Programmieren in C

Prof. Dr. Peter Thiemann
Hannes Saffrich
Sommersemester 2021

Universität Freiburg
Institut für Informatik

Übungsblatt 5

Abgabe: Montag, 31.05.2021, 9:00 Uhr morgens

In diesem Übungsblatt geht es weiterhin um Zeiger, dynamisch allozierten Speicher und Datenstrukturen.

Dieses mal sollen Sie eine Datenstruktur für `int`-Arrays implementieren, die sich wie Python-Listen verhalten – also für die es eine Funktion `append` gibt, die ein weiteres Element zum Array hinzufügt und bei Bedarf das Array automatisch vergrößert.

Anschließend sollen Sie ein Programm schreiben, das es erlaubt beliebig viele Datenpunkte (Zahlen) einzugeben und für die Datenpunkte eine statistische Auswertung ausgibt. Da im Voraus nicht bekannt ist wie viele Datenpunkte eingegeben werden, ist dies gerade eine Anwendung des `int`-Arrays.

Zunächst lernen Sie aber noch eine Technik kennen, die beim Debuggen von Speicherzugriffsfehlern hilft.

Be gone, segmentation faults!

Im vorherigen Übungsblatt hatten Sie vermutlich bereits das Vergnügen sich mit Buffer Overflows und Segmentation Faults herumzuärgern.

Diese sind ein großes Problem bei den meisten Low-Level-Sprachen wie C und für einen signifikanten Teil an Sicherheitslücken verantwortlich.[1]

Der Grund hierfür ist, dass man bei Low-Level-Sprachen die Möglichkeit haben möchte, Programme so schreiben zu können, dass sie so performant wie möglich sind. Hierzu gehört insbesondere, dass man selbst entscheiden kann wie die Daten im Speicher abgelegt werden und ob Daten kopiert werden sollen oder man nur einen Zeiger auf die Daten übergibt. CPUs kennen nur Zahlen und erlauben es manche Zahlen als Speicheradressen zu benutzen – man kann bei C also sehr genau steuern wie das Programm ausgeführt wird.

Wenn man z.B. in Python versucht ein Array an einem ungültigen Index zu verändern, wird diese Änderung gar nicht erst ausgeführt und stattdessen das Programm mit einem Laufzeitfehler abgeschossen:

[1] Die Programmiersprache *Rust* ist hier eine nennenswerte Ausnahme. Das Typsystem von Rust ist, trotz der manuellen Speicherverwaltung, stark genug um sicherzustellen, dass Rust-Programme nur dann kompilieren, wenn garantiert keine Speicherzugriffsfehler auftreten - und zwar egal mit welchen Benutzereingaben man das Programm aufruft. Die Technik die wir hier für C kennenlernen, findet Fehler lediglich während der Ausführung des Programms mit einer bestimmten Benutzereingabe. Dies schließt es aber nicht aus, dass das Programm bei anderen Eingaben den Speicher verletzt.

```
>>> xs = [1,2,3]
>>> xs[3925] = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

Dies ist aber nur möglich, da das Python Programm vor jedem Array-Zugriff Extracode ausführt, der überprüft ob der Array-Zugriff überhaupt zulässig ist. Dies geht aber auf Kosten der Performance und bei manchen Anwendungen will man sich solche Extrakosten aus verschiedenen Gründen nicht leisten:

- Im Bereich des Maschinellen Lernens kann das Trainieren von Neuronale Netzen gerne mal mehrere Tage dauern. Wenn man hier statt mehreren Tagen nur ein paar Stunden auf das Ergebnis warten muss, dann ist das eine sehr willkommene Verbesserung.
- Wenn man ein 3D-Computerspiel programmiert, dann macht ein Faktor 2 in Performance den Unterschied zwischen 30 oder 60 FPS auf gängiger Hardware.
- Wenn man, im Bereich der Eingebetteten Systeme, das Programm für eine Thermostat-Steuerung schreibt, welches dann auf einem Mikrocontroller laufen muss der gerade mal 1 Cent kostet und dementsprechend sehr langsam ist und nur ein paar Kilobyte an Arbeitsspeicher hat, dann muss man mit seinen Resourcen gut haushalten. Desweiteren bedeutet doppelt so schneller Code hier auch, dass man nur halb so viel Strom verbraucht und die Batterie länger hält.
- Wenn man bei einer der großen Internetfirmen an der Serversoftware arbeitet und der Code doppelt so schnell läuft, dann können Millionen Euro an Stromkosten gespart werden, da die Serverfarm halbiert werden kann.

Um sich bei C trotz der scharfen Waffen der manuellen Speicherverwaltung das Leben etwas einfacher zu machen, gibt es sogenannte *Address Sanitizer*. Diese prüfen ob während der Ausführung des Programms auf ungültigen Speicher zugegriffen wird und bringen dann das Programm mit einer aussagekräftigen Fehlermeldung zum Absturz.

Da dieses Überprüfen auch wieder Extrakosten verursacht, verwendet man die Address Sanitizer im Regelfall nur während der Entwicklung und dem Testen. Wenn das Programm dann am Ende fertig ist kann man die Address Sanitizer einfach wieder weglassen zu Gunsten der Performance.

In den Übungen sollen Sie ab jetzt auch Address Sanitizer verwenden. Die Compiler `gcc` und `clang` enthalten bereits einen Address Sanitizer den man beim Kompilieren und Linken über Kommandozeilenargumente einschalten kann.

Angenomen wir haben in der `evil.c` bösen Code geschrieben:

```
int main(void) {
  int xs[5];
  xs[23] = 42;  /* Warning: this line may spontaneously combust! */
  return 0;    }
```

Wenn wir das Programm ausführen, dann stürzt es trotz des Fehlers häufig nicht mal ab, sondern es wird stillschweigend irgendwo Speicher überschrieben, egal ob an der Stelle `xs[23]` freier Speicher liegt oder irgendwelche Daten aus anderen Variablen. Wenn wir Glück haben und die Adresse von `xs[23]` so falsch ist, dass sie außerhalb des Speicherbereichs unseres Prozesses liegt, bekommen wir wenigstens noch einen `segmentation fault`, der aber auch nicht besonders hilfreich ist.

Wenn wir beim Kompilieren und Linken aber die Kommandozeilenargumente `-fsanitize-address` und `-g` angeben

```
$ gcc -fsanitize=address -g -c evil.c -o evil.o
$ gcc -fsanitize=address -g evil.o -o evil
```

dann wird uns beim Ausführen des Programms liebevoll mitgeteilt was wir alles falsch gemacht haben:

```
$ ./evil
=================================================================
==16907==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffced7b9908 at pc 0x000000401156 bp 0x7ffced7b98b0 sp 0x7ffced7b98a8
WRITE of size 4 at 0x7ffced7b9908 thread T0
    #0 0x401155 in main /home/m0rphism/Work/phd/teaching/lecture-cpp/2021-SS/svn/exercises/sheet04/evil.c:3
    #1 0x7f52b6411cbc in __libc_start_main (/nix/store/0c7c96gikmzv87i7lv3vq5s1cmfjd6zf-glibc-2.31-74/lib/libc.so.6+0x23cbc)
    #2 0x4011e9 in _start (/media/drive4a/progrep/teaching/advanced-programming/2021/exercises/sheet04/evil+0x4011e9)

Address 0x7ffced7b9908 is located in stack of thread T0 at offset 72 in frame
    #0 0x40108f in main /home/m0rphism/Work/phd/teaching/lecture-cpp/2021-SS/svn/exercises/sheet04/evil.c:1

