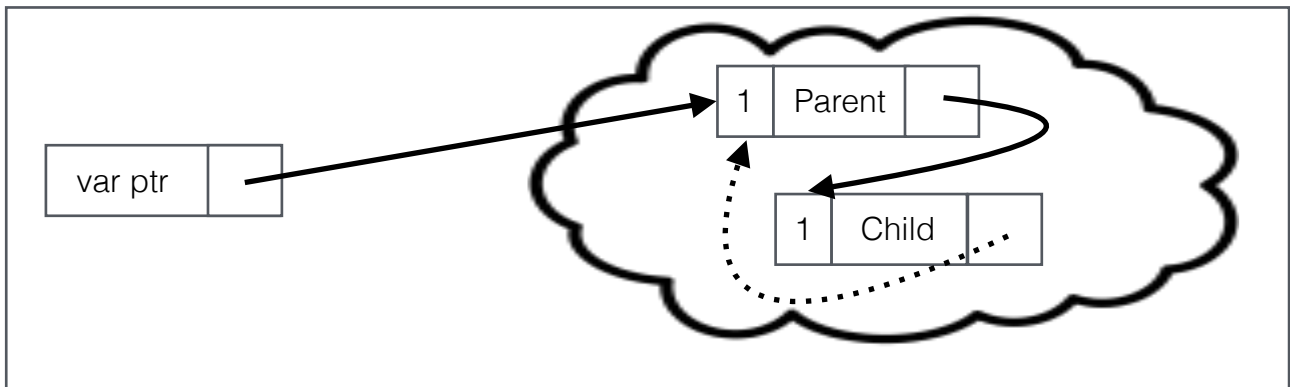


Bild 3: Parent-Child-Beziehung mit einem Weak Pointer: *Child* kennt zwar *Parent*, erhöht aber dessen Referenzzähler nicht. Wenn die Variable *ptr* gelöscht wird, kann *Parent* deallokiert werden, da der Zähler auf 0 sinkt.

Da der Speicherbereich einer weak-deklarierten Variable freigegeben werden kann, bevor die Variable gelöscht wird, kann es vorkommen, dass beim Dereferenzieren der Speicher nicht mehr gültig ist. Aus diesem Grund sind weak-Variablen in Swift immer vom Typ **Optional** und müssen vor der Verwendung entpackt werden, um Nullpointer oder ungültige Speicherzugriffe zu vermeiden.

Dies trifft auf unowned-deklarierte Variablen nicht zu. Der Speicherbereich kann zwar dennoch ungültig sein, die Prüfung bleibt aber dem Programmierer überlassen: Im Falle der Parent-Child-Beziehung könnte man den Zeiger von Child zu Parent als unowned implementieren, wenn man sicherstellen kann, dass Child niemals durch ein anderes Objekt referenziert wird. Dann wäre während der Lebenszeit von Child die Referenz auf Parent immer gültig, und eine Überprüfung im Stil von Optional bei jedem Zugriff wäre eine Verschwendung.

Bei unowned gibt es noch einmal zwei Varianten: **unowned** und **unowned(unsafe)**. Bei unowned wird bei versuchtem Zugriff auf einen ungültigen Speicherbereich ein Runtime Error auftreten, was bedeutet, dass das Programm abstürzt. Bei unowned(unsafe) wird der Speicherbereich nicht auf Gültigkeit überprüft, sondern ein Zugriff einfach zugelassen. Dabei kann wie in C/C++ undefiniertes Verhalten auftreten^{Q2}.

Reguläre Referenzen, die den Referenzzähler beeinflussen, heißen in Swift **strong**. Im Folgenden sind mit dem Wort 'Referenz' ohne weitere Erläuterung immer strong-Referenzen gemeint.

Im folgenden Codebeispiel wird eine Klasse definiert, die einen einfachen Wert speichert und bei Entfernen aus dem Heap eine Nachricht in stdout schreibt. Daraufhin wird eine Instanz dieser Klasse angelegt und ein unowned-deklarerter Pointer zur Instanz erstellt. Als nächste Instruktion wird der ursprüngliche Pointer gelöscht, wodurch sich die Anzahl der Referenzen auf Null verringert. Der Speicher kann damit freigegeben werden. Am Ende wird noch versucht, die unowned-Referenz zu dereferenzieren:

<pre>class SomeClass { var someValue: Int = 45054 deinit { print("SomeClass deinit") } } var anInstance: SomeClass? = SomeClass() unowned let unownedPointer = anInstance! anInstance = nil print("Trying to access pointer") print(unownedPointer.someValue)</pre>	<pre>SomeClass deinit Trying to access pointer (lldb)</pre>
--	---

Thread 1: EXC_BAD_ACCESS (code=2, address=0x7fff5fbff6a0)

Code 1: ARC in Swift in Aktion: Code und Konsolenausgabe

Wie man sieht, findet die Freigabe des Speichers bei ARC unmittelbar nach dem Löschen der letzten Referenz statt, ohne dass das Programm angehalten werden muss (wie bei GC). Bereits bei der nächsten Instruktion tritt beim Zugriffsversuch ein Laufzeitfehler auf.

Im nächsten Beispiel ist die zweite Referenz mit **unowned(unsafe)** statt **unowned** deklariert. Der Zugriff findet ohne Laufzeitfehler statt (und liefert in diesem Fall auch den korrekten Wert, da der Speicherbereich noch nicht neu vergeben wurde):

<pre>class SomeClass { var someValue: Int = 45054 deinit { print("SomeClass deinit") } } var anInstance: SomeClass? = SomeClass() unowned(unsafe) let unownedPointer = anInstance! anInstance = nil print("Trying to access pointer") print(unownedPointer.someValue)</pre>	<pre>SomeClass deinit Trying to access pointer 45054 Program ended with exit code: 0</pre>
--	--

Code 2: ARC in Swift in Aktion: Code und Konsolenausgabe.

Auslesen des Speichers

Um der Implementierung von ARC auf den Grund zu gehen, wird eine Methode benötigt, den belegten Speicher - besonders von Referenztypen - auszulesen. Zu diesem Zweck eignen sich das Playground-Feature von Xcode:

Zuerst wird die Größe einer Klasseninstanz bestimmt:

<pre>class SomeClass { var someValue = 0xf0f0f0f0 var anotherValue = 0x12345 } let anInstance = SomeClass() let size = sizeofValue(anInstance)</pre>	<div>SomeClass</div> <div>8</div>
--	-----------------------------------

Die Größe von 8 Byte spricht für einen Pointer (welche in Swift 8 Byte groß sind):

<pre>let pointer = unsafeBitCast(anInstance, UnsafePointer<UInt64>.self)</pre>	"UnsafePointer(0x7FACEB200A80)"
--	---------------------------------

Um den Pointer aus der Klasse zu extrahieren kann die Funktion ***unsafeBitCast*** verwendet werden, welche das Bitmuster des Wertes einer Variable in eine andere Variable beliebigen Typs schreibt (falls die Länge übereinstimmt). In diesem Fall wird das Bitmuster des Zeigers in einen UnsafePointer<Memory> kopiert³. Dabei handelt es sich um eine generische Struktur, die auf ein Objekt vom Typ ***Memory*** im Speicher zeigt. Ein UnsafePointer bietet keine automatische Speicherverwaltung, was für Analysezwecke gut ist, da es den Referenzzähler eben nicht verändert.

In unserem Fall zeigt pointer auf eine vorzeichenlose 64-Bit-Zahl. Der Speicherbereich eines UnsafePointer kann durch das ***memory***-Property ausgelesen werden.³

<pre>pointer.memory</pre>	"0x00000000112FB0038"
---------------------------	-----------------------

Durch die Methoden ***successor*** und ***predecessor*** erhält man die nachfolgenden beziehungsweise vorhergehenden Speicherblöcke:

<pre>pointer.successor().memory pointer.successor().successor().memory</pre>	<div>"0x00000000200000004"</div> <div>"0x00000000F0F0F0F0"</div>
--	--

³ Die hier ausgewerteten Zahlen werden durch die im Anhang beschriebene hex()-Funktion ausgegeben

Alternativ kann man Speicherblöcke durch ein Subscript auslesen:

<code>pointer[0]</code>	"0x0000000112FB0038"
<code>pointer[1]</code>	"0x0000000200000004"
<code>pointer[2]</code>	"0x00000000F0F0F0F0"
<code>pointer[3]</code>	"0x0000000000012345"

Aufgrund der ausgelesenen Werte kann man eine grundlegende Struktur der Objekte erkennen:

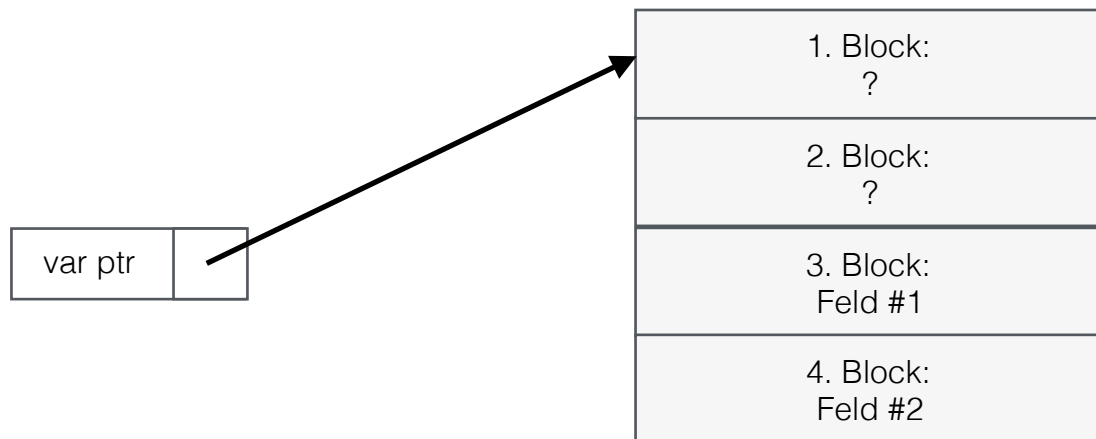


Bild 4: Struktur von Heapobjekten in Swift

Der 3. und 4. 64-Bit-Block entspricht offensichtlich den Werten in den Feldern der Instanz. Nachfolgende Blöcke ändern ihren Wert bei jedem Kompilieren woraus gefolgert werden kann, dass das Objekt hier zu Ende ist, da auch manchmal der Wert 0 auftaucht und es sich somit nicht um einen weiteren Pointer handeln kann.

ARC im Quellcode

Um die Inhalte des 1. und 2. Block aufzuschlüsseln lohnt sich ein Blick in den Quellcode. Klassen in Swift sind Referenztypen und werden auf dem Heap abgelegt. Jedes Objekt hat einen Header-Block, der am Anfang seines Speicherbereichs auf dem Heap liegt:

HeapObject.h

```
/// The Swift heap-object header.
struct HeapObject {
    /// This is always a valid pointer to a metadata object.
    HeapMetadata const *metadata;

    StrongRefCount refCount;
    WeakRefCount weakRefCount;

    ... // Constructors
};
```

Swift Code 1: Deklaration der Referenzzähler

Der erste Block ist ein Pointer auf ein Metadaten-Objekt, das Informationen über die Klasse enthält, beispielsweise Name, Anzahl der Felder, Anzahl der generischen Parameter, Oberklassen, Kompatibilität zu Objective-C, und viele weitere. Der zweite Block besteht aus den Referenzzählern für strong- und weak-Referenzen. Es handelt sich jeweils um eine Struktur, die eine einzige 32-Bit-Zahl beinhaltet:

RefCount.h

```
class StrongRefCount {
    uint32_t refCount;

    enum : uint32_t {
        RC_PINNED_FLAG = 0x1,
        RC_DEALLOCATING_FLAG = 0x2,

        RC_FLAGS_COUNT = 2,
        RC_FLAGS_MASK = 3,
        RC_COUNT_MASK = ~RC_FLAGS_MASK,

        RC_ONE = RC_FLAGS_MASK + 1
    };
    ... // Methods for in/decrementing
};

class WeakRefCount {
    uint32_t refCount;

    enum : uint32_t {
        RC_UNUSED_FLAG = 1,

        RC_FLAGS_COUNT = 1,
        RC_FLAGS_MASK = 1,
        RC_COUNT_MASK = ~RC_FLAGS_MASK,

        RC_ONE = RC_FLAGS_MASK + 1
    };
    ... // Methods for in/decrementing
};
```

Swift Code 2: Bedeutung der Bits in den Referenzzählern

Der zweite Block einer Klasse besteht also aus den strong- und weak-Referenzzählern, die jeweils noch um 2 Bits (im Falle von weak 1 Bit) verschoben sind, um Platz für Flags zu bieten. Die Zahl "0x00000000200000004" im zweiten Block bedeutet:

000000000000000010	000000000000000100
--------------------	--------------------

Weak References = 1

Strong References = 1

RC_UNUSED_FLAG = false

RC_DEALLOCATING_FLAG = false

RC_PINNED_FLAG = false

Code 3: Erklärung des gezeigten Bitmusters als Referenzzähler

ARC-Analyse zur Laufzeit

Zur genaueren Analyse des Verhaltens der Referenzzähler wurde eine App entwickelt, welche im wesentlichen aus einer Klasse zum Anlegen und Löschen verschiedener Referenzen besteht.

