PROGRAMMIEREN MIT C

Allgemeine hinweise

Alles was hier beschrieben wird, soll auch ausprobiert werden. Warum C? Weil die coolen Dinge mit C am einfachsten gehen. Das werden wir in den folgenden Übungen noch sehen.

1. C-Code kompilieren

Ich empfehle die Anleitung http://c.learncodethehardway.org/book/index.html im Internet als begleitende Literatur für die Programmieraufgaben. Wenn etwas zum Programmieren mit C unklar ist, findet man in der Regel im Internet sehr schnell eine Antwort. Für alle Programmieraufgaben empfehle ich auch wärmstens Linux als Betriebssystem und gcc als Compiler zu verwenden. Als Editor kann man zum Beispiel gedit verwenden.

Schreibe mit C ein Programm, welches "Hallo Welt" in die Konsole schreibt, kompiliere es und führe es aus.

Das einfachste C-Programm sieht folgendermaßen aus:

```
main(int argc, char** argv){

}
```

, wird kompiliert mit dem Befehl

```
gcc -o programm programmcode.c
```

(-o steht für output) und ausgeführt mit dem Befehl

```
./programm
```

Mit

```
printf("Hallo Welt\n");
```

kann man Dinge auf die Konsole schreiben, muss aber die entsprechende Bibliothek einbinden, indem man am Anfang des Programmcodes

```
#include <stdio.h>
```

angibt.

Um den Befehl zum Kompilieren nicht jedes Mal neu eingeben zu müssen kann man ihn einfach in eine Datei (die am Besten immer "compile" heißt) schreiben, die man dann mit Hilfe des Befehls

```
chmod +x
```

ausführbar macht und wie ein normales Programm mit

```
./compile
```

startet. Man kann in dieser Datei auch mehrere Befehle hintereinander ausführen indem man sie getrennt durch " && " hintereinander schreibt. Es empfiehlt sich, auch eine Datei "clean" anzulegen, welche die durch "compile" erzeugten Dateien wieder löscht.

2. Funktionen, Standarddatentypen

Schreibe zwei Funktionen, die, x und eine boolsche Variable b verlangen und $\begin{cases} |x^3| & , falls\, b\, wahr \\ x^3 & , falls\, b\, falsch \end{cases}$ berechnen und zurückgeben, eine für int und eine für double. Um

```
bool
```

zu benutzen muss

```
#include <stdbool.h>
```

geladen werden. Den Inhalt von den Dateien *.h findet man schnell im Internet.

Pointer

In C gibt es im Prinzip keine Arrays, aber man kann auf dem heap (Arbeitsspeicherbereich, in dem nicht das Programm und nicht die lokalen Variablen liegen) n Standarddatentypen hintereinander abspeichern und diese dann mittels ihrer Speicheradressen abrufen. Ein sogenannter *Pointer* beinhaltet eine Speicheradresse und kennt auch die Größe des bei dieser Adresse abgespeicherten Datentyps. Man *definiert* einen solchen Pointer mit dem Befehl

```
double* meinNeuerPointer;
```

Bevor wir eine Pointervariable verwenden können, müssen wir ihr eine Adresse zuweisen, an der wir auch Speicher reserviert haben. Speicher kann man mit

```
void* malloc(int i)
```

, speziell also

```
void* malloc(sizeof(Datentyp)*Arraylaenge)
```

bestellen. "malloc" ist im Paket "stdlib.h" (am Anfang unseres Quellcodes müssen wir jetzt immer #include <stdlib.h> schreiben) und hat als Rückgabetyp einen void-Pointer (void*), der weiß, wo der bestellte Speicher sitzt, aber nicht, wie groß die darin enthaltenen Datentypen sind (zum Beispiel: double und long haben 64bit, char und byte haben 8 bit). Man kann aber ohne weiteres den void-Pointer, den man von malloc bekommt zu beliebigem Pointer-typ *casten*. Das geht zum Beispiel so

```
double* meinNeuerPointer;
meinNeuerPointer = malloc(sizeof(double));
```

malloc schreibt nichts in den Speicher, das heißt man kann auf den bestellten Speicher zugreifen, aber er besitzt am Anfang keine definierten Werte sondern recht zufällige Daten, die an der Stelle vorher im Speicher waren. Wenn man den reservierten Speicherbereich mit Nullen gefüllt haben will, dann kann man statt "malloc" auch "calloc" verwenden.

Jetzt wo wir den Pointer auch initialisiert haben, können wir zum Test seine Speicheradresse auslesen.

```
printf("Pointeradresse ist %p\n", meinNeuerPointer);
```

(%p steht für pointer) gibt die Speicheradresse des Pointers in Dezimaldarstellung aus.

Mit dem Stern auf der linken Seite bekommt man den Inhalt einer Speicheradresse und kann ihn auslesen beziehungsweise schreiben. Zum Beispiel so

```
double x = *meinNeuerPointer;
```

oder so

```
*meinNeuerPointer = 5.5;
```

Man nennt das *Dereferenzieren* des Pointers.