  This frame has 1 object(s):
    [32, 52) 'xs' (line 2) <== Memory access at offset 72 overflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind mechanism, swapcontext or vfork
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow /home/m0rphism/Work/phd/teaching/lecture-cpp/2021-SS/svn/exercises/sheet04/evil.c:3 in main
Shadow bytes around the buggy address:
  0x10001daef2d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10001daef2e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10001daef2f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10001daef300: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10001daef310: 00 00 00 00 00 00 00 00 f1 f1 f1 f1 00 00 04 f3
=>0x10001daef320: f3[f3]f3 f3 00 00 00 00 00 00 00 00 00 00 00 00
  0x10001daef330: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10001daef340: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10001daef350: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10001daef360: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x10001daef370: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
  Shadow gap:              cc
==16907==ABORTING
```

Die interessanten Informationen sind dabei bereits in den ersten Zeilen zu finden:

- Es handelt sich um einen `stack-buffer-overflow` - also Speicher der nicht mit `malloc` angelegt wurde.
- Aus der Stacktrace geht hervor, dass der Fehler in der `main`-Funktion passiert ist und zwar an der Stelle `evil.c:3` - also Zeile 3 in der `evil.c`.

Der Address-Sanitizer kann ebenfalls fehlerhafte Speicherzugriffe bei dynamisch alloziertem Speicher erkennen (`malloc`) und sogenannte memory leaks - also wenn man vergisst dynamisch allozierten Speicher wieder mit `free` freizugeben. Ich empfehle die anderen Fehlerfälle an zwei weiteren Dateien `doom.c` und `total_damnation.c` auszuprobieren um sich an die Fehlermeldungen zu gewöhnen.

Von nun an soll der Address Sanitizer Ihr persönlicher Schutzpatron sein und Sie auf

all Ihren Ausführungspfaden begleiten. Vergessen Sie daher nicht ihn bei *allen* Ihrer `gcc`-Aufrufe durch die beiden Kommandozeilenargumente zu beschwören.

Rückblick & Motivation

In der Vorlesung wurde der Datentyp `intarray` implementiert, der eine Funktion `ia_write` bereitstellt, die es erlaubt dem Array Zahlen an einem bestimmten Index hinzuzufügen. Ist das Array nicht groß genug um eine Zahl an diesem Index abzulegen, so wird der Speicher automatisch vergrößert:

```
intarray* ia = ia_new(1, 0); /* Create an array of size 1 and default element 0. */
ia_write(ia, 0, 23);         /* Write value 23 to index 0. */
ia_write(ia, 5, 42);         /* Write value 42 to index 5. */
```

Beim ersten Aufruf von `ia_write` ist der Speicherbereich groß genug, deshalb wird lediglich der Wert 23 eingetragen.

Beim zweiten Aufruf von `ia_write` liegt der Index 5 außerhalb des Speicherbereichs, deshalb wird

- ein neuer Speicherbereich angelegt, der groß genug ist um 6 `int`s zu speichern;
- die bereits existierenden Elemente (`23` an Index 0) vom alten Speicherbereich in den neuen Speicherbereich kopiert;
- der alte Speicherbereich freigegeben;
- das neue Element `42` an Index 5 eingetragen; und
- das Default-Element `0` an Index 1 bis 4 eingetragen.

Das Problem bei dieser Implementierung ist, dass sie nicht sehr effizient ist, wenn man das Array Stück für Stück erweitern muss und nicht bereits im Voraus weiß wie groß das Array am Ende sein wird. Dies tritt z.B. auf wenn wir solange Zahlen vom Terminal einlesen und in einem Array speichern möchten bis ein Text eingegeben wird der keine Zahl darstellt:

```
intarray* ia = ia_new(0, 0);
int user_input;
size_t i = 0;
while (read_int(&user_input)) {
  ia_write(ia, i++, user_input);
}
```

Die Funktion `bool read_int(int* i)` ist dabei eine fiktive Funktion, die auf eine Benutzereingabe wartet und wenn eine Zahl eingegeben wurde, die Zahl in `i` speichert und `true` zurückgibt, und für alle anderen Eingaben `false` zurückgibt.

Für jede neue Zahl die eingegeben wird ist der bestehende Speicherbereich von `ia` zu klein und es müssen alle alten Zahlen und die neue Zahl in einen neuen Speicherbereich kopiert werden. Wenn n Zahlen eingegeben werden, so werden also n Speicherbereiche angelegt und $\sum_{i=1}^{n} i$ Zahlen kopiert.

Die Datenstrukturen für dynamische Arrays verwenden daher meistens eine effizientere Implementierung, die Sie in der folgenden Aufgabe selbst implementieren sollen. Diese Implementierung wird übrigens auch in `std::vector` von C++, `std::Vec` von Rust und Python's Listen verwendet. Das Nachimplementieren sollte also auch dabei helfen die Performance-Charakteristiken von dynamischen Arrays im Allgemeinen besser zu verstehen.

Aufgabe 5.1 (Resizable Arrays; 10 Punkte)

In dieser Aufgabe sollen Sie eine effizientere Version des `intarray`'s implementieren.

Die *Kernidee* ist dabei wie folgt: wenn eine neue Zahl am Ende des Arrays angehängt werden soll und der Speicherbereich des Arrays bereits voll ausgenutzt wird, dann vergrößert man den Speicherbereich nicht nur um Platz für eine weitere Zahl zu schaffen, sondern man verdoppelt die Größe des Speicherbereichs. Wenn das Array also zuvor n Zahlen enthalten hat, dann ist der Speicherbereich danach groß genug um $2 \cdot n$ Zahlen fassen zu können und erst nach dem Einfügen von weiteren n Zahlen muss wieder vergrößert und kopiert werden. Im Schnitt müssen also für das Einfügen einer Zahl nur noch 2 Zahlen kopiert werden, statt wie zuvor alle bisher im Array enthaltenen Zahlen.[2]

Das zugehörige `struct` sieht dabei wie folgt aus:

```
struct Vec {
  int* data;       /* dynamic memory area containing the integers */
  size_t length;   /* how many integers are currently stored in data */
  size_t capacity; /* how many integers can be stored in data */
};
```

Im Gegensatz zum `intarray` aus der Vorlesung hat unser `Vec` kein Feld für ein Default-Element, da wir bei unseren Array-Funktionen immer explizit die Zahlen angeben die eingefügt werden sollen.

In Ihrem Repository finden Sie im Verzeichnis `blatt05` die Dateien `vec.c` und `vec.h`.[3] In der `vec.h` sind bereits die Funktionen deklariert und dokumentiert, die unser Array unterstützen soll.

Schreiben Sie für jede Funktion, die in `vec.h` deklariert ist, eine zugehörige Definition in `vec.c`. Überprüfen Sie Ihre Definitionen mit sinnvollen Unittests in `vec_test.c`.

Neben der Dokumentation in der `vec.h` gibt es dabei folgendes zu beachten:

- Die Funktion `vec_print` soll für den Code

[2] Man sagt hier auch, dass der optimierte Algorithmus zum Einfügen eines weiteren Elements eine "amortisiert konstante Laufzeit" hat. Eine einzelne Einfügeoperation, kann zwar lineare Laufzeit haben, da alle bisherigen n Elemente kopiert werden müssen, aber dies gleicht sich über die nächsten n Einfügeoperationen wieder aus, die in konstanter Zeit stattfinden.

[3] Vergessen Sie nicht `git pull --rebase` auszuführen, sodass die Dateien auch in Ihrer Arbeitskopie ankommen.