Diese Klasse ist der ***PointerController***⁴:

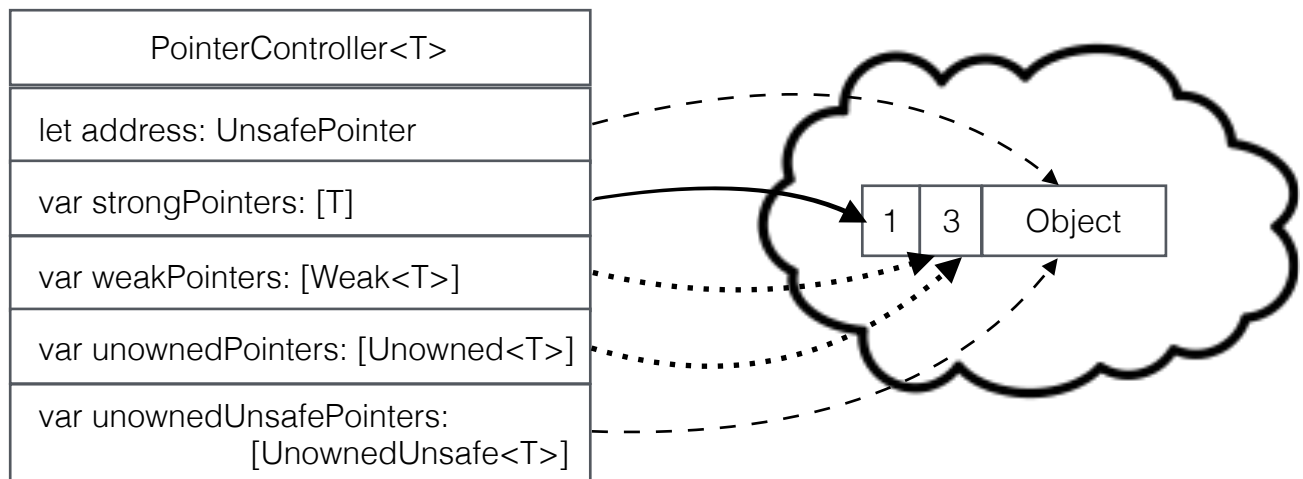


Bild 5: Der PointerController verwaltet unterschiedliche Zeiger auf ein Objekt

Die Klasse verwaltet einen `UnsafePointer` auf einen Speicherbereich (welcher den Referenzzähler des verwalteten Objektes nicht beeinflusst). Man kann die verschiedenen Flags sowie die aktuellen Referenzzähler auslesen und Zeiger aller Art hinzufügen oder entfernen, welche in entsprechenden Arrays gespeichert werden.

Die Klasse ist generisch im beobachteten Objekt. Damit sowohl Referenztypen als auch Typen mit Wertsemantik aber Implementierung mit Referenztypen untersucht werden können, sind einige Funktionen auf echte Referenztypen beschränkt: Zum Beispiel ist es nicht möglich, `weak-Pointer`, die auf ein Array oder einen String zeigen, zu erstellen.

Eine Besonderheit sind die Arrays von `weak`-, `unowned`- und `unowned(unsafe)`-Referenzen: Da Arrays in Swift ihre enthaltenen Objekte immer mit einem `strong Pointer` referenzieren, sind für andere Zeigerarten Hilfsstrukturen nötig. Ein Beispiel:

```
struct Weak<T: AnyObject> {  
    weak var value : T?  
    init(_ value: T) {  
        self.value = value  
    }  
}
```

In dieser Struktur wird ein `weak-Pointer` gespeichert. Im `PointerController` gibt es ein Array von `Weak`-Strukturen, um die Konvertierung in `strong-Pointer` zu verhindern.

In der App gibt es eine minimale UI, welche die

⁴ Genaue Implementierung siehe Anhang

PointerController-Klasse bedient. Mithilfe der App kann man interaktiv Zeiger aller Art hinzufügen oder entfernen und die Reaktionen des Speicherblocks, der die Referenzzähler beinhaltet, beobachten. Durch die oberen + und - Tasten kann man neue Zeiger zu den Arrays hinzufügen oder entfernen. Unten wird der Inhalt des Speicherblock angezeigt. In der Mitte gibt es Knöpfe zum Zurücksetzen der App sowie zum erneuten Auslesen des Speicherinhalts.

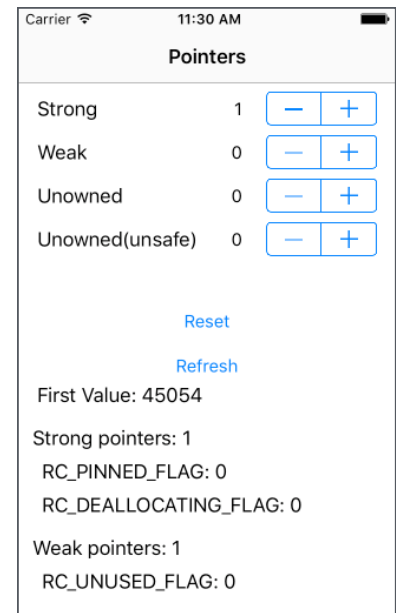
Die App startet mit einem einzelnen strong Pointer im entsprechenden Array. Beachtenswert ist unter anderem, dass die Anzahl der weak Pointers jetzt bereits auch 1 beträgt. Die Flags sind alle nicht gesetzt.

Generell verhalten sich die Referenzzähler wie erwartet: Zusätzliche strong Pointer erhöhen den die Anzahl der strong Pointers im Speicherblock, während weak und unowned Pointer die Anzahl der weak Pointers erhöhen.

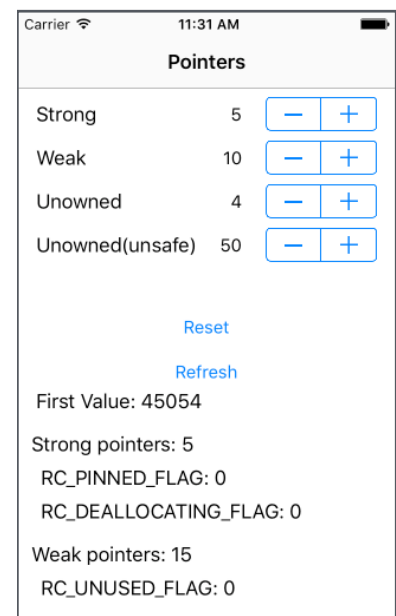
Die Anzahl der unowned(unsafe) Pointer beeinflusst keinen Referenzzähler. Im Prinzip sind solche Referenzen eine andere Schreibweise für die UnsafePointer-Klasse: Beide haben einen einfachen Zeiger auf einen Speicherbereich, kein Memory-Management und undefiniertes Verhalten bei Zugriffen auf ungültige Speicherbereiche.

Interessant ist das Verhalten der Referenzzähler, wenn die Anzahl der strong Pointers auf 0 fällt: Der zusätzliche Zeiger, der am Anfang durch den strong Pointer hinzugefügt wurde, ist nicht mehr da. Wenn man in der verwalteten Klasse einen deinit()-Block definiert hat, wird dieser ausgeführt. Außerdem ist jetzt das RC_DEALLOCATING_FLAG gesetzt.

Trotzdem erhält man auch nach mehrfachem Auslesen die korrekte Anzahl an weak Pointers zurück. Über einen unowned(unsafe)-Pointer können die Werte der Felder der verwalteten Klasse ausgelesen werden, welche korrekt geliefert werden. Der Speicherbereich der Klasse im Heap ist also noch vorhanden und mit korrekten Daten gefüllt.



App 1: Initialzustand: ein einzelner starker Pointer ist gesetzt



App 2: Einige Pointer sind gesetzt

Erst wenn sich die kombinierte Anzahl von weak und unowned Pointers auf 0 verringert wird der Speicherbereich deallokiert. Beim Auslesen des Speichers erhält man jetzt zufällige Werte. Die Anzahl der unowned(unsafe) Pointer hat auf das tatsächliche Freigeben des Speichers keinen Einfluss. Beim Zugriff auf die Werte der Felder über diese Zeiger erhält man wieder Zufallswerte.

Neben dem Freigeben der Zeiger gibt es noch eine weitere Möglichkeit, das Deallokieren des Speichers Auszulösen: Zugriffsversuche durch weak Pointer, wenn das `RC_DEALLOCATING_FLAG` gesetzt ist: Die Zeiger geben alle wie erwartet den Wert nil zurück. Sobald alle über alle weak-Pointer ein Zugriffsversuch erfolgt ist, wird der Speicher ebenfalls freigegeben. Die Referenzen selbst "merken" sich, dass ihr referenziertes Objekt deallokiert werden soll, und bei Variablenzugriff nil zurückgegeben werden muss. Das referenzierte Objekt selbst wird dafür nicht benötigt und kann freigegeben werden. Dieses Verhalten bezeichnet man als "zeroing"^{Q4}.

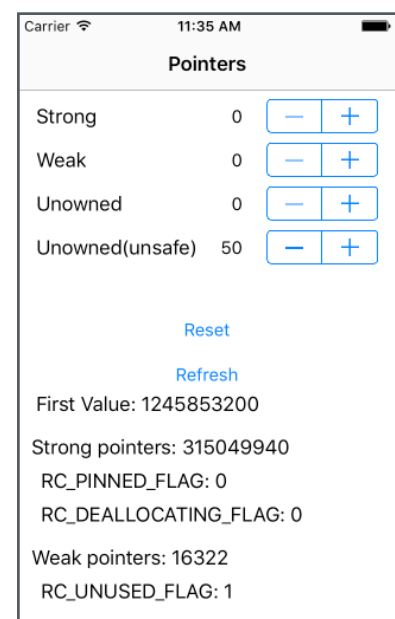
Offensichtlich läuft die Dekonstruktion von Objekten in zwei Phasen ab: Wenn keine strong Pointers mehr auf ein Objekt zeigen, wird in einer ersten Phase das Objekt als ungültig markiert (`RC_DEALLOCATING_FLAG`). Der Speicher des Objekts selbst wird aber tatsächlich erst deallokiert, wenn keine weak oder unowned Pointer mehr auf das Objekt zeigen, oder alle weak Pointer über das ungültige Objekt informiert sind.

Das kann ein Problem darstellen, wenn das Objekt einen großen Speicherbereich verwaltet, und dieser durch eine schwache Referenz am Leben erhalten wird. Bei den typischerweise speicherintensiven Typen wie Arrays, Strings und Dictionaries ist dies in Swift nicht möglich, da diese als Struct implementiert sind und daher keine schwachen Referenzen zulassen.

Um das Verhalten der Referenzzähler und der Deallokation nachvollziehen zu können lohnt sich ein Blick in den Quellcode:



App 3: Die Anzahl der strong Pointers fällt auf 0



App 4: Die Anzahl der weak Pointers fällt auf 0. Der Speicher wurde freigegeben

Implementierung der Referenzzähler-Modifikatoren

Swift bietet Funktionen zum Auslesen und Verändern der Referenzzähler eines Heapobjekts. Diese Funktionalität ist ebenfalls in RefCount.h in den Klassen StrongRefCount und WeakRefCount implementiert. Die Funktionen zum Auslesen und Inkrementieren sind vergleichsweise einfach:

RefCount.h

```
// Increment the reference count.
void increment() {
    __atomic_fetch_add(&refCount, RC_ONE, __ATOMIC_RELAXED);
}

// Return the reference count.
// During deallocation the reference count is undefined.
uint32_t getCount() const {
    return __atomic_load_n(&refCount, __ATOMIC_RELAXED) >> RC_FLAGS_COUNT;
}

// Return whether the reference count is exactly 1.
// During deallocation the reference count is undefined.
bool isUniquelyReferenced() const {
    return getCount() == 1;
}
```

Swift Code 3: Atomare Operatoren zur Modifikation von Referenzzählern

Dabei werden lediglich die entsprechenden Werte im RefCount-Struct modifiziert (beziehungsweise ausgelesen) und mit entsprechenden Offsets versehen, um die Flags außer Acht zu lassen. Diese Operationen sind atomar, um Race Conditions zu vermeiden. Weiterhin gibt es eine Funktion um zu überprüfen, ob das Objekt nur durch einen einzigen Pointer referenziert wird.

Neben der einfachen increment-Funktion gibt es noch weitere, welche den Referenzzähler nicht atomar erhöhen oder um andere Werte als 1 erhöhen können.

Deutlich aufwändiger ist die Funktion zum Dekrementieren des Referenzzählers, da sie entscheiden muss, ob das Objekt deallokiert werden soll. Aus diesem Grund gibt die Funktion auch einen boolschen Wert zurück, der aussagt, ob der Aufrufer der Funktion das Objekt deallokieren soll.⁵

Bei der Dekrementierung gibt es analog zum Inkrementieren Varianten mit nichtatomarer Ausführung und anderen Werten als 1.

⁵ Anmerkung: Code, der mit Object Pinning verbunden ist, wurde gekürzt

RefCount.h

```
// Class StrongRefCount
bool doDecrementShouldDeallocate() {
    uint32_t newval = __atomic_sub_fetch(&refCount, RC_ONE, __ATOMIC_RELEASE);

    // If we didn't drop the reference count to zero, or if the
    // deallocating flag is already set, we're done; don't start
    // deallocation.
    if ((newval & (RC_COUNT_MASK | RC_DEALLOCATING_FLAG)) != 0) {
        // Refcount is not zero. We definitely do not need to deallocate.
        return false;
    }

    // Refcount is now 0 and is not already deallocating. Try to set
    // the deallocating flag. This must be atomic because it can race
    // with weak retains.
    //
    // This also performs the before-deinit acquire barrier if we set the flag.
    static_assert(RC_FLAGS_COUNT == 2,
        "fix decrementShouldDeallocate() if you add more flags");
    uint32_t oldval = 0;
    newval = RC_DEALLOCATING_FLAG;
    return __atomic_compare_exchange(&refCount, &oldval, &newval, 0,
        __ATOMIC_ACQUIRE, __ATOMIC_RELAXED);
}
```

Swift Code 4: Operator zum Dekrementieren des starken Referenzzählers

Dabei wird der Wert im RefCount-Struct dekrementiert. Anschließend wird überprüft, ob der Wert ungleich 0 ist oder das deallocating-Flag gesetzt ist: In diesen Fällen muss der Aufrufer das Objekt nicht deallokieren, da entweder noch Referenzen auf das Objekt vorhanden sind oder bereits ein anderer Aufrufer aufräumt.

Wenn der Zähler auf 0 fällt, wird versucht, das deallocating-Flag zu setzen. Wenn dies gelingt, wird der Aufrufer dazu aufgefordert, das Objekt aufzuräumen. Andernfalls ist wieder ein anderer Aufrufer der Funktion in der Pflicht.

Sollte der Aufrufer aufgefordert werden, das Objekt aufzuräumen, wird der weak-Referenzzähler überprüft: Wenn dieser bei 1 liegt, wird das Objekt sofort aufgeräumt. Ansonsten wird der weak-Referenzzähler dekrementiert. Dies erklärt auch, warum der Zähler bei Initialisierung eines Objekts bereits bei 1 liegt: Die tatsächliche Deallokierung hängt vom weak-Zähler ab.

Das kann ein Problem darstellen, wenn das Objekt einen großen Speicherbereich verwaltet, und dieser durch eine schwache Referenz am Leben erhalten wird. Bei den typischerweise speicherintensiven Typen wie Arrays, Strings und Dictionaries ist dies in Swift nicht möglich, da diese als Struct implementiert sind und daher keine schwachen Referenzen zulassen.