Sobald man den Speicher auf den ein Pointer zeigt nicht mehr verwendet, sollte man immer

```
free(meinNeuerPointer);
```

aufrufen, um den Speicher wieder freizugeben.

Was kann man jetzt mit diesen Pointern machen? Man kann sie zum Beispiel als Argument an eine Funktion übergeben, die Funktion ändert den Speicher auf den der Pointer zeigt, und nachdem die Funktion beendet ist, kann man den Speicher auslesen.

Ab jetzt gehört es zu jeder Programmieraufgabe mit

```
valgrind ./programm
```

zu überprüfen, ob jeglicher mit "malloc" bestellter Speicher ordnungsgemäß mit "free" wieder abgegeben wurde. valgrind beschwert sich zum Beispiel, wenn man einen Pointer benutzt ohne ihn initialisiert zu haben oder nachdem man ihn ge*free*t hat oder wenn man vergisst am Programmende alle verwendeten Pointer zu *free*n. Es sollte immer

```
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

am Ende erscheinen und am Besten auch keine Warnungen geben.

Arrays

Um in C ein Array zu definieren, kann man einfach einen Pointer verwenden, und statt sizeof(Datentyp) einfach sizeof(Datentyp)*n an Speicherplatz reservieren. Also folgendermaßen:

```
int n = 3;
double* meinZweiterPointer = malloc(sizeof(double)*n);
```

Damit zeigt der Pointer jetzt auf die erste Adresse eines Speicherbereichs, der so lang ist, dass n Datentypen vom Typ double hineinpassen, hintereinander. Um den i-ten double aus diesem Array aufzurufen, kann man mittels des Additionsoperators "+" den Pointer i Stellen weiter schieben und dann mit dem * den double im Speicher anfordern, also folgendermaßen (Klammern sind wichtig):

```
*(meinZweiterPointer+1) = 5.5;
double x = *(meinZweiterPointer+1);
```

Eine Kurzschreibweise hierfür, die ich auch empfehle, ist

```
meinZweiterPointer[1] = 5.5;
double x = meinZweiterPointer[1];
```

Wenn ein Array nicht mehr verwendet wird, muss es genauso, wie alle anderen Pointer mit free freigegeben werden:

```
free(meinZweiterPointer);
```

Die Länge von einem solchen "Array" muss man sich separat abspeichern und zum Beispiel an eine Funktion als separates Argument übergeben. Zum Beispiel so

```
int meineFunktion(double* arrayPointer, int laenge)
```

Structs

Structs sind so etwas wie Objekte in Java. Man definiert zum Beispiel ein Struct mit Namen Vector wie folgt:

```
struct Vector{
        double* values;
        int height;
};
```

Benutzen kann man das struct dann so:

```
struct Vector* meinVector = malloc(sizeof(struct Vector));
(*meinVector).height = 3;
```

Da meinVector als Pointer auf ein Struct definiert ist, müssen wir erst dereferenzieren und dann können wir wie in Java mit dem .-Operator das Datenfeld auslesen.

Da man dies recht häufig tut, gibt es auch wieder eine alternative schreibweise. Äquivalent ist

```
meinVector->height = 3;
```

Wie auch bei allen anderen Pointern folgt am Ende

```
free(meinVector);
```

3. Eine Matrix-"Klasse"

Schreibe ein Struct namens Matrix, welches eine Matrix der Größe $width \times height$ darstellt und ein Struct namens Vector für einen Vektor der Höhe $height$.

Implementiere außerdem die Funktionen

```
struct Matrix* new_Matrix(int width, int height)
struct Vector* new_Vector(int height)
```

und

```
void delete_Matrix(struct Matrix* m)
void delete_Vector(struct Vector* v)
```

welche diese structs initialisieren und allen benötigten Speicher reservieren beziehungsweise den Speicher freigeben. Beachte dass man für jede dieser Funktionen jeweils zweimal malloc beziehungsweise free aufrufen muss.

Schreibe auch folgende Funktionen

```
struct Matrix* ones(int height, int width)
struct Matrix* eye(int size)
void printMatrix(struct Matrix* matrix)
void printVector(struct Vector* vector)
```

welche eine Matrix voll mit Einsen und eine Identitätsmatrix erstellen beziehungsweise eine Matrix oder einen Vektor in die Konsole ausgeben.

Implementiere auch die folgenden Funktionen:

```
struct Matrix* transpose(struct Matrix* m)
struct Matrix* multiplyMatrixMatrix(struct Matrix* m1, struct Matrix* m2)
struct Vector* multiplyMatrixVector(struct Matrix* m1, struct Vector* v)
```

SDL und OpenGL

Da wir später im Kurs SDL 2.0 (simple direct media layer) und OpenGL 1.3 (open graphics library) verwenden werden, kann man sich die Webseiten dieser Projekte schonmal ansehen. Für OpenGL sind die Kapitel 1, 2 und 3 aus http://www.glprogramming.com/red/ für uns relevant. Alle nötigen Funktionen werden aber auch während der Übung kurz erläutert.