```
Vec* xs = vec_new();
vec_push(xs, 10);
vec_push(xs, 20);
vec_push(xs, 30);
vec_print(xs);
vec_free(xs);
```

die folgende Ausgabe erzeugen:[4]

```
Vector at address 0x603000000010 has 3 elements and capacity 4.
  vec[0] = 10 (address 0x602000000050)
  vec[1] = 20 (address 0x602000000054)
  vec[2] = 30 (address 0x602000000058)
```

– Die Funktion `vec_sort` soll die Zahlen des Arrays in aufsteigender Reihenfolge sortieren, d.h. nach dem Sortieren eines Arrays `xs` gilt für alle `i < j < vec_length(xs)`, dass `*vec_at(xs, i) <= *vec_at(xs, j)`.

  Verwenden Sie hierzu folgenden Sortieralgorithmus: Suchen Sie einen Zeiger auf die kleinste Zahl im Array und vertauschen Sie diese mit der ersten Zahl im Array. Da die kleinste Zahl am Anfang steht muss danach nur noch der Rest des Arrays sortiert werden. Wiederholen Sie diesen Vorgang so lange bis das gesamte Array sortiert ist.

  Hinweis. Es bietet sich an die Funktion `vec_min_between` zu verwenden und eine Hilfsfunktion `void swap_int(int* x, int* y)` zu definieren, die die Werte hinter zwei Pointern vertauscht, z.B.

```
int x = 5;
int y = 3;
swap_int(&x, &y);
assert(x == 3 && y == 5);
```

– Die Ausgabe zu folgendem Beispiel ist in Abbildung 1 zu finden und demonstriert wie sich bei wiederholtem Benutzen von `vec_push` und `vec_pop` die Speicherbereiche vergrößern und wieder verkleinern.

```
Vec* xs = vec_new();
vec_print(xs);
/* Append 9 numbers to our vector. */
for (size_t i = 0; i < 9; ++i) {
  vec_push(xs, i * 10);
  vec_print(xs);
}
/* Remove the 9 numbers again. */
for (size_t i = 0; i < 9; ++i) {
  vec_pop(xs);
  vec_print(xs);
}
vec_free(xs);
```

[4] Die genauen Adressen sind dabei natürlich bei jedem Aufruf des Programms anders, da diese davon abhängen an welchen Stellen `malloc` gerade freien Speicher findet.

```
Vector at address 0x603000000010 has 0 elements and capacity 1.

Vector at address 0x603000000010 has 1 elements and capacity 1.
  vec[0] = 0 (address 0x602000000010)

Vector at address 0x603000000010 has 2 elements and capacity 2.
  vec[0] = 0 (address 0x602000000030)
  vec[1] = 10 (address 0x602000000034)

Vector at address 0x603000000010 has 3 elements and capacity 4.
  vec[0] = 0 (address 0x602000000050)
  vec[1] = 10 (address 0x602000000054)
  vec[2] = 20 (address 0x602000000058)

Vector at address 0x603000000010 has 4 elements and capacity 4.
  vec[0] = 0 (address 0x602000000050)
  vec[1] = 10 (address 0x602000000054)
  vec[2] = 20 (address 0x602000000058)
  vec[3] = 30 (address 0x60200000005c)

Vector at address 0x603000000010 has 5 elements and capacity 8.
  vec[0] = 0 (address 0x603000000040)
  vec[1] = 10 (address 0x603000000044)
  vec[2] = 20 (address 0x603000000048)
  vec[3] = 30 (address 0x60300000004c)
  vec[4] = 40 (address 0x603000000050)

Vector at address 0x603000000010 has 6 elements and capacity 8.
  vec[0] = 0 (address 0x603000000040)
  vec[1] = 10 (address 0x603000000044)
  vec[2] = 20 (address 0x603000000048)
  vec[3] = 30 (address 0x60300000004c)
  vec[4] = 40 (address 0x603000000050)
  vec[5] = 50 (address 0x603000000054)

Vector at address 0x603000000010 has 7 elements and capacity 8.
  vec[0] = 0 (address 0x603000000040)
  vec[1] = 10 (address 0x603000000044)
  vec[2] = 20 (address 0x603000000048)
  vec[3] = 30 (address 0x60300000004c)
  vec[4] = 40 (address 0x603000000050)
  vec[5] = 50 (address 0x603000000054)
  vec[6] = 60 (address 0x603000000058)