Analyse der Deallokation

Bei Referenztypen, die potentiell viel Speicher belegen (zB Bilder in Form von UIImage oder binären Bäumen) muss selbst dafür gesorgt werden, dass der Buffer freigegeben wird: Dies kann im deinit-Block geschehen, da dieser ausgeführt wird, sobald die Anzahl von starken Referenzen auf null fällt.

Objekte, die durch das als deallokierend markierte Objekt referenziert werden, können ebenfalls deallokiert werden, was Swift auch selbstständig erledigt. Um dieses Verhalten zu analysieren, wurde die App um ein Beispiel erweitert, das einen Binärbaum verwaltet und Teile seines Speicher untersucht:

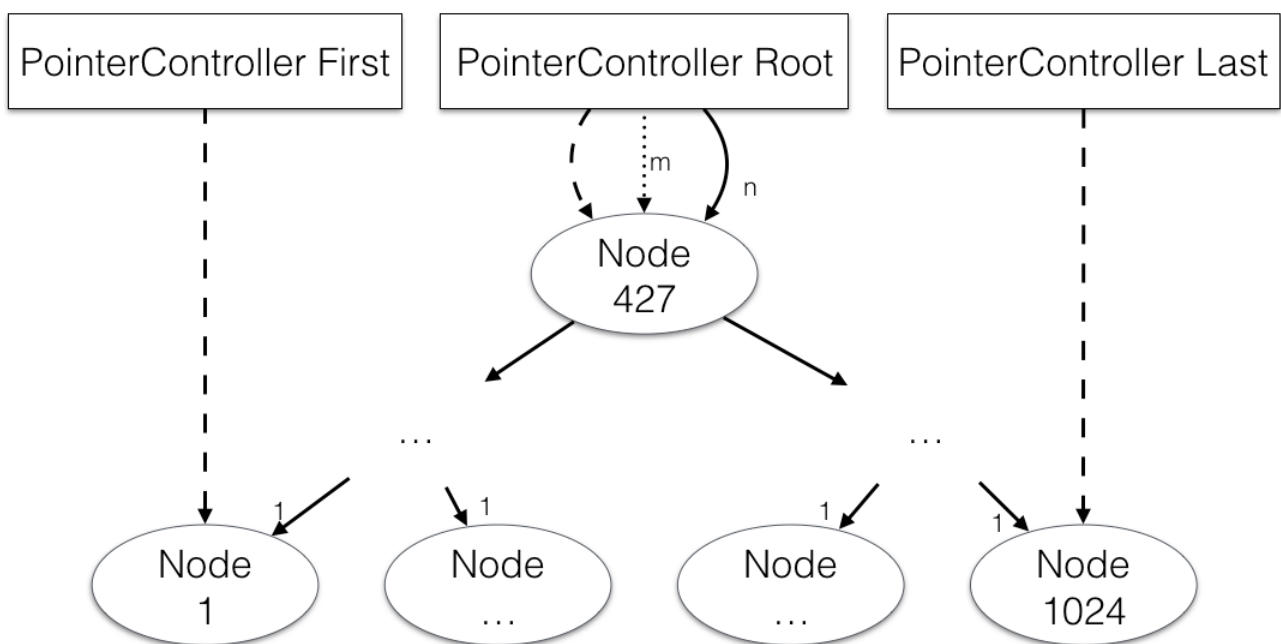


Bild 6: Ein Tree wird durch mehrere PointerController untersucht

Dabei wird der Binärbaum durch 3 PointerController untersucht: Einer, der den Wurzelknoten verwaltet (also strong und weak Pointer darauf besitzt) und untersucht, sowie zwei weitere, die das erste und das letzte Element des Tree untersuchen. Diese greifen nur lesend auf die Referenzzähler zu und zeigen lediglich die Anzahl in der UI der App an.

Der Binärbaum wurde mit Knoten aus Klassen implementiert, die jeweils eine starke Referenz zu ihren beiden Kindknoten besitzen und ein generisches Element beliebigen Typs speichern können. In diesem Beispiel wird ein Int in den Knoten abgelegt, da es sich dabei um eine Struktur handelt. Es muss eine Struktur sein, denn Klassen würden als Referenztyp wie die Kindknoten deallokiert werden, wenn ein Knoten als deallokierend markiert wird. Eine Analyse des Speichers wäre dann nicht möglich.

Die Knoten des Baums werden nur durch die Elemente in der Ebene darüber referenziert. Alle Elemente außer der Wurzel haben daher nur einen einzigen Zeiger, der die Elemente stark referenziert. Daraus folgt im Initialzustand ein Referenzzähler von 1 bei den starken sowie bei den schwachen Zählern.

Der Wurzelknoten des Baums verhält sich wie ein gewöhnliches Objekt: Wie im vorhergehenden Beispiel kann man sowohl starke als auch schwache Zeiger hinzufügen und entfernen, was sich wie erwartet im Referenzzähler widerspiegelt. Die Anzahl der schwachen Referenzen ist immer mindestens 1, solange man einen starken Zeiger auf das Objekt hat. Auch der erste und der letzte Knoten zeigen diesen impliziten schwachen Zeiger.

Interessant ist das Verhalten, wenn der Wurzelknoten als deallokierend markiert wird: Sämtliche Kindknoten werden freigegeben, auch wenn die Wurzel noch schwache Referenzen besitzt und das in der Wurzel gespeicherte Element noch gültig ist.

Um dieses Verhalten zu beschreiben lohnt sich erneut ein Blick in den Quellcode:

Tree		
Root:		
Strong	1	<input type="button" value="-"/> <input type="button" value="+"/>
Weak	1	<input type="button" value="-"/> <input type="button" value="+"/>
Element Value	427	
First Node:		
Strong	1	
Weak	1	
Element Value	1	
Last Node:		
Strong	1	
Weak	1	
Element Value	1024	
Reset Refresh		

App 5: Initialzustand:
Alle Knoten haben einen einzelnen strong Pointer

Tree		
Root:		
Strong	0	<input type="button" value="-"/> <input type="button" value="+"/>
Weak	2	<input type="button" value="-"/> <input type="button" value="+"/>
Element Value	427	
First Node:		
Strong	355236338	
Weak	1020	
Element Value	1260060675	
Last Node:		
Strong	311511948	
Weak	16322	
Element Value	0	
Reset Refresh		

App 6: Keine starken
Zeiger auf die Wurzel:
Alle Kindknoten sind
freigegeben

Phasenweise Deallokation von Objekten

Als Reaktion auf das Ergebnis der Funktion *doDecrementShouldDeallocate* sorgt die Swift-Runtime dafür, dass ein Objekt seine Instanzvariablen aufräumt. Dies geschieht durch Aufruf des Destruktors von Heapobjekten:

HeapObject.cpp

```
void SWIFT_RT_ENTRY_IMPL(swift_release)(HeapObject *object)
    SWIFT_CC(RegisterPreservingCC_IMPL) {
    if (object && object->refCount.decrementShouldDeallocate()) {
        _swift_release_dealloc(object);
    }
}
// ...
void _swift_release_dealloc(HeapObject *object)
    SWIFT_CC(RegisterPreservingCC_IMPL) {
    asFullMetadata(object->metadata)->destroy(object);
}
```

Swift Code 5: Reaktion auf Dekrementieren des Referenzzählers wenn dieser auf 0 fällt

Konkret im Beispiel der Klasse Node:

Node.swift

```
final class Node<T: Comparable where T: CustomStringConvertible> {
    var element: T
    private var left: Node?
    private var right: Node?

    init(_ element: T) {
        self.element = element
    }

    deinit {
        print("deinit Node #\(element)")
    }
}
```

Code 4: Die Klasse Node<T>

Die Klasse **Node** speichert ein Element (*element*) ab und verwaltet zwei weitere Knoten (*left* und *right*). Im deinit-Block wird nur das aktuell gespeicherte Element ausgegeben und explizit kein Speicher durch Entfernen der Referenzen freigegeben.

Durch einen Aufruf des Swift-Compilers erhält man folgenden SIL-Code (stark gekürzt):

```
Node.sil (swiftc Node.swift -emit-sil > Node.sil)
```

```
// Node.Node.deinit
// ...
bb0(%0 : $Node<T>):
  // ...
  // function_ref Swift.print...
  // ... (Ausführung Ausgabe deinit-Closure mit Ausgabe)

  %58 = ref_element_addr %0 : $Node<T>, #Node.element
  destroy_addr %58 : $*T

  %60 = ref_element_addr %0 : $Node<T>, #Node.left
  destroy_addr %60 : $*Optional<Node<T>>

  %62 = ref_element_addr %0 : $Node<T>, #Node.right
  destroy_addr %62 : $*Optional<Node<T>>

  return %57 : $Builtin.NativeObject
}
```

Code 5: SIL-Repräsentation der Klasse Node<T> (deinit)

Dabei wird zuerst der deinit-Block der Klasse ausgeführt (hier gekürzt, da das Ausgeben eines String-Literals mit Interpolation in SIL sehr viel Code erfordert). Danach werden die Properties der Klasse durch **destroy_addr** freigegeben. In der Beschreibung zu SIL wird die Funktion folgendermaßen beschrieben:

```
SIL.rst
```

```
destroy_addr %0 : $*T
```

Destroys the value in memory at address %0. If T is a non-trivial type, This is equivalent to:

```
%1 = load %0
strong_release %1
```

If T is a trivial type, then **destroy_addr** is a no-op.

Swift Code 6: Erklärung der SIL-Funktion destroy_addr

In unserem Beispiel werden also die starken Referenzzähler des linken und des rechten Knoten durch strong_release erniedrigt, was zu deren Deallokation führt, da die Referenz durch den darüberliegenden Knoten die einzige ist. Der in der Variable element gespeicherte Integer bleibt unverändert, da es sich hierbei um einen im Sinne von SIL trivialen Typ handelt.

Der vollständige Prozess bei Dekrementierung von Referenzzählern in Swift sieht also so aus:

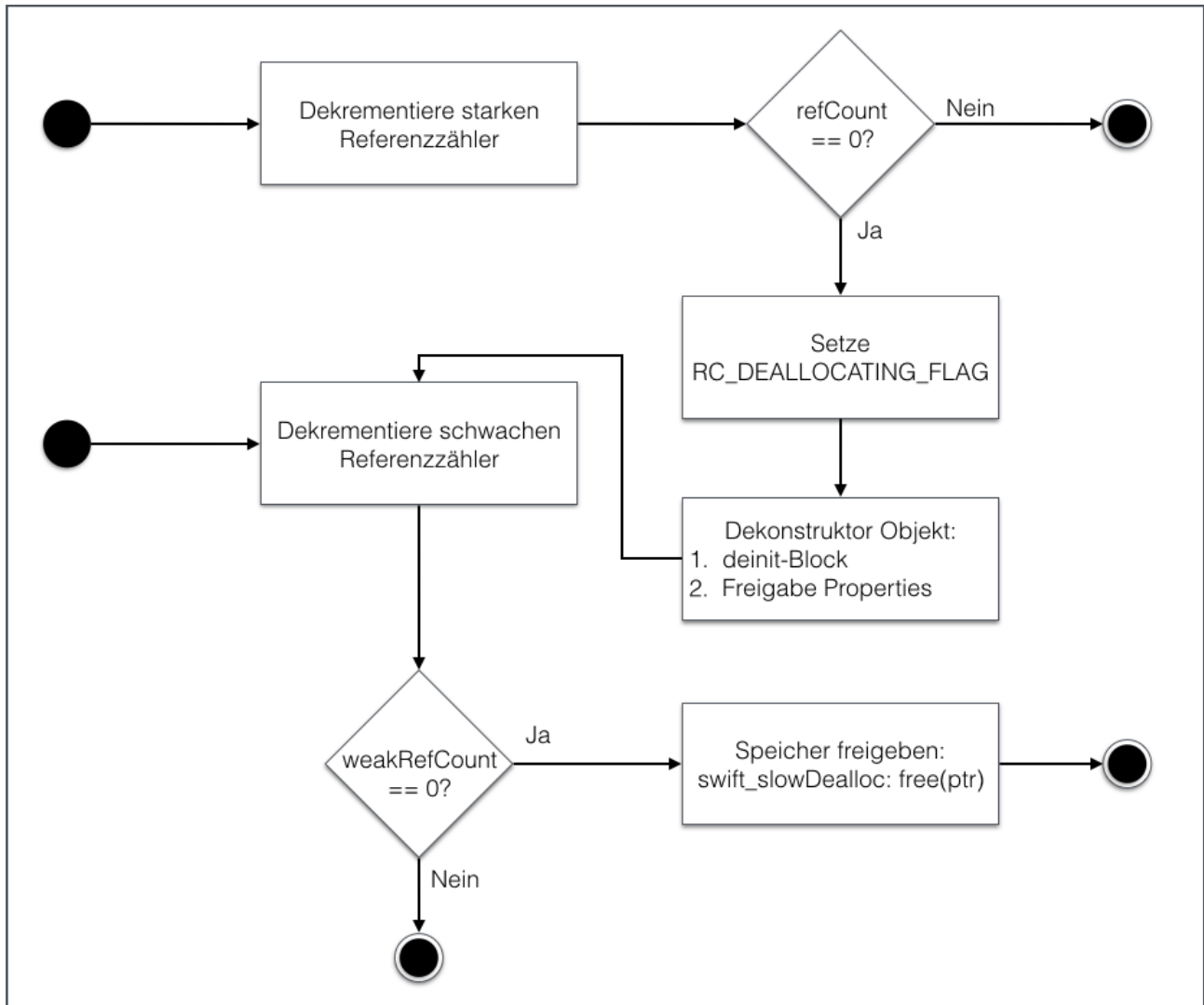


Bild 7: Aktivitätsdiagramm Dekrementierung und Deallokation

Nach einer Dekrementierung eines starken Zählers wird überprüft, ob dieser der letzte war. Falls nicht, passiert nichts, da andere Variablen das Objekt noch mit strong referenzieren. Falls der Zähler der letzte war, wird das Objekt als deallozierend markiert, indem das RC_DEALLOCATING_FLAG gesetzt wird. Daraufhin wird der deinit-Block des Objekts ausgeführt und die Properties des Objekts freigegeben: Wenn es sich bei einem Property um einen Referenztyp handelt, wird dessen Referenzzähler verringert. Wenn nicht, passiert nichts.

