3.5 Speicherverwaltung

Eigentlich sollte man sich bei höheren Programmiersprachen nicht um die Verwaltung des Laufzeitspeichers kümmern, weil dies die Aufgabe des Laufzeitsystems der jeweiligen Sprachimplementierung ist. Allerdings wird dies nicht in allen Sprachen voll unterstützt bzw. bieten Sprachen auch die Möglichkeit, das Laufzeitsystem zum Zweck einer effizienten Speicherverwaltung zu unterstützen. In diesem Kapitel wollen wir zunächst allgemein betrachten, wie der Laufzeitspeicher bei höheren Programmiersprachen verwaltet wird, um danach ein fortgeschrittenes Konzept zur Speicherverwaltung genauer anzuschauen: die Programmiersprache Rust und ihr „Besitzkonzept“ (ownership) zur effizienten Speicherverwaltung.

3.5.1 Einführung

Wie wir schon am Beispiel dynamischer Felder und Zeiger gesehen haben, ist es manchmal zur Laufzeit notwendig, neuen Speicherplatz für Objekte, deren Größe zur Übersetzungszeit nicht bekannt ist, zu reservieren. Dies kann z.B. Speicherplatz für eingelesene Strings, Datei- oder Datenbankinhalte oder daraus generierte Daten sein. Aus diesem Grund gibt es häufig explizite Sprachkonstrukte, wie `new` oder `malloc`, um neuen Speicherplatz anzulegen. Ebenso wird bei jedem Prozeduraufruf Speicherplatz für lokal definierte Variablen angelegt. Wenn allerdings immer nur Speicherplatz angelegt und nicht wieder freigegeben wird, steht irgendwann keiner mehr zur Verfügung!

Aus diesem Grund verwalten Implementierungen von Programmiersprachen zwei Arten von Laufzeitspeicher:

- Einen **Stack** (oder Prozedurkeller), auf dem bei Prozedureintritt Speicherplatz für lokale Variablen angelegt wird, der dann am Ende der Prozedurabarbeitung wieder freigegeben wird.
- Einen **Heap** für Strukturen, die eventuell länger existieren als eine Prozedurabarbeitung, z.B. in der Prozedur generierte Objekte, die als Rückgabewert verwendet werden.

Während der Stack regelmäßig bei Beendigung von Prozeduren verkleinert wird, ist es bei dem Heap nicht offensichtlich, wann er verkleinert werden kann. Daher gibt es zur Verkleinerung des Heaps zwei Möglichkeiten:

1. Der Programmierer benutzt Operationen (`free`, `deallocate`), um den Speicher für nicht mehr benötigte Strukturen freizugeben.
2. Ein automatisch ablaufender **Garbage Collector** durchsucht (ausgehend vom Stack) den Heap nach nicht mehr referenzierten Strukturen und gibt deren Speicher frei.

Die erste Alternative wurde in älteren Programmiersprachen (Pascal, C) eingesetzt, aber diese ist sehr fehleranfällig: falls Speicher freigegeben wird, auf den es noch Verweise gibt (**dangling pointers**), kann dies zu unverhersehbaren Ergebnissen führen.

Aus diesem Grund ist der Garbage Collector für sichere Programme vorzuziehen, sodass diese Methode in funktionalen, logischen und auch manchen imperativen Sprachen wie Java oder Go fest eingebaut ist. Manchmal gibt es auch Bibliotheken, wie für C, die einen Garbage Collector implementieren, aber deren Benutzung verlangt eine disziplinierte Programmerstellung.

Ein Nachteil des Garbage Collector ist sein Zeitverbrauch, denn dieser muss ausgehend vom Stack alle im Heap erreichbaren Strukturen markieren, eventuell auch kompaktizieren, um den nicht benötigten Speicher freizugeben. Zwar kann dies auch nebenläufig erfolgen, damit beim regulären Programmablauf keine Pausen entstehen, aber trotzdem verlangt der Garbage Collector Ressourcen. Aus diesem Grund wird bei der Systemprogrammierung wie in C häufig darauf verzichtet und eine unsichere Programmierung in Kauf genommen.

