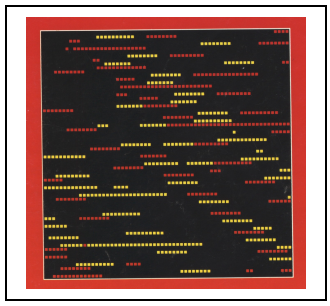
# Einführung in CORE WAR Teil 1

## Computerprogramme mal ganz anders



Ein ungewöhnliches Szenario in einem Computerspeicher: Zwei Programme, anstatt zusammenzuarbeiten, legen sich gegenseitig Fallen, beschießen sich mit Datenmüll und lassen voneinander nicht eher ab, als bis eines von ihnen die Segel streichen muss. Um aus diesem erbarmungslosen Kampf siegreich hervorzugehen, versuchen sich die Programme in den unterschiedlichsten Strategien: Man spielt va banque, indem man Bomben wirft und hofft, dass damit der Gegner eher zur Strecke gebracht wird als man selbst; oder man geht in die Verteidigung, verbindet seine Wunden und wartet auf eine günstige Gelegenheit, um dem Widersacher präzise einen empfindlichen Hieb zu versetzen. Eine grundsätzlich andere Möglichkeit ist aus der Natur der Kaninchen entlehnt. Ein Programm pflanzt sich möglichst schnell fort. Wenn der Gegner nun ein Kaninchen – etwa einen Enkel des ursprünglichen Programms – erlegt hat, werden die Vettern, Töchter und Tanten sowie alle anderen darüber wohl bekümmert sein, aber sie selbst haben nichtsdestotrotz gute Überlebenschancen, da es inzwischen möglicherweise hunderte ihrer Art gibt.

Dies ist die grausam schöne Welt des 1984 von A. K. Dewdney erfundenen Spiels Core War – zu deutsch Krieg der Kerne. Es gibt zwei Spieler, von denen jeder ein solches Kampfprogramm entwirft. Sind die beiden Kämpfer fertiggestellt, werden sie in die Freiheit entlassen und ihre geistigen Mütter oder Väter können ab jetzt für ihre Schützlinge nichts mehr tun.

## MARS heißt der Gott des Krieges

Leider sind die wilden Jahre der Computertechnologie vorbei, in denen jedes Programm alles überall in den Speicher schreiben durfte. Man kann Core War nicht ohne weiteres in dem Speicher eines Großcomputers, zu dem man vielleicht von Berufs wegen Zugang hat, spielen – ganz davon abgesehen, welchen Ärger Sie sich damit einhandelten, wenn Sie wichtige Daten anderer Leute mit Ihrer Spielfreude vernichten.

Damit nicht völlige Anarchie in Ihrem Computer ausbricht, gibt es einen obersten Herrn – im weiteren MARS (= Memory Array Redcode Simulator) genannt –, der die Kämpfer in seine Obhut nimmt und das Kampfgeschehen kontrolliert. MARS reserviert einen gewissen Platz im Speicher des Computers, auf dem Sie spielen, als Arena für den Kampf und geleitet die beiden Streithähne einzeln dorthin, so dass keiner weiß, wo der andere steht. Während der Schlacht sind die Kämpfer weitgehend sich selbst und ihrer Kampfeslust überlassen: MARS achtet aber peinlich genau darauf, dass die Programme abwechselnd zum Zuge kommen. Nachdem der Kampf dann sein bitteres Ende gefunden hat, übernimmt MARS wieder vollständig die Kontrolle und räumt das Schlachtfeld auf.

## Der Speicher ist der Kampfplatz

Die Speicherarena – oder kurz Arena – ist aus einzelnen Zellen aufgebaut. Da es sich bei den Kämpfern um Computerprogramme handelt, die aus Anweisungen bestehen, sind die Speicherzellen der Arena so angelegt, dass in jede Zelle eine Anweisung passt.

Die Zellen der Arena sind durchnumeriert; die Nummer einer Speicherzelle heißt Adresse. Die Arena ist ringförmig geschlossen, wobei die Adressierung mit Null beginnt. Eine Arena mit 8000 Zellen hat also die absoluten Adressen 0, l, ..., 7999. Höhere und niedrigere Adressen werden durch modulo-Arithmetik in den vorhandenen Adressbereich abgebildet. So ist in unserem Beispiel die Adresse 8000 gleichbedeutend mit der Adresse 0, die Adresse -2 gleichbedeutend mit 7998.