Anschließend wird der schwache Referenzzähler des Objekts erniedrigt und überprüft, ob der Zähler jetzt gleich 0 ist. Falls nicht, bleibt das Objekt (ohne seine Referenz-Properties) im Speicher und wartet, bis die schwachen Referenzen aufgelöst werden: Dies kann durch explizites Löschen oder durch Zugriffsversuche durch weak-Referenzen geschehen. Erst wenn die Anzahl der schwachen Referenzen auf 0 fällt, wird der Speicher des Objekts selbst freigegeben.

Wertsemantik mit Referenztypen

Werttypen mit Referenztyp-Implementierung

Werttypen spielen eine große Rolle in Swift: In der Standardbibliothek sind ca. 90% der Typen als Struct, 5% als Enumeration und der Rest als Klasse implementiert⁵. Dies liegt hauptsächlich an der leicht nachvollziehbaren und verargumentierbaren Wertsemantik, die Werttypen besitzen: Wertkopien und Immutabilität beugen vielen Flüchtigkeitsfehlern vor und vereinfachen die Logik in vielen Szenarien.

Andererseits führt häufiges Kopieren von großen Datenmengen zu einer schlechteren Performance als das vergleichsweise billige Kopieren von Zeigern auf große Datenmengen. Ein häufiges Beispiel ist die Parameterübergabe in Funktionen: Meist wird im Rumpf der Funktion nur lesend auf die Parameter zugegriffen, eine volle Kopie eines großen Objekts oder eines Arrays wäre reine Verschwendung.

In Swift wird dieses Problem durch Werttypen, die intern einen Referenztyp verwenden, umgangen: Strings und Arrays sind Structs, die im Hintergrund eine Klasse verwenden, um ihre Daten zu verwalten:

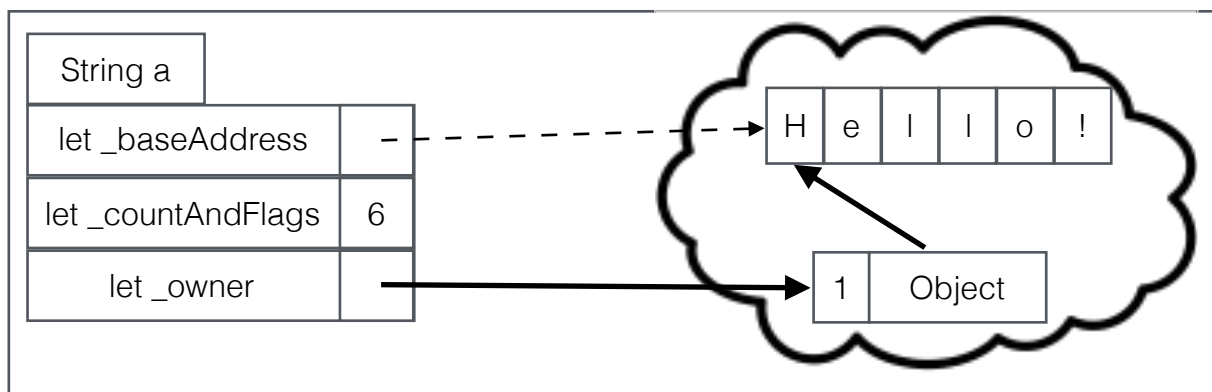


Bild 8: Struktur von Strings in Swift

In Swift besteht ein String aus einem einzelnen `_StringCore`-Struct. Dieses Struct hat:

- Einen `UnsafePointer` auf den Buffer des Strings (`_baseAddress`)
- Ein `UInt` (`_countAndFlags`), der die Länge und Flags⁶ des Strings beinhaltet
- Einen Pointer auf ein Objekt (`_owner`), das den Lebenszyklus des Buffers verwaltet.

⁶ Ein mögliches Flag ist beispielsweise, ob der Buffer ein zusammenhängender Swift-Buffer ist, oder ein Objective-C-typischer `NSString`.

Wird der String kopiert, so wird nur eine Kopie des _StringCore-Structs angelegt:

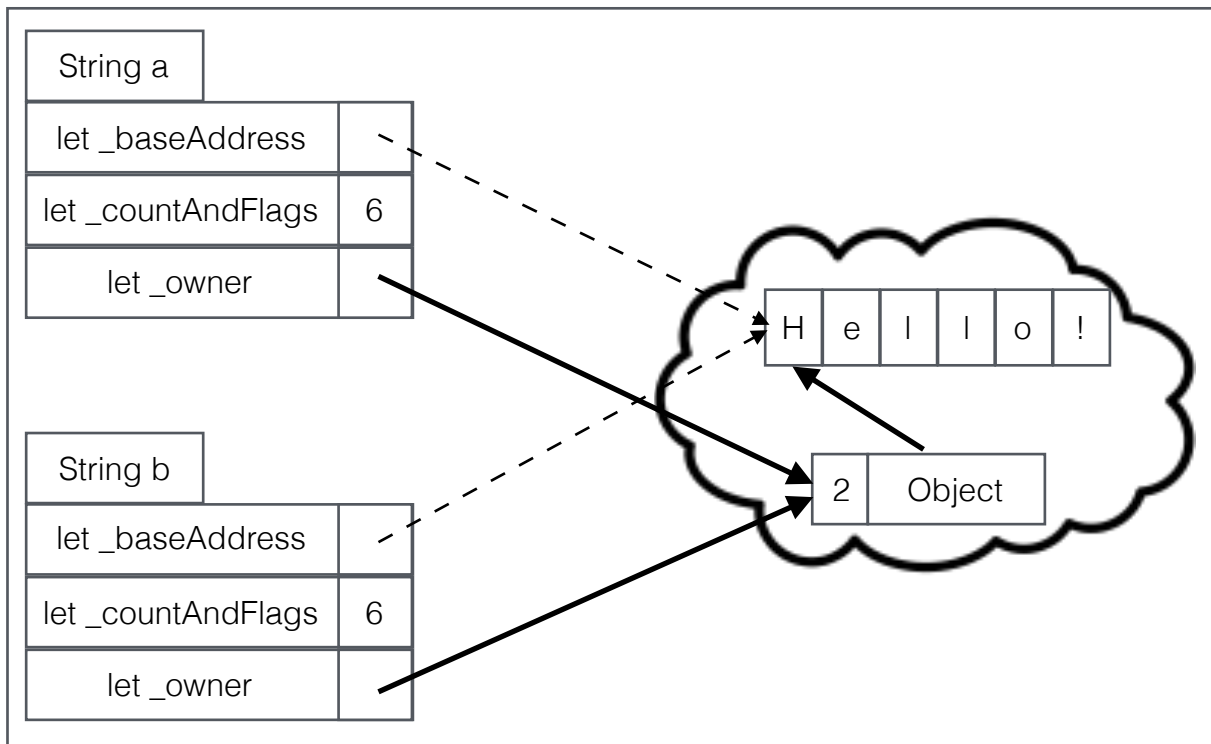


Bild 9: Ein String und eine Kopie davon teilen sich den selben Buffer

Diese Implementierung ist genauso effizient wie eine Implementierung durch Klassen: Der Buffer wird wiederverwendet und muss nicht bei jeder Kopie mitkopiert werden. Das Problem dabei: Die Wertsemantik ist nicht gegeben, da Mutationen eines Strings in allen Kopien sichtbar sind.

Copy-On-Write

Swift löst dieses Problem durch Copy-On-Write: Jedes Mal, wenn der Buffer verändert werden soll, wird eine Kopie angelegt und nur die Kopie verändert. Andere Variablen, die auf den selben Buffer zeigen bemerken die Mutation nicht und verhalten sich wie Werttypen. Auf diese Weise wird Wertsemantik gesichert, während in den meisten Fällen eine Effizienz wie bei Werttypen gegeben ist.

Copy-On-Write verwendet die Referenzzähler-Mechanik, um zu entscheiden, wann eine Kopie nötig ist: Bei Strings kommt hier das `_owner`-Objekt ins Spiel: Da es eine Klasse ist, hat es ein Feld für die Anzahl von starken Referenzen auf sich selbst. Diese Zahl muss immer gleich der Anzahl von Referenzen auf den Buffer sein: Schwache Referenzen auf den Buffer (und auf das `_owner`-Objekt) sind nicht möglich: Da Strings als Struct implementiert sind erlaubt der Compiler keine mit `weak` deklarierten Variablen. Es ist ebenfalls nicht möglich, die Referenz zum Buffer (oder dem `_owner`-Objekt) direkt zu kopieren und zu modifizieren, da das komplette `_StringCore`-Struct `private` ist. Daher benötigt der Buffer keinen eigenen Zähler und kann das eingebaute System für Klassen im `_owner`-Objekt⁷ nutzen.

Beim Versuch, den Buffer zu modifizieren wird der Referenzzähler des Objekts mit Hilfe der Funktion `isUniquelyReferencedNonObjC`⁸ überprüft, welche intern die in `RefCount.h` deklarierte Funktion `isUniquelyReferenced` aufruft. Für den Fall, dass nur eine einzelne Referenz existiert, kann der Buffer problemlos direkt modifiziert werden, da niemand anders die Mutation bemerken wird. Wenn mehrere Referenzen vorliegen, muss vor der Mutation eine Kopie des Buffers gemacht werden.

⁷ Das Objekt hat weiterhin die Aufgabe den Buffer freizugeben, wenn die Anzahl von Referenzen auf Null fällt: Da es eine Klasse ist, hat es eine `deinit`-Methode, welche in diesem Fall aufgerufen wird und den vom Buffer belegten Speicher freigibt.

⁸ Diese Funktion überprüft auch, ob die Swift-Implementierung von ARC verwendet wird (und nicht die von Objective-C). Es gibt eine weitere Funktion in der Runtime, `isUniquelyReferenced`, welche mit Objective-C-ARC funktioniert (aber nicht mit Swift-ARC). Für Swift 3 gibt es einen Vorschlag, diese API zu vereinheitlichen (SE-125)^{Q6}.

Ein Beispiel: Binary Tree mit Wertsemantik

Da die Copy-On-Write-Implementierung bei Strings, Arrays und anderen Standardbibliothek-Typen eher komplex ist, wird die Implementierung von Copy-On-Write an einem eigenen Typ gezeigt. Für dieses Beispiel wird der Binärbaum erweitert, der nach außen hin ein Struct ist, intern aber Klassen benutzt:

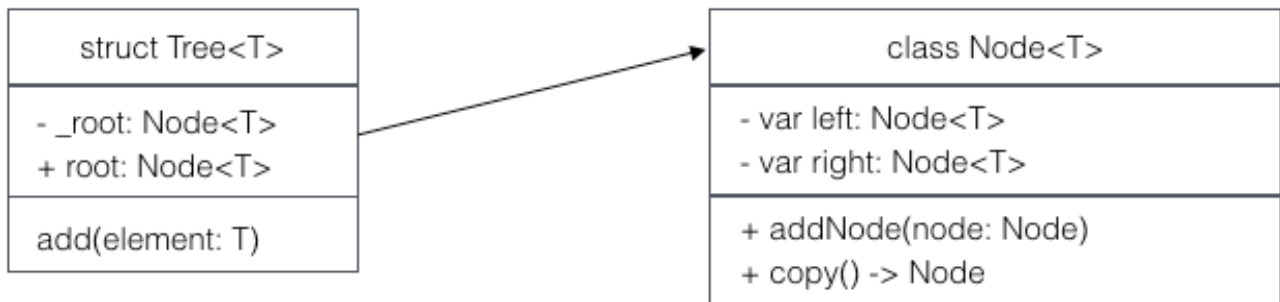


Bild 10: Strukturdiagramm Binärbaum mit Copy-On-Write-Semantik

Der Baum besteht aus Knoten der Klasse `Node`, welche die Elemente speichert und die Struktur abbildet. `Node` stellt Funktionen zum Hinzufügen von Elementen und zum Ausgeben des Tree bereit:

Die Kindknoten sind dabei *private*, damit keine Referenzen darauf angelegt werden können.

```
Node.swift

final class Node<T: Comparable where T: CustomStringConvertible> {
    var element: T
    private var left: Node?
    private var right: Node?

    init(_ element: T) {
        self.element = element
    }

    func addNode(newElement: T) {
        // Logik zum einfügen ...
    }
}

extension Node: CustomStringConvertible {
    var description: String {
        get {
            // Rekursives Traversieren und Konkatenieren der Elemente...
        }
    }
}
```

Code 6: Die Klasse `Node<T>` als Knoten des Binärbaums

Ferner gibt es eine copy-Methode: Sie legt eine Kopie des eigenen Elements an⁹ und kopiert auch sämtliche Kind-Knoten.

Node.swift

```
extension Node {
    func copy() -> Node {
        let element: T
        if self.element is NSObject {
            element = ((self.element as! NSObject).copy() as! T)
        } else {
            element = self.element
        }
        let node = Node(element)
        node.left = left?.copy()
        node.right = right?.copy()

        return node
    }
}
```

Code 7: Deep-Copy Methode der Klasse Node<T>

Der Tree an sich ist als Struct deklariert. Er bietet Methoden, die die Funktionen der Klasse Node bedienen:

Tree.swift

```
struct Tree<T: Comparable where T: CustomStringConvertible> {
    private var _root: Node<T>

    var root: Node<T> {
        // Öffentlicher Zugriff auf die Wurzel...
    }
    func add(element: T) {
        _root.addNode(element)
    }
    init(_ element: T) {
        _root = Node(element)
    }
}

extension Tree: CustomStringConvertible {
    var description: String {
        get { return _root.description }
    }
}
```

Code 8: Die Struktur Tree<T>

⁹ Dabei gibt es eine signifikante Lücke: Nicht alle Klassen definieren eine Copy-Methode. In Objective-C war das kein Problem, da die Wurzelklasse für alle Objekte (NSObject) eine copy-Methode definiert hat. In Swift gibt es ebenfalls kein zentrales Protocol zum Anlegen von Kopien von Klassen, auf das man sich beschränken könnte. Deswegen werden in der copy-Methode von Node nur Kopien von NSObject-Unterklassen explizit angelegt. Aus diesem Grund funktioniert das hier implementierte Copy-On-Write nur mit Trees, die Structs oder NSObject-Unterklassen speichern.