3.5.2 Besitzerprinzip von Rust

Eine interessante Alternative wird in der Programmiersprache Rust[6] realisiert. Diese ist als sichere Alternative zu C oder C++ konzipiert, weswegen sie auf eine Speicherverwaltung durch den Programmierer verzichtet. Sie verzichtet aber auch auf einen Garbage Collector. Hierzu wird der Heap Stack-ähnlich verwaltet, allerdings kann es auch dynamische Objekte geben, die in einer Prozedur erzeugt werden und länger leben. Um beides miteinander zu kombinieren, verwendet Rust ein **Besitzerprinzip** (**ownership principle**):

- Jeder Wert (ausgenommen sind einfache Werte skalarer Typen[7]) gehört zu einer Variablen, die als **Besizer** (**owner**) bezeichnet wird.
- Jeder Wert hat zu jedem Zeitpunkt höchstens einen Besitzer.
- Wenn ein Besitzer nicht mehr sichtbar ist (z.B. am Prozedurende), wird dessen Wert gelöscht bzw. sein Speicherplatz freigegeben.

Hier spielen also die Sichtbarkeitsregeln eine Rolle, denn diese bestimmen, wie lange Objekte gespeichert werden. Üblicherweise sind Variablen, die in Prozeduren deklariert werden, nur bis zum Prozedurende sichtbar. Die Sichtbarkeit kann aber auch durch Blöcke eingeschränkt werden (vgl. Kapitel 3.2.1):

```
{
  let x = 42;
  ... // x is visible and can be used here
}
... // here x is not visible and cannot be used
```

[6] https://www.rust-lang.org/

[7] Dies ist etwas vereinfacht dargestellt, denn Rust erlaubt es mittels der Copy-Eigenschaft auch andere Typen so zu markieren, dass diese immer ohne Besitzer kopiert werden.

Da skalare Werte nur konstanten Speicherplatz benötigen, werden diese nicht nach dem Besitzerprinzip verwaltet, sondern immer kopiert. Aus diesem Grund betrachten wir als einfaches Beispiel für komplexe Werte den Typ `String`, dessen Werte beliebig lange Zeichenketten darstellen, die z.B. mit der Methode `from` erzeugt werden können (in der `println`-Anweisung ist `{}` ein Platzhalter für die nachfolgenden Argumente):

```
let mut s = String::from("Hello");
s.push_str(", world!");    // append a string literal to a String
println!("Result: {}", s); // this will print "Result: Hello, world!"
```

Hier ist die Variable `s` der Besitzer des Strings und kann diesen verändern und ausgeben. Da ein Wert gelöscht wird, wenn sein Besitzer nicht mehr sichtbar ist, kann man auch auf seinen Wert nur zugreifen, wenn man diesen besitzt. Da es nur einen Besitzer für jeden Wert geben kann, muss bei einer Zuweisung der Besitz zwangsläufig weitergereicht werden. Aus diesem Grund liefert dieses Programm einen Compilerfehler:

```
let s1 = String::from("Hello");
let s2 = s1;                // now s2 is the owner of the string
println!("Result: {}", s2);
println!("Result: {}", s1); // ERROR: cannot use s1 here!
```

Durch die Zuweisung des Wertes von `s1` an `s2` ist danach die Variable `s2` der Besitzer des Strings. Dies ist auch sinnvoll, weil der String gelöscht werden soll, wenn `s2` nicht mehr sichtbar ist. Ansonsten könnte bei folgendem Programm, wenn es korrekt wäre, ein Laufzeitfehler (Zugriff auf nicht existierenden Speicherbereich) entstehen:

```
let s1 = String::from("Hello");
{
   let s2 = s1;    // now s2 is the owner of the string
   println!("Result: {}", s2);
}
println!("Result: {}", s1); // this will not compile!
```

Dies kann man vermeiden, falls es beabsichtigt ist, indem man mit der Methode `clone` ein Objekt kopiert, d.h. aus dem alten Objekt ein neues mit gleichem Inhalt erzeugt:

```
let s1 = String::from("Hello");
{
   let s2 = s1.clone();  // now s2 is the owner of a NEW string object
   println!("Result: {}", s2);
} // delete clone of the string owned by s2
println!("Result: {}", s1);
```

Mit dem Besitzerprinzip wird also jeder angelegte Speicher wieder freigegeben, und zwar dann, wenn der Besitzer nicht mehr zugreifbar ist. Ein expliziter Garbage Collector ist damit unnötig!