Der Benutzer kennt die absoluten Adressen der Arena nicht. Man stellt sich vielmehr auf den Standpunkt, dass man der Nabel der Welt ist, und adressiert die Speicherzelle, aus deren Blickwinkel man die Arena gerade betrachtet, mit Null. Die anderen Speicherzellen werden dann mit -10, +4, +2 usw. adressiert, je nachdem, wie weit sie von jener Speicherzelle in positiver oder negativer Richtung entfernt sind. Man bezeichnet dieses Vorgehen als relative Adressierung.

## Einige Gladiatoren

Die Kämpfer sind Computerprogramme, die in einer maschinennahen, assembler-artigen Programmiersprache namens Redcode formuliert sind. Dies klingt, als ob Ihnen eine solide Hackerausbildung bevorstünde; seien Sie aber beruhigt: Redcode kennt im wesentlichen nur sechs verschiedene Anweisungen.

Es ist wohl die beste Methode, eine Programmiersprache zu erlernen, wenn Sie sich an Hand einiger Beispielprogramme die Wirkungsweise der Anweisungen und ihr Zusammenspiel klarmachen; durch eigene Versuche werden Sie dann sehr schnell herausfinden, welch große Wirkung Sie mit kleinen Änderungen erzielen können.

### Knirps - klein, aber oho!

Redcode-Programme müssen mindestens eine Anweisung enthalten; länger ist auch das kürzeste Redcode-Programm nicht, das den bezeichnenden Namen *Knirps* trägt.

```
; Redcode-Programm „Knirps“:
      MOV 0 1     ; Die Anweisung kopiert sich selbst in die
                  ; nächste Speicherzelle
      END
```

Zunächst können Sie an diesem Programm die generelle Form einer Redcode-Anweisung erkennen: Jede Anweisung besteht aus einem Anweisungskürzel – hier **MOV** (nach dem englischen Wort move für bewege) – und zwei Operanden, die im weiteren mit Operand A und Operand B bezeichnet werden.

Was geschieht nun, wenn die obige **MOV**-Anweisung ausgeführt werden soll? Die beiden Operanden geben jeweils eine Adresse einer Speicherzelle an. Die **MOV**-Anweisung kopiert nun den Inhalt von der ersten Speicherzelle in die zweite. Da Adressen immer relativ zu der Adresse der aktuellen Anweisung angegeben werden, gibt der Operand A die Speicherzelle an, in der die Anweisung selbst steht. Der Operand B ist eins; er gibt also die direkt nachfolgende Speicherzelle an. Also kopiert sich die Anweisung selbst in die nachfolgende Speicherzelle.

Die letzte Zeile des Programms *Knirps* enthält die **END**-Direktive, die dem Assembler anzeigt, dass an dieser Stelle der Programmtext beendet ist.

Leerzeilen oder die mit dem Semikolon (;) beginnenden Kommentare dienen nur der besseren Verständlichkeit der Programme, werden aber bei der Ausführung ignoriert.

| Speicher-adresse | Anzahl ausgeführter Anweisungen | | |
|---|---|---|---|
| | 0 | 1 | 2 |
| *n - 1* | | | |
| *n* | **-> MOV 0 1** | **MOV 0 1** | **MOV 0 1** |
| *n + 1* | | **-> MOV 0 1** | **MOV 0 1** |
| *n + 2* | | | **-> MOV 0 1** |

Der Pfeil in der Tabelle markiert stets die Anweisung, die gerade ausgeführt werden soll.

Betrachten Sie nun das Verhalten von *Knirps*: Zu Beginn wird die Anweisung **MOV 0 1** in die Speicherzelle n +1 kopiert. Da Redcode-Anweisungen nacheinander abgearbeitet werden, wird Knirps im nächsten Schritt diejenige Anweisung ausführen, die in der (n+ l)-ten Speicherzelle steht – also **MOV 0 1**. Schritt für Schritt schreibt Knirps jeweils in eine neue Speicherzelle **MOV 0 1**. Er rast somit in atemberaubendem Tempo durch die Arena, eine lange Spur von **MOV**-Anweisungen hinter sich lassend. Darin liegt nun gerade sein Erfolg begründet: Wenn es dem Gegner nicht binnen kürzester Zeit gelingt, Knirps mit einem gezielten Schuss aus dem Rennen zu werfen, wird er von dem tumben Knirps überrannt.

Quelle: Aus dem Begleitheft zu *Core War* – Krieg der Kerne, Spektrum der Wissenschaft 1993, S. 23 ff.