Das eigentliche Copy-On-Write findet in dem öffentlichen Zugriff auf die Wurzel über die Variable **root** statt:

Tree.swift

```
var root: Node<T> {  
    // Öffentlicher Zugriff auf die Wurzel  
    mutating get {  
        if !isUniquelyReferencedNonObjC(&_root) {  
            _root = _root.copy()  
        }  
        return _root  
    }  
    set {  
        _root = newValue  
    }  
}
```

Code 9: Copy-On-Write-Funktionalität im Getter der Wurzel

Dazu wird mit der Funktion `isUniquelyReferencedNonObjC` überprüft, ob der möglicherweise mutierende Zugriff möglich ist: Wenn der Wurzelknoten nur einmal referenziert ist, kann problemlos geschrieben werden. Andernfalls wird eine Kopie angelegt, und die neue Referenz hat ihren eigenen Tree (mit eigenem Speicherbereich), der nach Belieben modifiziert werden kann:

main.swift

```
let tree1 = Tree(3)  
tree1.add(1)  
tree1.add(2)  
tree1.add(4)  
tree1.add(5)  
  
var tree2 = tree1  
  
print("Tree 1: \(tree1)")  
print("Tree 2: \(tree2)")  
  
print("\n" + "Mutating Tree 2..." + "\n")  
tree2.root.element = 6  
  
print("Tree 1: \(tree1)")  
print("Tree 2: \(tree2)")
```

Code 10: Copy-On-Write in Aktion

Im Programm wird ein Baum **tree1** mit den Elementen 1 bis 5 angelegt. Dieser Baum wird kopiert und in der Variable **tree2** gespeichert. Daraufhin wird der Wurzelknoten vom in tree2 gespeichertem Baum verändert. Da auf diesen Knoten über die öffentliche Variable `root` (und nicht über die private Variable `_root`), greift der Copy-on-Write-Mechanismus:

Zunächst teilen sich die beiden Bäume eine Referenz auf denselben Speicherbereich:

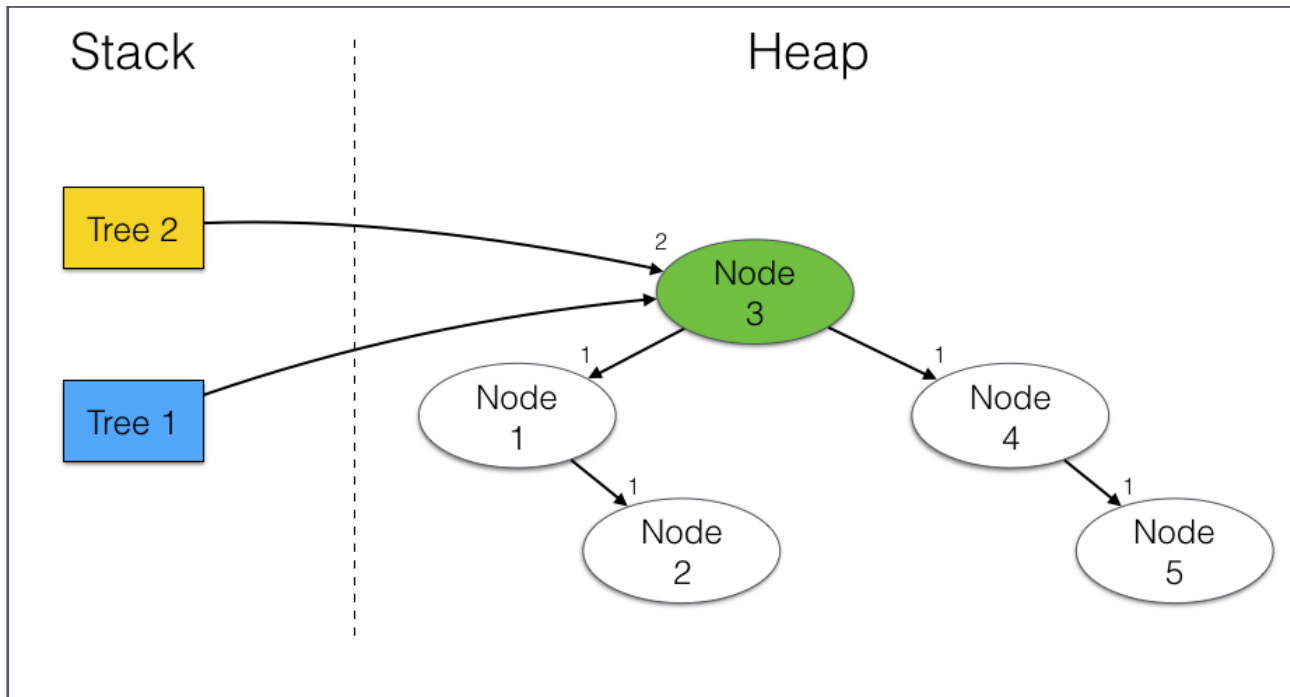


Bild 11: Copy-On-Write: Sharing vor Modifikation

Sobald aber in Zeile 13 das Element in tree2 verändert wird, wird durch den Getter eine Kopie des gesamten Tree angelegt, und nur dieser wird modifiziert:

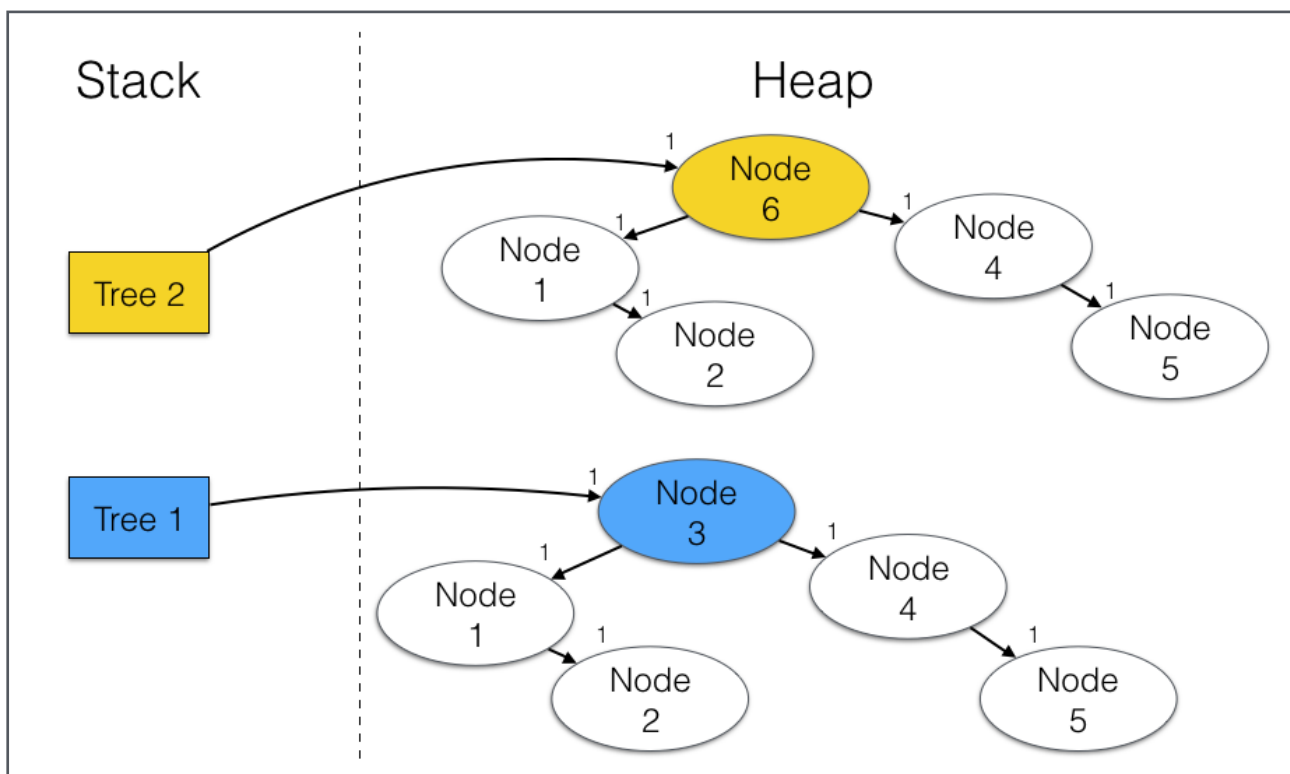


Bild 12: Copy-On-Write: Kopie bei Modifikation

Das Programm hat folgende Ausgabe:

STDOUT

```
Tree 1: 1, 2, 3, 4, 5
Tree 2: 1, 2, 3, 4, 5

Mutating Tree 2...

Tree 1: 1, 2, 3, 4, 5
Tree 2: 1, 2, 6, 4, 5
```

Code 11: Ausgabe des Programms: Copy-On-Write sichert Wertsemantik

Dabei ist zu betonen, dass auch bei lesendem Zugriff über das öffentliche Property `root` schon eine Kopie angelegt wird. Aus diesem Grund ist die Extension für `CustomStringConvertible` über das private `_root` implementiert: Ein Zugriff auf `root` würde im Falle von mehreren Referenzen eine Kopie bedeuten, auch wenn diese nicht modifiziert werden würde. Andere lesende Zugriffe (wie Iteratoren) sollten daher ebenfalls das private `_root` nutzen, während potentielle Schreibzugriffe (wie Subscript-Zuweisungen) das öffentliche `root` nutzen sollten, um die Semantik zu wahren.

Idealerweise kontrolliert man die komplette Schnittstelle eines Werttyps zur internen Repräsentation durch Referenztypen: Bei Arrays und Strings in Swift ist es nur schwer möglich, eine Referenz auf den tatsächlichen Speicherbereich des Buffers zu erhalten: Dies liegt zum Teil an der Kompatibilität zu `NSArray`, ist aber auch für CoW wichtig: Bei sämtlichen Operationen kann somit entschieden werden, ob eine Kopie angelegt werden muss oder nicht.

Bei Strings im Besonderen wird bei der Implementierung von ***replaceSubrange*** überprüft, ob die Funktion nur zum Anhängen (appending) führt. In diesem Fall kann die mutierende Funktion den Buffer wiederverwenden, auch wenn dieser von mehreren Stellen aus referenziert wird:

StringCore.swift

```
public mutating func replaceSubrange<C>(_ bounds: Range<Int>,
                                         with newElements: C) {
    // ...
    let appending = bounds.lowerBound == endIndex

    let existingStorage = !hasCocoaBuffer && (
        appending || isUniquelyReferencedNonObjC(&_owner)
    ) ? _claimCapacity(newCount, minElementWidth: width).1 : nil
    // ...
}
```

Swift Code 7: Optimierung von Copy-On-Write bei Strings in der Swift-Standardbibliothek

Dadurch, dass die komplette Schnittstelle auf den internen Buffer durch das String(Core)-Struct kontrolliert wird, kann die Absicht des Benutzers festgestellt werden. Dadurch wiederum können Kopiervorgänge gespart werden.

Bei Arrays gibt es einen kompletten Typ, ***ArraySlice***^{Q7}, der auf dem Sharing des Buffers von Array und ArraySlice basiert. Er ist dafür gedacht, eine Sicht auf einen Teil des Buffers zu repräsentieren, kann aber auch verwendet werden, um Mutationen (und damit Kopien des Buffers) zu bewirken.

Ein ArraySlice erhöht den Referenzzähler des Buffers und hält damit potentiell große Arrays, von denen nur ein kleiner Teil im ArraySlice noch benötigt wird, am Leben. In der Dokumentation wird daher davon abgeraten, ein ArraySlice länger zu behalten als das zugrundeliegende Array.

Let und Var

Einfluss des Variablentyps auf ARC

In Swift gibt es bei allem Variablen eine grundlegende Unterscheidung bezüglich der Beschreibbarkeit: Mit **let** deklarierte Variablen können genau einmal gesetzt werden und werden danach zu Konstanten, während das mit **var** spezifizierte Gegenstück beliebig oft überschrieben werden kann.

Bei let und var gibt es weiterhin Unterschiede zwischen Klassen und Strukturen: Bei Strukturen sorgt eine let-Deklaration dafür, dass sämtliche Properties darauf sich wie Konstanten verhalten, während bei Klassen nur die Referenz konstant wird, nicht aber das referenzierte Objekt:

1	<code>class Class {</code>	
2	<code> var a = 5</code>	
3	<code> let b = 6</code>	
4	<code>}</code>	
5		
6	<code>struct Struct {</code>	
7	<code> var a = 5</code>	
8	<code> let b = 6</code>	
9	<code>}</code>	
10		
11	<code>let c = Class()</code>	Class
12	<code>c.a = 10</code>	Class
13	<code>print("Class(a: \(c.a), b: \(c.b))")</code>	"Class(a: 10, b: 6)\n"
14		
15	<code>let s = Struct()</code>	Struct
16	<code>s.a = 10</code>	Struct
17	<code>print(s)</code>	"Struct(a: 10, b: 6)\n"
18		

Code 12: Unterschiedliches Verhalten von Variablenarten bei Wert- und Referenztypen

Diese Unterschiede lassen sich auf die Referenzsemantik zurückführen: Man könnte eine Referenz in einer var-Variable kopieren und dort das Objekt modifizieren, um das konstante Verhalten der Property zu umgehen. Statt diesen Workaround zu erzwingen, lässt Swift die Modifikation von let-deklarierten Klassen direkt zu.

Swift allokiert unter Umständen Klassen auf dem Stack, zum Beispiel wenn der Compiler erkennt, dass eine Instanz durch einen Gültigkeitsbereich eine begrenzte Lebensdauer hat und sichergestellt werden kann, dass keine Referenzen den Gültigkeitsbereich verlassen^{Q8}. Bei einer derartigen Allokation ist kein Referenzzähler nötig, da die Lebensdauer begrenzt ist und die Deallokation automatisch beim Stackabbau passiert. Diese Optimierung findet unabhängig von der Variablenart statt.

Eine ähnliche Optimierung könnte man bei Klassen, die ohnehin keine Referenzsemantik bieten, ebenfalls durchführen: Klassen, deren Properties alle mit let deklariert sind, sind nicht von

konstanten Strukturen zu unterscheiden. Hier stellt sich die Frage, ob Swift in diesem Fall auch eine Stackallokation ohne Referenzzähler vornimmt.

Um dieses Verhalten zu untersuchen werden folgende Kombinationen von Klassen und Strukturen erstellt und daraus SIL-Code generiert, um die Arten der Allokationen von Instanzen zu analysieren:

1. let- und var-Instanz von
2. einer Klasse und einer Struktur
3. mit einer let- und einer var-Instanz
4. einer Klasse und einer Struktur als Property

Daraus ergeben sich 16 mögliche Kombinationen. Diese Instanzen sind jeweils in einer weiteren Struktur verpackt, um die vorher genannten Optimierungen durch begrenzte Lebenszeit zu verhindern.