Was passiert nun bei Prozeduraufrufen? Wie wir schon gesehen haben, werden bei einer Wertübergabe die aktuellen Parameter den formalen Parametern zugewiesen. Damit geht auch der Besitz an die formalen Parameter über. Wenn die Werte der formalen Parameter am Ende der Prozedur nicht weitergegeben werden, werden diese also gelöscht. Daher wird das folgende Programm durch den Compiler nicht akzeptiert:

```
fn print_hello(s: String) { // s gets ownership
    println!("Hello {}", s);
}                           // s no longer in scope: delete s

fn main() {
    let w = String::from("world!");
    print_hello(w);
    println!("Result: {}", w); // ERROR: no ownership!
}
```

Falls der String nach Ausführung der Prozedur noch benötigt wird, können wir dies dadurch erreichen, dass dieser als Wert zurückgegeben wird und dadurch wieder eine Variable Besitzer des Strings wird. Das folgende Programm ist also korrekt:

```
fn print_hello(s: String) -> String { // s gets ownership
    println!("Hello {}", s);
    s                                 // return object s
}

fn main() {
    let w = String::from("world!");
    let w = print_hello(w);
    println!("Result: {}", w);
}
```

Man beachte, dass die neue Zuweisung der Variablen `w` einer neuen Deklaration entspricht, d.h. die Funktion `main` ist äquivalent zu folgender Definition:

```
fn main() {
    let w = String::from("world!");
    let z = print_hello(w);
    println!("Result: {}", z);
}
```

Wenn bei einem Prozeduraufruf der Besitz weitergegeben wird und wir den Besitz danach wieder benötigen, müssen wir den Wert zurückgeben. Dies ist etwas umständlich bei Prozeduren, mit denen man eigentlich etwas anderes berechnen will, aber mittels Tupel als Ergebnis ist dies einfach möglich. In dem folgenden Beispiel wird die Länge eines Strings berechnet, ohne dass der Besitz verloren geht:

```
fn string_length(s: String) -> (String, usize) {
    let length = s.len();    // len() returns the length of a String
    (s, length)
}

fn main() {
    let s = String::from("Rust language");
    let (s,n) = string_length(s);
    println!("Length of '{}' is {}", s,n);
}
```

Dieser Code sieht etwas umständlich aus, da wir nun Objekte in Prozeduren hineingeben und wieder herausbekommen, nur um den Besitz zu bewahren. Zum Glück gibt es eine einfache Methode, dies zu vermeiden. Statt das Objekt selbst zu übergeben, können wir auch eine **Referenz** auf das Objekt übergeben, die dann innerhalb der Funktion verwendet wird. D.h. der übergebene Parameter ist eine Referenz, die am Ende verworfen wird, aber das Objekt selbst wird nicht gelöscht. In Rust werden Referenztypen durch den Präfix „`&`“ gekennzeichnet, und „`*`“ und „`&`“ sind die Dereferenzierungs- und Adressoperatoren.

Somit ist das folgende Programm auch korrekt:

```
fn string_length(s: &String) -> usize {
    s.len()   // this abbreviates: (*s).len()
}

fn main() {
    let s = String::from("Rust language");
    let n = string_length(&s);
    println!("Length of '{}' is {}", s,n);
}
```

Wie man in der Definition von `string_length` sehen kann, fügt der Rust-Compiler automatisch Derefenzierungen ein, wo es notwendig ist.

Falls man in den referenzierten Objekten, die einer Prozedur übergeben werden, etwas verändern möchte, dann müssen diese als mutierbar deklariert werden, was analog zu Variablen mit dem Schlüsselwort `mut` gekennzeichnet wird. In dem folgenden Beispiel wird durch den Aufruf von `add_world` der übergebene String verändert, ohne dass der Aufrufer den Besitz darüber verliert.

```
fn add_world(s: &mut String) {
    s.push_str(", world");
}

fn main() {
```

```
    let mut w = String::from("Hello");
    add_world(&mut w);
    println!("Result: {}", w);
}
```

Wenn wir mit Hilfe von mutierbaren Referenzen Objekte verändern können, ohne deren Besitzer zu sein, könnte es dann nicht Probleme geben, wenn mehrere mutierbare Referenzen auf das gleiche Objekte verweisen und diese ändern? Tatsächlich ist dies ja in typischen imperativen Programmiersprachen zulässig, aber wir wissen auch, dass dies Probleme verursachen kann:

1. Wenn ein Objekt von verschiedenen Programmstellen und Variablen verändert wird, können die Effekte schwer zu überschauen sein, insbesondere weil dann auch die genaue Ausführungsreihenfolge für die Effekte relevant ist.