Vector at address 0x603000000010 has 8 elements and capacity 8.
  vec[0] = 0 (address 0x603000000040)
  vec[1] = 10 (address 0x603000000044)
  vec[2] = 20 (address 0x603000000048)
  vec[3] = 30 (address 0x60300000004c)
  vec[4] = 40 (address 0x603000000050)
  vec[5] = 50 (address 0x603000000054)
  vec[6] = 60 (address 0x603000000058)
  vec[7] = 70 (address 0x60300000005c)

Vector at address 0x603000000010 has 9 elements and capacity 16.
  vec[0] = 0 (address 0x606000000020)
  vec[1] = 10 (address 0x606000000024)
  vec[2] = 20 (address 0x606000000028)
  vec[3] = 30 (address 0x60600000002c)
  vec[4] = 40 (address 0x606000000030)
  vec[5] = 50 (address 0x606000000034)
  vec[6] = 60 (address 0x606000000038)
  vec[7] = 70 (address 0x60600000003c)
  vec[8] = 80 (address 0x606000000040)

Vector at address 0x603000000010 has 8 elements and capacity 8.
  vec[0] = 0 (address 0x603000000070)
  vec[1] = 10 (address 0x603000000074)
  vec[2] = 20 (address 0x603000000078)
  vec[3] = 30 (address 0x60300000007c)
  vec[4] = 40 (address 0x603000000080)
  vec[5] = 50 (address 0x603000000084)
  vec[6] = 60 (address 0x603000000088)
  vec[7] = 70 (address 0x60300000008c)

Vector at address 0x603000000010 has 7 elements and capacity 8.
  vec[0] = 0 (address 0x603000000070)
  vec[1] = 10 (address 0x603000000074)
  vec[2] = 20 (address 0x603000000078)
  vec[3] = 30 (address 0x60300000007c)
  vec[4] = 40 (address 0x603000000080)
  vec[5] = 50 (address 0x603000000084)
  vec[6] = 60 (address 0x603000000088)

Vector at address 0x603000000010 has 6 elements and capacity 8.
  vec[0] = 0 (address 0x603000000070)
  vec[1] = 10 (address 0x603000000074)
  vec[2] = 20 (address 0x603000000078)
  vec[3] = 30 (address 0x60300000007c)
  vec[4] = 40 (address 0x603000000080)
  vec[5] = 50 (address 0x603000000084)

Vector at address 0x603000000010 has 5 elements and capacity 8.
  vec[0] = 0 (address 0x603000000070)
  vec[1] = 10 (address 0x603000000074)
  vec[2] = 20 (address 0x603000000078)
  vec[3] = 30 (address 0x60300000007c)
  vec[4] = 40 (address 0x603000000080)

Vector at address 0x603000000010 has 4 elements and capacity 4.
  vec[0] = 0 (address 0x602000000070)
  vec[1] = 10 (address 0x602000000074)
  vec[2] = 20 (address 0x602000000078)
  vec[3] = 30 (address 0x60200000007c)

Vector at address 0x603000000010 has 3 elements and capacity 4.
  vec[0] = 0 (address 0x602000000070)
  vec[1] = 10 (address 0x602000000074)
  vec[2] = 20 (address 0x602000000078)

Vector at address 0x603000000010 has 2 elements and capacity 2.
  vec[0] = 0 (address 0x602000000090)
  vec[1] = 10 (address 0x602000000094)

Vector at address 0x603000000010 has 1 elements and capacity 1.
  vec[0] = 0 (address 0x6020000000b0)