LetVsVar/Classes.swift

```
// ...
class ClassWithLetStruct {
    let content = 5
}

class ClassWithVarStruct {
    var content = 5
}
// ...
```

Code 13: Beispiele von Kombinationen von Variablenarten und Typen

Der SIL-Code wurde mit der kanonischen Option `-emit-sil` erstellt^{Q9}, welche grundsätzliche Sprachoptimierungen wie Stackallokation zulässt. Der erstellte Code wird daraufhin auf einige Merkmale untersucht, um die Unterschiede zwischen `let` und `var` deutlich zu machen:

- Getter und Setter für die Instanz
- Getter und Setter für das `content`-Property der Instanz
- wird `strong_retain` (also ARC) bei der Instanz verwendet?
- wird `strong_retain` (also ARC) bei dem `content`-Property verwendet?

Aus den Ergebnistabellen wird klar, dass die Wahl der Variablenart unabhängig von ARC ist:

StructWith...

Variable Type		Getter	Setter	.content Getter	.content Setter	strong_retain (instance)	strong_retain (content Property)
let	...LetStruct	1	0	1	0	0	0
let	...VarStruct	1	0	1	1	0	0
let	...LetClass	1	0	1	0	0	1
let	...VarClass	1	0	1	1	0	1
var	...LetStruct	1	1	1	0	0	0
var	...VarStruct	1	1	1	1	0	0
var	...LetClass	1	1	1	0	0	1
var	...VarClass	1	1	1	1	0	1

ClassWith...

Variable Type		Getter	Setter	.content Getter	.content Setter	strong_retain (instance)	strong_retain (content Property)
let	...LetStruct	1	0	1	0	1	0
let	...VarStruct	1	0	1	1	1	0
let	...LetClass	1	0	1	0	1	1
let	...VarClass	1	0	1	1	1	1
var	...LetStruct	1	1	1	0	1	0
var	...VarStruct	1	1	1	1	1	0
var	...LetClass	1	1	1	0	1	1
var	...VarClass	1	1	1	1	1	1

Die Unterscheidung von let und var schlägt sich lediglich darin nieder, ob für Instanzvariablen ein Setter generiert wird oder nicht. Auch wenn bei Klassen in Kombinationen wie beispielsweise "ClassWithLetStruct" keine Referenzsemantik möglich ist, bewahrt der Swift-Compiler die Allokation auf dem Heap.

Closures

Capture-Semantik

Closures sind Funktionen, die als Variable abgespeichert werden. Dabei werden Variablen, die für die Ausführung der Funktion benötigt werden, gespeichert, da eine Closure den Gültigkeitsbereich dieser Variablen verlassen kann. Dieses Abspeichern oder "Fangen" von Variablen nennt sich **Capture**. Um die Semantik eines Capture zu untersuchen, werden folgende Typen und Funktionen verwendet:

ClosureUtils.swift

```
func delay(seconds: NSTimeInterval, closure: ()->()) {
    let time = dispatch_time(DISPATCH_TIME_NOW,
                             Int64(seconds * Double(NSEC_PER_SEC)))
    dispatch_after(time, dispatch_get_main_queue()) {
        print("🕒 \((seconds) seconds passed 🕒")
        closure()
    }
}

class HeapWrapper {
    var content: Int
    init(content: Int) {
        self.content = content
    }
    deinit { print("\(content) (Wrapper) deinit call") }
}

struct StackWrapper {
    var content: Int
    init(content: Int) {
        self.content = content
    }
}
```

Code 14: Hilfsmittel zum Untersuchen von Closures

Die Funktion **delay** akzeptiert ein Zeitintervall und eine Closure als Parameter. Die übergebene Closure wird nach einem gewünschten Zeitraum ausgeführt. Durch die zeitlich versetzte Ausführung wird sichergestellt, dass der Stack Frame, in dem die Funktion aufgerufen wird, und dessen Variablen ungültig wird. Es ist auch möglich, dass die übergebene Closure in einem anderen Thread ausgeführt wird als der aufrufende Stack Frame.

Die Klasse **HeapBox** bildet einen Wrapper um einen beliebigen Typ. Der Typ wird dann unabhängig von seiner Natur auf dem Heap gespeichert. Wenn HeapBox deallokiert wird, wird der aktuelle Wert ausgegeben. Die Variable **content** ist absichtlich mutierbar.

Die Struktur **StackBox** bildet ebenfalls einen Wrapper um einen beliebigen Typ, diesmal werden aber Werttypen auf dem Stack gespeichert. Der Grund für die Existenz von StackBox ist die

ebenfalls mutierbare Variable **content**. So ist es möglich, eine Variable von StackBox zu mutieren, anstatt sie wie bei Structs üblich zu ersetzen.

In Swift werden die so gespeicherten Variablen zum Aufrufzeitpunkt der Closure evaluiert^{Q10}. Um dieses Verhalten zu zeigen, wird eine Instanz von HeapBox gespeichert und verspätet ausgeführt:

CaptureSemantics.swift	STDOUT
<pre>func test1() { var wrapper = HeapWrapper(content: 5) print("before closure: \(wrapper.content)") delay(1) { print("inside closure: \(wrapper.content)") } wrapper = HeapWrapper(content: 10) print("after closure: \(wrapper.content)") }</pre>	<pre>before closure: 5 5 (Wrapper) deinit call after closure: 10 ⌚ 1.0 seconds passed ⌚ inside closure: 10 10 (Wrapper) deinit call</pre>

Code 15: Capture-Semantik von Referenztypen: Test1 und STDOUT

In einem ersten Test wird eine Variable von HeapBox (also einem Referenztyp) angelegt, welche in einer Closure (angelegt durch die Trailing Closure Syntax der Funktion delay) gefangen wird. Danach wird der Variable ein komplett neuer Wert zugewiesen. Nach Ablauf einer Sekunde wird die Closure ausgeführt: Dabei kann beobachtet werden, dass die gefangene Variable die Neuzuweisung ebenfalls bemerkt hat.

Werttypen verhalten sich in Swift in diesem Fall genau so:

CaptureSemantics.swift	STDOUT
<pre>func test2() { var wrapper = StackWrapper(content: 5) print("before closure: \(wrapper.content)") delay(1) { print("inside closure: \(wrapper.content)") } wrapper = StackWrapper(content: 10) print("after closure: \(wrapper.content)") }</pre>	<pre>before closure: 5 after closure: 10 ⌚ 1.0 seconds passed ⌚ inside closure: 10</pre>

Code 16: Capture-Semantik von Werttypen: Test2 und STDOUT

Hier werden die selben Schritte wie im ersten Test ausgeführt, nur diesmal mit dem Werttypen StackBox. Dabei kann wieder beobachtet werden, dass die Closure bemerkt, dass sich die gefangene Variable verändert hat.

Gefangene Variablen können standardmäßig in der Closure mutiert oder neu zugewiesen werden. Diese Änderungen sind außerhalb der Closure sichtbar.

Offenbar wird hier eine Referenz zur gefangenen Variable gespeichert, und nicht eine Referenz zum Wert der Variable selbst. Closure Capture ist also Pass-By-Reference und nicht Pass-By-Value.

Um eine Pass-By-Value Semantik zu erreichen, muss eine Kopie der Variable für die Closure angelegt werden:

```
let wrapperCopy = wrapper
delay(1) {
// ...
```

Code 17: Closures mit Pass-By-Value-Semantik: Manuelles Anlegen einer Kopie

Stattdessen kann auch eine **Capture List** verwendet werden: Dabei werden verwendete Parameter in einer Liste am Anfang der Closure benannt. Folgende Notationen haben den gleichen Effekt:

```
delay(1) { [wrapperCopy = wrapper] in
delay(1) { [wrapper = wrapper] in
delay(1) { [wrapper] in
```

Code 18: Definieren von Kopien in einer Capture List

Dabei wird eine immutable Kopie von box angelegt. Wenn die Variable neu zugewiesen wird, bleibt die Kopie in der Closure davon unberührt:

CaptureSemantics.swift

```
func test3() {
    var wrapper = HeapWrapper(content: 5)
    print("before closure: \(wrapper.content)")
    delay(1) { [wrapper] in
        print("inside closure: \(wrapper.content)")
    }
    wrapper = HeapWrapper(content: 10)
    print("after closure: \(wrapper.content)")
}
```

STDOUT

```
before closure: 5
after closure: 10
10 (Wrapper) deinit call
🕒 1.0 seconds passed 🕒
inside closure: 5
5 (Wrapper) deinit call
```

Code 19: Capture-Semantik mit Capture List bei Referenztypen: Test3 und STDOUT

Das gleiche gilt für Werttypen:

CaptureSemantics.swift	STDOUT
<pre>func test4() { var wrapper = StackWrapper(content: 5) print("before closure: \(wrapper.content)") delay(1) { [wrapper] in print("inside closure: \(wrapper.content)") } wrapper = StackWrapper(content: 10) print("after closure: \(wrapper.content)") }</pre>	<pre>before closure: 5 after closure: 10 ⌚ 1.0 seconds passed ⌚ inside closure: 5</pre>

Code 20: Capture-Semantik mit Capture List bei Werttypen: Test4 und STDOUT

Bei der Mutation der gefangenen Variable gelten nun die selben Regeln für Wert- und Referenztypen wie beim Kopieren außerhalb von Closures: Bei Klassen sind Mutationen für alle Kopien sichtbar, während bei Strukturen die Kopien nicht von Mutationen betroffen sind:

CaptureSemantics.swift	STDOUT
<pre>func test5() { var wrapper = HeapWrapper(content: 5) print("before closure: \(wrapper.content)") delay(1) { [wrapper] in print("inside closure: \(wrapper.content)") } wrapper.content = 10 print("after closure: \(wrapper.content)") }</pre>	<pre>before closure: 5 after closure: 10 ⌚ 1.0 seconds passed ⌚ inside closure: 10 10 (Wrapper) deinit call</pre>
CaptureSemantics.swift	STDOUT
<pre>func test6() { var wrapper = StackWrapper(content: 5) print("before closure: \(wrapper.content)") delay(1) { [wrapper] in print("inside closure: \(wrapper.content)") } wrapper.content = 10 print("after closure: \(wrapper.content)") }</pre>	<pre>before closure: 5 after closure: 10 ⌚ 1.0 seconds passed ⌚ inside closure: 5</pre>

Code 21: Capture-Semantik mit Capture List: Kopien und Mutationen wie gewohnt in Test 5 und 6

Beim Verwenden von Capture Lists wird eine Liste von Variablen angegeben, die explizit für die Closure kopiert werden, und zwar als immutable Kopie. Diese Kopien verhalten sich wie Kopien außerhalb von Closures üblich: Bei Referenztypen ist die Kopie ein weiterer starker Zeiger auf das referenzierte Objekt, wodurch Mutationen sichtbar werden. Bei Werttypen ist die Kopie eine komplette Kopie des Wertes. Mutationen des Originals sind dabei nicht sichtbar.

Speicherlayout von Closures

Um die Capture-Semantik sowie die Leak-Problematik von Closures besser zu verstehen ist es wichtig, das Speicherlayout von Closures zu kennen. Dazu wird der generierte SIL-Code untersucht. Die vorgenommenen Allokationen werden dadurch sichtbar (Auszug aus dem SIL-Code der Funktion test1):

SIL

```
// main.test1 () -> ()
sil hidden @_TF4main5test1FT_T_ {
bb0:

    %0 = alloc_box $HeapWrapper, var, name "wrapper"
    // ...
}
```

Code 22: Anlegen der Variable "wrapper" vom Typ HeapWrapper in SIL

Dabei fällt sofort auf, dass die Klasse HeapWrapper nicht als Heap-Referenz (alloc_ref) oder als Stackallokation (alloc_stack) erstellt wird, sondern als Box auf dem Heap (**alloc_box**). Dabei wird auf dem Heap ein Referenzzähler-verwaltetes Objekt angelegt, welches den eigentlichen Typ beinhaltet.

Statt die Variable direkt zu modifizieren oder neu zuzuweisen, agiert der Code mit der Box, sowohl innerhalb als auch außerhalb von der Closure. Dies ist bei Referenz- und bei Werttypen der Fall.

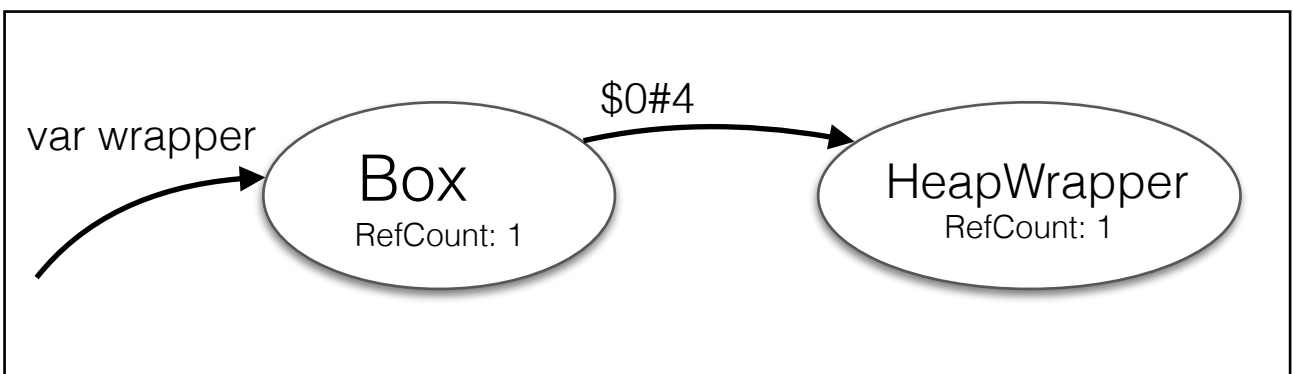


Bild 13: Closure Capture: Variablen werden auf den Heap verschoben

In den Beispielen, die eine Capture List verwenden wird keine solche Box angelegt. Das liegt daran, dass für jede Variable in der Liste eine Kopie erstellt wird und die Kopie in der Closure landet. Da die Kopie nicht mehr an anderer Stelle verändert wird kann der Compiler die Allokation der Box vermeiden^{Q11} und eine reguläre Referenz- oder Stackallokation durchführen.