2. Wenn in einem nebenläufigen Programmablauf ein Objekt durch verschiedene Threads verändert und gelesen wird, kann es zu sog. *race conditions* kommen, d.h. ohne spezielle Synchronisationsmechanismen können logisch unerwartete Ergebnisse produziert werden (dies sollte aus Fortgeschrittener Programmierung bekannt sein, aber wir werden dies später auch noch diskutieren).

Aus diesen Gründen sind in Rust mehrere veränderbare Referenzen auf ein Objekt unzulässig. Somit führt das folgende Programm zu einem Compilerfehler:

```
let mut w = String::from("Hello");
let r1 = &mut w;
let r2 = &mut w;
println!("Results: {} and {}", r1, r2);
```

Wenn die Referenzen aber nicht veränderbar sind, entstehen diese Probleme nicht, sodass dieses Programm zulässig ist:

```
let w = String::from("Hello");
let r1 = &w;
let r2 = &w;
println!("Results: {} and {}", r1, r2);
```

Es ist nicht grundsätzlich verboten, mehrere veränderbare Referenzen auf das gleiche Objekt zu haben, nur dürfen diese nicht zur gleichen Zeit existieren. Aus diesem Grund ist das Programm

```
let mut w = String::from("Hello");
{
    let r1 = &mut w;
    print!("Results: {} ", r1);
}
let r2 = &mut w;
```

```
println!("and {}", r2);
```

zulässig, weil die Referenz `r1` nicht mehr zugreifbar ist, wenn `r2` eingeführt wird, weil der Gültigkeitsbereich von `r1` vorher endet. Interessanterweise ist auch das Programm

```
let mut w = String::from("Hello");
let r1 = &mut w;
print!("Results: {} ", r1);
let r2 = &mut w;
println!("and {}", r2);
```

zulässig. Zwar endet der Gültigkeitsbereich von `r1` nicht vor der Deklaration von `r2`, aber `r1` wird ab dieser Programmstelle nicht mehr benutzt, was durch den Rust-Compiler geprüft wird.

Wenn Zeiger in Programmiersprachen verwendet werden, gibt es immer ein potenzielles Problem: **dangling pointers** („herumhängende Zeiger"), d.h. es könnte sein, dass ein Zeiger auf ein Objekt verweist, welches schön gelöscht wurde (bei expliziter Speicherverwaltung durch den Programmierer). Da Rust auf Sicherheit setzt, schließt der Rust-Compiler so etwas aus. So ist das folgende Programm unzulässig:

```
fn dangle() -> &String {
    let hs = String::from("hello");
    &hs                              // ERROR: dangling pointer
}

fn main() {
    let s = dangle();
    println!("Result: {}", s);
}
```

Weil `hs` nach Abarbeitung der Funktion `dangle` nicht mehr gültig ist, wird das String-Objekt gelöscht. Damit wäre die zurückgegebene Referenz auf dieses Objekt ungültig!

Wenn man die Prozedur aber so verändert, dass nicht eine Referenz, sondern das Objekt selbst zurückgegeben wird, ist alles in Ordnung:

```
fn nodangle() -> String {
    let hs = String::from("hello");
    hs
}

fn main() {
    let s = nodangle();
    println!("Result: {}", s);
}
```

Somit können wir die zwei folgenden Prinzipien von Referenzen in Rust festhalten:

1. Zu jedem Zeitpunkt gibt es immer nur höchstens eine mutierbare Referenz oder alternativ mehrere nicht-mutierbare Referenzen auf ein Objekt.

2. Jede Referenz ist gültig, d.h. sie verweist nicht auf ein Objekt ohne Besitzer.

Diese Prinzipien von Besitz und Referenzen werden durch den Rust-Compiler geprüft, sodass man damit sichere und effiziente Programme (d.h. ohne Garbage Collector) erstellen kann. Damit ist Rust also eine sichere Alternative zu anderen Sprachen für die Systemprogrammierung, wie z.B. C. Diese Sicherheit verlangt etwas mehr Programmierdisziplin und ein interessantes Sprachkonzept von Besitz, welches wir hier vorgestellt haben. Die hier skizzierten Sprachkonstrukte sind noch nicht ganz ausreichend für die praktische Programmierung. So bietet Rust noch ein Konzept von *Lebenszeit* von Objekten, welches benutzt werden muss, wenn der Rust-Compiler alleine nicht in der Lage ist, die korrekten „Besitzverhältnisse“ zu analysieren. Darüberhinaus bietet Rust auch generische Typen, Methoden, Überladung, Pakete und einiges mehr, wodurch Rust zu einer vollwertigen Sprache auch für komplexe Anwendungen wird.