Vector at address 0x603000000010 has 0 elements and capacity 1.
```

Abbildung 1: Ausgabe des Beispiels zu `vec_push` und `vec_pop`

Statistics 101

In der Empirischen Forschung geht es um die systematische Auswertung eines Experiments mit statistischen Methoden. Bei der Durchführung eines Experiments misst man bestimmte Daten, die man danach interpretieren möchte um Schlüsse aus dem Experiment zu ziehen. Zwei bekannte Methoden zur statistischen Auswertung sind dabei der Durchschnitt und die Mittlere Absolute Abweichung – also die durchschnittliche Abweichung vom Durchschnitt.

Der Durchschnitt hat aber das Problem, dass er sensibel auf sogenannte *Outlier* reagiert. Wird zum Beispiel fälschlicherweiße ein riesiger Wert gemessen oder gibt eine einzelne Person bei einer Umfrage zum Zeitbedarf der Übungsblätter einen massiv falschen Wert an, so verzerrt dies stark den Durchschnitt.

Bestehen die gemessenen Datenpunkte zum Beispiel aus den Zahlen

$$5,\ 3,\ -4,\ 8,\ 3,\ 1000000,\ 2,\ 7$$

so ist der Durchschnitt 125003 und die Standardabweichung 218749.25.

Eine Alternative zum Durchschnitt, die resistenter gegen Outlier ist, sind die sogenannten *Quantile*. Hier sortiert man zunächst die Datenpunkte in aufsteigender Reihenfolge und betrachtet dann Datenpunkte an bestimmten Stellen.

Haben wir n Datenpunkte gemessen und sortiert, so nennt man den Datenpunkt mit Index $\lfloor \frac{n}{4} \rfloor$ das untere Quartil; $\lfloor \frac{n}{2} \rfloor$ den Median; und $\lfloor \frac{3n}{4} \rfloor$ das obere Quartil.

Der Median erfüllt eine ähnliche Rolle wie der Durchschnitt: die Hälfte der Datenpunkte ist kleiner wie der Median und die andere Hälfte ist größer.

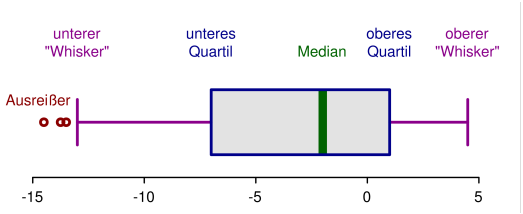
Die Quartile erfüllen eine ähnliche Rolle wie die Mittlere Absolute Abweichung: die Hälfte der Datenpunkte liegt zwischen dem unteren Quartil und dem oberen Quartil.

Sortieren wir die 8 Datenpunkte in unserem Beispiel so erhalten wir

$$-4,\ 2,\ \mathbf{3},\ 3,\ \mathbf{5},\ 7,\ \mathbf{8},\ 1000000$$

Das untere Quartil ist hier die Zahl mit Index 2 – also 3; der Median ist hier die Zahl mit Index 4 – also 5; und das obere Quartil ist hier die Zahl mit Index 6 – also 8.

Die Quantile werden in wissenschaftlichen Dokumenten auch häufig durch sogenannte *Boxplots* dargestellt:



Wie bereits angedeutet, finden Sie die Quartile und den Median auch jede Woche auf den Vorlesungsfolien zum Zeitbedarf der Übungsblätter.

Aufgabe 5.2 (Statistik; 4 Punkte)

Schreiben Sie ein Programm in `statistics.c`, welches beliebig viele Zahlen vom Terminal einliest und eine statistische Auswertung der Zahlen ausgibt.

Beispiel (Benutzereingaben in blau):

```
$ ./statistics
1
5
10
-13
8
-25
42
99999999
3
9
Number of data points: 10
Minimum:               -25
Lower Quartile:        1
Median:                8
Upper Quartile:        10
Maximum:               99999999
Average:               10000004.000000
```

Ungültige Eingaben sollen dabei ignoriert werden. Ist mindestens eine ungültige Eingabe vorhanden, so soll in der Ausgabe zusätzlich vermerkt werden wie viele ungültige Eingaben es insgesamt gegeben hat. Beispiel (Benutzereingaben in blau):

```
$ ./statistics
1
I'm not a number!
5
8
I'm also not a number.
9
10
Number of data points: 5
Minimum:               1
Lower Quartile:        5
Median:                8
Upper Quartile:        9
Maximum:               10
Average:               6.600000