Speicherlecks bei Closures, @noescape

Closures sind eine häufig anzutreffende Quelle von Speicherlecks, da hier unbemerkt Referenzzyklen entstehen können. Für einen Zyklus gelten zwei Bedingungen:

1. In der Closure muss **self** oder eine Instanzvariable mit **strong** referenziert sein
2. Die Closure muss als Variable gespeichert sein und die selbe oder eine längere Lebensdauer als **self** haben

Aufgrund der ersten Bedingung erzwingt der Compiler das in Swift bei Instanzvariablen sonst unnötige **self.**-Präfix, um die Capture-Semantiken und daraus möglicherweise entstehende Zyklen explizit darzustellen.

Wenn diese Bedingungen gelten, können Zyklen folgendermaßen entstehen:

Leak.swift

```
class Leak {
    var closure: (() -> ())!

    init() {
        closure = {
            print(self)
        }
    }

    deinit {
        print("Class 'Leak' deinit")
    }
}
```

1. 'Leak' hat einen strong Pointer auf 'closure'
2. 'closure' hat einen strong Pointer auf 'Leak'

Code 23: Eine Klasse mit Speicherleck durch eine Closure

Zur Laufzeit ergibt sich folgende Struktur:

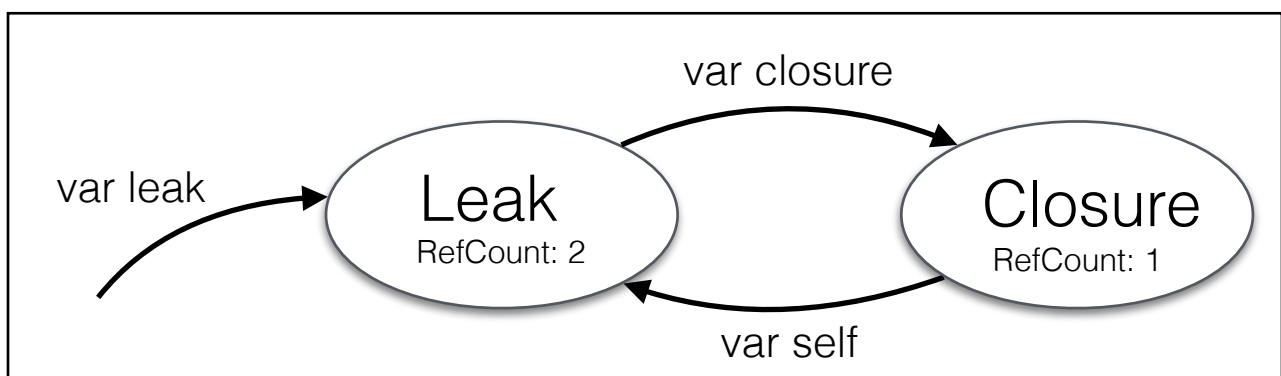


Bild 14: Eine Closure Erzeugt einen Referenzzyklus

Da Closures in Swift Referenztypen sind und durch ARC verwaltet werden und die Instanz der Klasse Leak einen strong Pointer zur Closure hat, wird die Closure so lange im Speicher gehalten, wie die Instanz von Leak im Speicher bleibt.

Gleichzeitig hat die Closure einen strong Pointer zur Instanz der Klasse Leak und hält diese Instanz so lange im Speicher, wie Closure im Speicher liegt.

Um einen solchen Zyklus aufzulösen, genügt es, eine der genannten Bedingungen zu verletzen:

1. Die Closure kann Klassen mit einem *weak* oder *unowned* Pointer referenzieren:

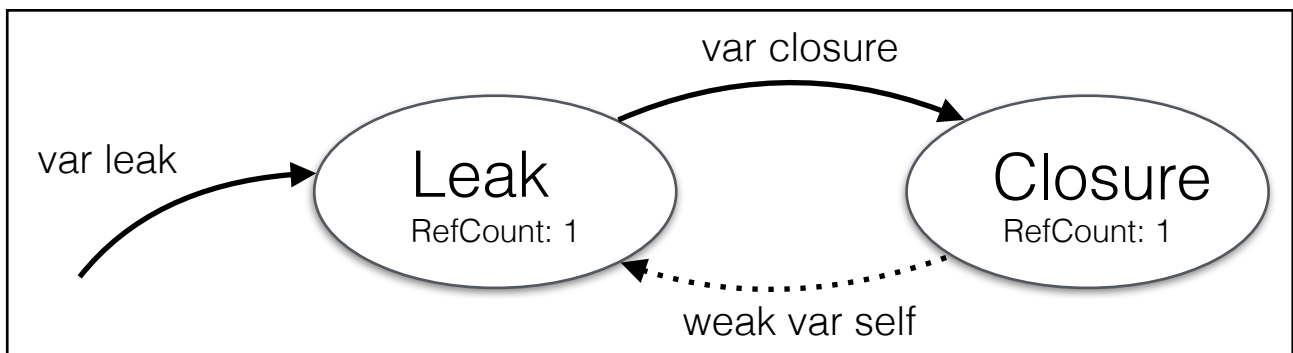


Bild 15: Durch die schwache Referenz wird ein Zyklus vermieden

Dazu muss eine Variable von self, die weak oder unowned deklariert ist, angelegt werden. Dies kann manuell oder in der Capture List geschehen:

```
Leak.swift

class Leak {
    // ...
    weak var weakSelf = self           // Manuelle Kopie
    closure = { [weak self] in         // Capture List
        print(weakSelf)
        print(self)
    }
    // ...
}
```

Code 24: Auflösung des Speicherlecks durch weak Capture in der Closure

Die beiden Ausdrücke sind semantisch gleich. In einer Capture List können auch andere Variablen von Referenztypen mit den Attributen *weak* oder *unowned* spezifiziert werden.

Was passiert mit Strukturen, die Speicherzyklen verursachen, wenn self in der Closure verwendet wird? Es ist nicht möglich, Strukturen mit weak oder unowned zu versehen, da diese nicht durch ARC verwaltet werden. Um hier eine Lösung zu finden, ist es hilfreich, das Speicherlayout von Closures mit gefangenen Strukturen anzusehen:

Dazu wird eine Struktur erstellt, die in einer Closure die Variable self fängt:

StructLeak.swift

```
struct StructLeak {  
    var closure: (() -> ())!  
  
    init() {  
        closure = {  
            print(self)  
        }  
    }  
}
```

Code 25: Eine Struktur mit Speicherleck

Hier wird im generierten SIL-Code der init-Funktion die Variable mit alloc_box erzeugt, was wieder ein Objekt mit Referenzzähler auf dem Heap anlegt:

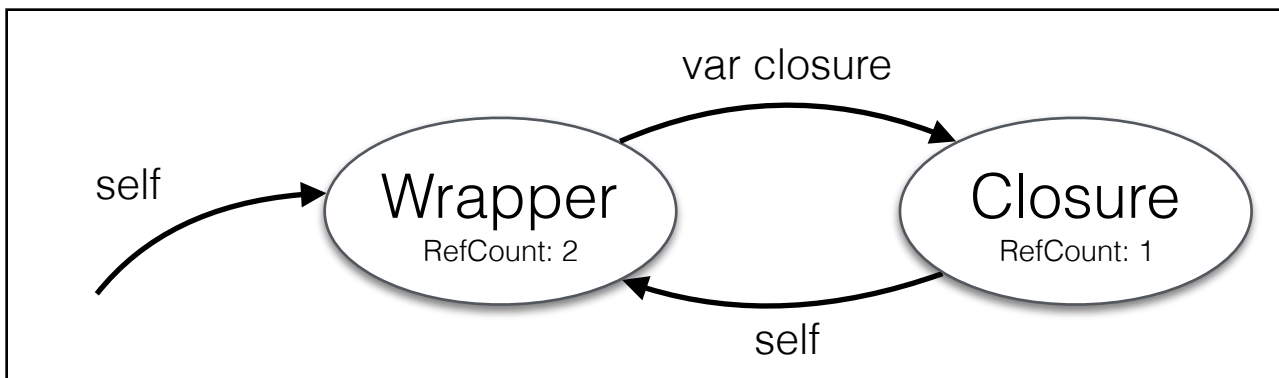


Bild 16: Speicherabbild der Struktur mit Speicherleck

Als Nebeneffekt geht auch die Wertsemantik von self in der Closure verloren: Da sich self jetzt auf dem Heap befindet referenzieren Variablen von StructLeak die Box (und es werden keine Kopien angelegt). Um diese Konstellation aufzulösen kann erneut in der Capture List oder manuell eine Kopie von self angelegt werden:

StructLeak.swift

```
closure = { [copy = self] in  
    print(copy)  
}
```

Code 26: Auflösung des Speicherlecks durch Capture einer Kopie in der Closure

Mit der Kopie in der Closure kann der Compiler wieder den Aufruf von alloc_box weglassen und eine Stackallokation durchführen. Durch die Kopie von self ist wieder Wertsemantik gegeben: Die Closure bemerkt nun keine Mutationen an self.

In Swift 3 sind Speicherzyklen mit alloc_box bei Werttypen nicht mehr möglich, da ein impliziter Capture von self bei Strukturen nur noch bei @noescape-Closures möglich ist¹⁰.

¹⁰ SE-0035: Limiting inout capture to @noescape contexts^{Q12}

2. Die Lebensdauer der Closure begrenzen:

Wenn die Closure deallokiert wird, bevor die Klasseninstanz deallokiert wird, hat diese keinen Zeiger mehr auf die Instanz und ARC kann die Instanz aufräumen. In unserem Beispiel dürfte die Klasse selbst keinen Zeiger auf die Closure speichern (oder müsste diesen Zeiger irgendwann löschen).

Bei Closures, die als Parameter an eine Funktion übergeben werden, kann man über das **@noescape**-Attribut sicherstellen, dass die Closure den Gültigkeitsbereich der Funktion nicht verlässt. Als Autor einer @noescape-Funktion hat man dadurch einige Einschränkungen^{Q13}:

- Man kann keine Referenzen auf die Closure anlegen
- Es ist nicht möglich, die Closure an eine Funktion zu übergeben, die kein @noescape-Attribut besitzt
- Man kann die Closure nicht auf einen anderen Thread übertragen

Als Nutzer einer @noescape-Funktion ist es nicht mehr nötig, Instanzvariablen mit dem Präfix **self.** zu adressieren, da ein Speicherzyklus ausgeschlossen ist.

Ein typisches Beispiel für @noescape ist die map-Funktion von Collections und Optionals:

```
enum Optional (declaration)

public enum Optional<Wrapped> : _Reflectable, NilLiteralConvertible {
    // ...
    @warn_unused_result
    public func map<U>(@noescape f: (Wrapped) throws -> U) rethrows -> U?
    // ...
}
```

Swift Code 8: Beispiel einer Funktion mit @noescape: Die Funktion map des Optional-Enums

Dabei wird über den Inhalt von Collections oder einem Optional iteriert und auf die Elemente die übergebene Closure angewendet. Danach wird die Closure nicht mehr benötigt. Das ganze geschieht synchron und auf dem gleichen Thread wie der Funktionsaufruf von map. Daher ist es nicht nötig, eine Variable der Closure zu speichern oder die Lebensdauer der Closure auf sonstige Weise zu verlängern und man kann den Parameter mit dem @noescape-Attribut versehen.

Ab Swift 3.0 werden @noescape-Closures zum Standard¹¹. Closures, die das alte Standardverhalten beibehalten wollen, müssen mit **@escaping** annotiert werden.

¹¹ SE-103: Make non-escaping closures the default^{Q14}

Fazit und Ausblick

In der Arbeit wurde gezeigt, wie Automatic Reference Counting implementiert ist und im Detail funktioniert. Durch den offenen Quellcode von Swift und den Zwischencode in Form der Swift Intermediate Language (SIL) konnte ein tiefer Einblick gegeben werden. Dabei wurde besonders auf die Deallokationsphase geachtet, welche in Swift sowohl die starken als auch die schwachen Zeiger verwendet.

Ferner wurde Copy-On-Write als weitere Anwendungsmöglichkeit der Referenzzähler gezeigt und Strings und Arrays als Swift-eigene Typen als Beispiele für diese Technik angeführt. Mit einem Binärbaum wurde ein eigener Copy-On-Write Datentyp, der Wertsemantik besitzt, im Inneren aber Referenztypen benutzt, implementiert.

Die Analyse der Variablentypen (let und var) ergab, dass diese orthogonal zu ARC operieren und generell keinen Einfluss auf die Referenzzählermechanik der Sprache hat.

Zum Ende wurden Closures, deren Speicherverwaltung von Variablen und die Capture-Semantik untersucht, wobei speziell letztere im Zusammenspiel mit self zu unerwarteten Speicherlecks führen kann. Daher wurde die Capture List besonders beleuchtet, welche diese Lecks auflösen kann.

Weitere Themen, die mit ARC verwandt sind, aber nicht behandelt wurden, sind ARC-Optimierungen durch den Compiler wie zum Beispiel Stack-Allokation von Referenztypen und das Versetzen strong_retain/release-Aufrufen in der SIL-Phase um diese beispielsweise aus Schleifen zu entfernen und somit nicht öfter als nötig durchzuführen. Außerdem wäre das Thema der Speicher-Fragmentierung generell bei Reference-Counting-Speicherverwaltungssystemen interessant: Bei Tracing Garbage Collection wird meist eine Defragmentierung durchgeführt, während die Laufzeit des Programms unterbrochen ist. Da das aber bei ARC nie passiert, ist eine solche Defragmentierung unmöglich.

In der näheren Zukunft hat Swift in der Version 3 nur kleinere Änderungen im Bezug auf ARC in Planung. Diese Änderungen betreffen fast ausschließlich Closures und versuchen Überraschungen und unerwartetes Verhalten besonders im Bezug auf Speicherlecks zu reduzieren. Ansonsten ist die ARC-Implementierung ein solider Grundstein für die Speicherverwaltung und Features wie Copy-On-Write und bedarf kaum an Verbesserungen. Für den Programmierer wären allerdings Hilfsmittel durch den Compiler wie statische Code-Analyse zum Auffinden von Speicherlecks hilfreich.