WARNING: Found 2 invalid data points.
```

Wenn im Terminal die Tastenkombination `STRG+D` gedrückt wurde, so soll dies dem Programm signalisieren, dass die Eingabe der Zahlen beendet ist und nun die Statistik berechnet werden kann. In den Beispielen wurde also nach der letzten blauen Zeile `STRG+D` gedrückt.

Um eine Zahl vom Terminal einzulesen, können Sie die folgende Funktion verwenden, die bereits in der `statistics.c` in Ihrem Repository vorhanden ist:

```
#define SUCCESS 0
#define ERROR_NO_FURTHER_INPUT -1
#define ERROR_STRING_IS_NOT_A_NUMBER -2

int read_int(int *user_input) {
  char *line = NULL;
  size_t line_len = 0;
  ssize_t status = getline(&line, &line_len, stdin);
  if (status < 0) {
    free(line);
    return ERROR_NO_FURTHER_INPUT;
  }
  errno = 0;
  char *end = NULL;
  int i = strtol(line, &end, 10);
  free(line);
  if (errno != 0 || end == line) {
    return ERROR_STRING_IS_NOT_A_NUMBER;
  } else {
    *user_input = i;
    return SUCCESS;
  }
}
```

Die Funktion versucht eine Benutzereingabe vom Terminal zu lesen und diese in einen `int` zu konvertieren:

- Ist dies erfolgreich, so wird die Zahl in `user_input` geschrieben und `SUCCESS` zurückgegeben.
- Stellt die Benutzereingabe keine Zahl dar, so bleibt `user_input` unverändert und es wird `ERROR_STRING_IS_NOT_A_NUMBER` zurückgegeben.
- Ist es nicht möglich eine Benutzereingabe zu lesen, so bleibt `user_input` unverändert und es wird `ERROR_NO_FURTHER_INPUT` zurückgegeben.

Der letzte Fall tritt auf, wenn im Terminal die Tastenkombination `STRG+D` gedrückt wurde. Die Tastenkombination bewirkt, dass die Eingabedatei `stdin` des Terminals geschlossen wird. Dies signalisiert dem Programm, dass keine weiteren Benutzereingaben folgen werden.

Dieser Mechanismus wird z.B. auch von dem Unix-Programm `sort` verwendet, welches Sie bereits im Terminal-Tutorial des ersten Übungsblattes kennengelernt haben.

Der Hintergrund ist dabei, dass man beim Ausführen eines Programms im Terminal die Benutzereingaben auch durch eine echte Datei oder durch die Ausgaben eines anderen Programms ersetzen kann. In diesem Fall ist die Datei irgendwann zu Ende und das Programm muss damit umgehen können, dass es keine weiteren Eingaben mehr geben wird.

Angenommen wir haben die Datenpunkte, die wir im ersten Beispiel von Hand eingegeben haben, in einer Datei `datapoints.txt` gespeichert, so können wir unser `statistics`-Programm also auch wie folgt aufrufen und anstatt die Benutzereingabe von Hand einzugeben den Inhalt der `datapoints.txt` verwenden:

```
$ ./statistics < datapoints.txt
Number of data points: 10
Minimum:               -25
Lower Quartile:        1
Median:                8
Upper Quartile:        10
Maximum:               99999999
Average:               10000004.000000
```

Aufgabe 5.3 (Erfahrungen; 2 Punkte)

Notieren Sie Ihre Erfahrungen mit diesem Übungsblatt in der Datei `erfahrungen.txt` (benötigter Zeitaufwand, Probleme, Bezug zur Vorlesung, Interessantes, etc.).

Der Zeitaufwand *muss* dabei in der ersten Zeile und in exakt dem folgenden Format notiert werden, da wir sonst nicht automatisiert eine Statistik erheben können:

```
Zeitaufwand: 3:30

<...Andere Erfahrungen...>
```

Die Angabe `3:30` steht hier für 3 Stunden und 30 Minuten.