Anhang

Prüfungsrechtliche Erklärung

Ich, Xaver Lohmüller, Matrikel-Nr. 2427688, versichere, dass ich die Arbeit selbständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Xaver Lohmüller

Erklärung zur Veröffentlichung der nachstehend bezeichneten Abschlussarbeit

Angaben des bzw. der Studierenden:

Name: Vorname: Matrikel-Nr.:

Fakultät: Studiengang:

Sommersemester Wintersemester

Titel der Abschlussarbeit:

Die Entscheidung über die vollständige oder auszugsweise Veröffentlichung der Abschlussarbeit liegt grundsätzlich erst einmal allein in der Zuständigkeit der/des studentischen Verfasserin/Verfassers. Nach dem Urheberrechtsgesetz (UrhG) erwirbt die Verfasserin/der Verfasser einer Abschlussarbeit mit Anfertigung ihrer/seiner Arbeit das alleinige Urheberrecht und grundsätzlich auch die hieraus resultierenden Nutzungsrechte wie z.B. Erstveröffentlichung (§ 12 UrhG), Verbreitung (§ 17 UrhG), Vervielfältigung (§ 16 UrhG), Online-Nutzung usw., also alle Rechte, die die nicht-kommerzielle oder kommerzielle Verwertung betreffen.

Die Hochschule und deren Beschäftigte werden Abschlussarbeiten oder Teile davon nicht ohne Zustimmung der/des studentischen Verfasserin/Verfassers veröffentlichen, insbesondere nicht öffentlich zugänglich in die Bibliothek der Hochschule einstellen.

Hiermit ☒ genehmige ich, wenn und soweit keine entgegenstehenden Vereinbarungen mit Dritten getroffen worden sind,
☐ genehmige ich nicht,

dass die oben genannte Abschlussarbeit durch die Technische Hochschule Nürnberg Georg Simon Ohm, ggf. nach Ablauf einer mittels eines auf der Abschlussarbeit aufgebrachten Sperrvermerks kenntlich gemachten Sperrfrist

von Jahren (0 - 5 Jahren ab Datum der Abgabe der Arbeit),




der Öffentlichkeit zugänglich gemacht wird. Im Falle der Genehmigung erfolgt diese unwiderruflich; hierzu wird der Abschlussarbeit ein Exemplar im digitalisierten PDF-Format auf einem Datenträger beigelegt. Bestimmungen der jeweils geltenden Studien- und Prüfungsordnung über Art und Umfang der im Rahmen der Arbeit abzugebenden Exemplare und Materialien werden hierdurch nicht berührt.

.....
Ort, Datum

.....
Unterschrift der bzw. des Studierenden

Legende

Zeiger in Diagrammen

	Strong Pointer: Erhöht die Anzahl der starken Referenzen auf dem Objekt
	Weak oder Unowned Pointer: Erhöht die Anzahl der schwachen Referenzen
	Unsafe, raw oder sonstiger Pointer: Erhöht keine Referenzzähler

Beschreibungen / Verzeichnisse

Code #	Eigener Code, der für diese Arbeit erstellt wurde
Bild #	Diagramm, das für diese Arbeit erstellt wurde
Swift Code #	Zitate aus dem Swift-Quellcode, der Standardbibliothek oder der Dokumentation

Fußnoten und Quellen

100	Fußnote auf der selben Seite
Q100	Quelle aus dem Quellenverzeichnis (Präfix "Q")

Verzeichnisse

Code

#	Beschreibung	Seite
1	ARC in Swift in Aktion: Code und Konsolenausgabe	11
2	ARC in Swift in Aktion: Code und Konsolenausgabe	11
3	Erklärung des gezeigten Bitmusters als Referenzzähler	15
4	Reaktion auf Dekrementieren des Referenzzählers wenn dieser auf 0 fällt	23
5	Die Klasse Node<T>	23
6	SIL-Repräsentation der Klasse Node<T> (deinit)	24
7	Die Klasse Node<T> als Knoten des Binärbaums	30
8	Deep-Copy Methode der Klasse Node<T>	31
9	Die Struktur Tree<T>	31
10	Copy-On-Write-Funktionalität im Getter der Wurzel	32
11	Copy-On-Write in Aktion	32
12	Ausgabe des Programms: Copy-On-Write sichert Wertsemantik	34
13	Unterschiedliches Verhalten von Variablenarten bei Wert- und Referenztypen	37
14	Beispiele von Kombinationen von Variablenarten und Typen	38
15	Hilfsmittel zum Untersuchen von Closures	41
16	Capture-Semantik von Referenztypen: Test1 und STDOUT	42
17	Capture-Semantik von Werttypen: Test2 und STDOUT	42
18	Closures mit Pass-By-Value-Semantik: Manuelles Anlegen einer Kopie	43
19	Definieren von Kopien in einer Capture List	43
20	Capture-Semantik mit Capture List bei Referenztypen: Test3 und STDOUT	43
21	Capture-Semantik mit Capture List bei Werttypen: Test4 und STDOUT	44
22	Capture-Semantik mit Capture List: Kopien und Mutationen wie gewohnt in Test 5 und 6	44
23	Anlegen der Variable "wrapper" vom Typ HeapWrapper in SIL	45
24	Eine Klasse mit Speicherleck durch eine Closure	46
25	Auflösung des Speicherlecks durch weak Capture in der Closure	47
26	Eine Struktur mit Speicherleck	48
27	Auflösung des Speicherlecks durch Capture einer Kopie in der Closure	48

Diagramme

#	Beschreibung	Seite
1	ARC: Objekte werden freigegeben, sobald der Referenzzähler auf 0 fällt	9
2	Obwohl die Variable gelöscht wurde, können die Objekte nicht freigegeben werden	9
3	Parent-Child-Beziehung mit einem Weak Pointer: <i>Child</i> kennt zwar <i>Parent</i> , erhöht aber dessen Referenzzähler nicht. Wenn die Variable <i>ptr</i> gelöscht wird, kann <i>Parent</i> deallokiert werden, da der Zähler auf 0 sinkt.	10
4	Struktur von Heapobjekten in Swift	13
5	Der PointerController verwaltet unterschiedliche Zeiger auf ein Objekt	16
6	Ein Tree wird durch mehrere PointerController untersucht	21
7	Aktivitätsdiagramm Dekrementierung und Dealloktion	25
8	Struktur von Strings in Swift	27
9	Ein String und eine Kopie davon teilen sich den selben Buffer	28
10	Strukturdiagramm Binärbaum mit Copy-On-Write-Semantik	30
11	Copy-On-Write: Sharing vor Modifikation	33
12	Copy-On-Write: Kopie bei Modifikation	33
13	Closure Capture: Variablen werden auf den Heap verschoben	45
14	Eine Closure Erzeugt einen Referenzzyklus	46
15	Durch die schwache Referenz wird ein Zyklus vermieden	47
16	Speicherabbild der Struktur mit Speicherleck	48

Swift Code

#	Beschreibung	Seite
1	Deklaration der Referenzzähler	14
2	Bedeutung der Bits in den Referenzzählern	15
3	Atomare Operatoren zur Modifikation von Referenzzählern	19
4	Operator zum Dekrementieren des starken Referenzzählers	20
5	Reaktion auf Dekrementieren des Referenzzählers wenn dieser auf 0 fällt	23
6	Erklärung der SIL-Funktion <code>destroy_addr</code>	24
7	Optimierung von Copy-On-Write bei Strings in der Swift-Standardbibliothek	34
8	Beispiel einer Funktion mit <code>@noescape</code> : Die Funktion <code>map</code> des <code>Optional-Enums</code>	50

App Screenshots

#	Beschreibung	Seite
1	Initialzustand: ein einzelner starker Pointer ist gesetzt	17
2	Einige Pointer sind gesetzt	17
3	Die Anzahl der strong Pointers fällt auf 0	18
4	Die Anzahl der weak Pointers fällt auf 0. Der Speicher wurde freigegeben	18
5	Initialzustand: Alle Knoten haben einen einzelnen strong Pointer	22
6	Keine starken Zeiger auf die Wurzel: Alle Kindknoten sind freigegeben	22

Quellenverzeichnis

#	Beschreibung	Quelle	Seite
1	<u>Python Documentation:</u> Reference Counts & Reference Counting in Python (Python v 2.7.12)	https://docs.python.org/2/extending/extending.html#reference-counts	10
2	Structs and Classes: Unowned References (Seite 136)	Chris Eidhof, Airspeed Velocity: Advanced Swift (2016)	11
3	<u>Apple Documentation:</u> UnsafePointer Structure Reference	https://developer.apple.com/library/ios/documentation/Swift/Reference/Swift_UnsafePointer_Structure/index.html	12
4	<u>Mike Ash:</u> Friday Q&A 2015-12-11: Swift Weak References	https://www.mikeash.com/pyblog/friday-qa-2015-12-11-swift-weak-references.html	18
5	Structs and Classes (Seite 117)	Chris Eidhof, Airspeed Velocity: Advanced Swift (2016)	27
6	<u>Swift Enhancement:</u> Proposal 125: Remove NonObjectiveCBase and isUniquelyReferenced	https://github.com/apple/swift-evolution/blob/master/proposals/0125-remove-nonobjectivebase.md	29
7	<u>Apple Documentation:</u> ArraySlice Structure Reference	https://developer.apple.com/library/ios/documentation/Swift/Reference/Swift_ArraySlice_Structure/index.html	35
8	<u>Apple: Swift Docs:</u> Guaranteed Optimization and Diagnostic Passes: Memory promotion	https://github.com/apple/swift/blob/master/docs/SIL.rst#guaranteed-optimization-and-diagnostic-passes	37
9	<u>Apple: Swift Docs:</u> SIL in the Swift Compiler: SILGen und canonical SIL	https://github.com/apple/swift/blob/master/docs/SIL.rst#sil-in-the-swift-compiler	38
10	<u>Olivier Halligon:</u> Crunchy Development Blog: Closure Capture Semantics, Part 1: "Captured variables are evaluated on execution" (Artikel vom 25. Juli 2016)	http://alisoftware.github.io/swift/closures/2016/07/25/closure-capture-1/	42
11	<u>Chris Lattner, Joe Groff:</u> Swift Intermediate Language: "Promotion eliminates byref capture" (Vortrag am LLVM Developer's Meeting im Oktober 2015)	http://llvm.org/devmtg/2015-10/slides/GroffLattner-SILHighLevelIR.pdf	45

#	Beschreibung	Quelle	Seite
12	<u>Swift Enhancement:</u> Proposal 35: Limiting inout capture to @noescape context	https://github.com/apple/swift-evolution/blob/master/proposals/0035-limit-inout-capture.md	49
13	<u>NSHint Team:</u> NSHint: @noescape Attribute	http://nshint.io/blog/2015/10/23/noescape-attribute/	50
14	<u>Swift Enhancement:</u> Proposal 103: Make non-escaping closures the default	https://github.com/apple/swift-evolution/blob/master/proposals/0103-make-noescape-default.md	50

Sämtliche Internetquellen wurden am 6. August zuletzt aufgerufen und überprüft.

Das zitierte Buch "Advanced Swift" von Chris Eidhof und Airspeed Velocity ist am 18. März 2016 erschienen. Für diese Arbeit wurde eine Ebook-Version verwendet, die am 25. März aktualisiert wurde. Diese Version verwendet Swift 2.2.

Code

HEX-FUNKTION ZUR SCHÖNEREN AUSWERTUNG VON SPEICHERBLÖCKEN:

```
extension UnsignedIntegerType {  
    func hex(length: Int = 16) -> String {  
        var hexValue = String(self, radix: 16)  
        while (hexValue.characters.count < length) {  
            hexValue = "0" + hexValue  
        }  
        return "0x" + hexValue.uppercaseString  
    }  
}
```

Die Funktion ist als Extension auf das Protocol *UnsignedIntegerType* definiert. Als optionalen Parameter kann man eine gewünschte Länge angeben, standardmäßig wird auf 16 Stellen verlängert (was 64 Bit entspricht).

Zuerst wird ein String mit der Zahl zur Basis 16 initialisiert. An diesen String werden links so lange Nullen angehängt, bis die gewünschte Länge erreicht ist.

Zurückgegeben wird der verlängerte String mit dem für Hex-Zahlen übliche Präfix "0x".

IMPLEMENTIERUNG POINTERCONTROLLER

```
class PointerController<T> {
    let address: UnsafePointer<UInt32>

    private(set) var strongPointers = [T]()
    private(set) var weakPointers = [Weak<AnyObject>]()
    private(set) var unownedPointers = [Unowned<AnyObject>]()
    private(set) var unownedUnsafePointers = [UnownedUnsafe<AnyObject>]()

    init(something: T) {
        let someObject = something
        address = unsafeBitCast(someObject, UnsafePointer<UInt32>.self)
        strongPointers.append(someObject)
    }
}

// MARK: - Computed Properties
// MARK: Pointer analysis
extension PointerController {
    var strongCount: UInt {
        get {
            return UInt(address[1] >> 2)
        }
    }
    var rcPinnedFlag: UInt {
        get {
            return UInt(address[1] & 0x1)
        }
    }
    var rcDeallocatingFlag: UInt {
        get {
            return UInt((address[1] & 0x2) >> 1)
        }
    }
    var weakCount: UInt {
        get {
            return UInt(address[2] >> 1)
        }
    }
    var rcUnusedFlag: UInt {
        get {
            return UInt(address[2] & 0x1)
        }
    }
}

// MARK: Add new pointer
extension PointerController {
    var pointer: T? {
        get {
            if let strongPointer = strongPointers.first {
                return strongPointer
            } else {
                return nil
            }
        }
    }
}
}
```

```

// MARK: Functions

extension PointerController {
    // MARK: Strong
    func addStrongPointer() {
        if let strongPointer = pointer {
            strongPointers.append(strongPointer)
        }
    }
    func removeStrongPointer() {
        strongPointers.popLast()
    }
}

extension PointerController where T: AnyObject {
    // MARK: Weak
    func addWeakPointer() {
        if let strongPointer = pointer {
            weakPointers.append(Weak(strongPointer))
        }
    }
    func removeWeakPointer() {
        weakPointers.popLast()
    }
    // MARK: Unowned
    func addUnownedPointer() {
        if let strongPointer = pointer {
            unownedPointers.append(Unowned(strongPointer))
        }
    }
    func removeUnownedPointer() {
        unownedPointers.popLast()
    }
    // MARK: Unowned(unsafe)
    func addUnownedUnsafePointer() {
        if let strongPointer = pointer {
            unownedUnsafePointers.append(UnownedUnsafe(strongPointer))
        }
    }
    func removeUnownedUnsafePointer() {
        unownedUnsafePointers.popLast()
    }
}

```