Dr. Michael J. Winckler
IWR, Raum 302
INF 368
69120 Heidelberg
Michael.Winckler@iwr.uni-heidelberg.de
http://www.iwr.uni-heidelberg.de/teaching/Mathe-Star/

Arbeitsblatt 05 *Vertiefung: Character und Strings* (Version 1.10)

Theorie

Übungsziele: Die bisher von uns entwickelten Programme und Module basierten auf einfachen Datentypen. Je komplexer die Beispiele werden, desto wichtiger ist es jedoch, Datentypen zu entwickeln, die die komplette Information über die verwendeten Entitäten enthalten. Hierzu stellt C Befehle zur Generierung *komplexer Datentypen* bereit.

Die verwendete Syntax ist denkbar einfach:

```
struct maze_s {
  int width   = 0;
  int height  = 0;

  int **field = NULL;
};
```

legt beispielsweise einen Datentypen `struct maze_s` an, den wir für die folgenden Aufagben gut brauchen können.

Übungen

Übung 6.1: Komplexe Datentypen

Definiere durch

```
struct bruch_s {
  int zaehler = 0;
  int nenner  = 1;
};
```

einen komplexen Datentypen zum Bruchrechnen. Deklariere und definiere zu diesem Datentyp Variablen und Zeiger. Belege die auftretenden Komponenten und schreibe eine Routine, um Brüche ausgeben zu können (Zugriffsoperatoren: `a.zaehler` bzw. `a_ptr->zaehler`!).

Verwende die definierten Routinen, um dir ein kleines Hauptprogramm zum Test des komplexen Datentyps zu schreiben. Implementiere zusätzlich die Rechenregeln und das Kürzen. Wie verarbeitest du die Rückgabewerte?

Übung 6.2: Ein Datentyp für Irrgärten: `maze`

Wir verwenden den im Theorieteil beschriebenen Datentypen, um ein Programm zur Generierung von Irrgärten zu schreiben. Zunächst definieren wir uns (wie oben angegeben) diesen Datentypen.

Danach können wir eine Reihe von Hilfsroutinen erstellen, die jeweils auf diesem Datentypen arbeiten. Implementiere dazu folgende Routinen:

1. `void Initialisieren(struct maze_s *m, int w, int h)` belegt ein Datenelement vom erstellten Datentyp mit ausreichend Speicher, um einen `w` breiten und `h` hohen Irrgarten darin unterzubringen. Warum verwendet man hier die Übergabe auf Basis eines Pointers?

2. `void Zufallsraum(struct maze_s *m)` sorgt für einen freien Raum innerhalb des Irrgartens. Spätestens hier müssen wir uns Gedanken um die Kodierung des Irrgartens machen und zudem spezielle Designfragen (z.B.: "Wie wird die Grösse des zufälligen Raumes gewählt?") entscheiden.

3. Weitere mögliche Routinen könnten z.B. Gänge vorgegebener oder zufälliger Länge oder Breite anlegen. Hier ist es sinnvoll, jede Routine in einem eigenen Namensraum benennen, um das gemeinsame Testen aller Routinen zu erleichtern (z.B. `void MW_Zufallsgang(struct maze_s *m)`).

4. Eine Ausgaberoutine macht es dir leichter, Testhauptprogramme für das Projekt zu schreiben.

5. Algorithmisch fortgeschritten ist die Frage, wie man den Zusammenhang des Irrgartens testen bzw. herstellen kann.

Übung 6.3: Datentypen benennen

Die Verwendung von `struct bruch_s` in jeder Deklaration und Definition ist auf die Dauer ziemlich umständlich. C erlaubt es aber, mit dem Befehl `typedef` Alias-Namen für Datentypen zu verwenden. Die Syntax ist im allgemeinen Beispiel denkbar einfach.

```
typedef int ganzzahl;
```

Dieser `typedef` beispielsweise implementiert das Signalwort `ganzzahl` als Synonym für `int`. Mit Strukturen wird das Ganze nur auf den ersten Blick unübersichtlich - in Wirklichkeit wird die obige Syntax einfach konsequent fortgesetzt:

```
typedef struct bruch_s {
  int zaehler = 0;
  int nenner  = 1;
} bruch;
```

So kann man also der Struktur, die wir für das Bruchrechnen benutzt haben, einen einfach zu verwendenden Aliasnamen geben. Schreibe das Beispiel aus Aufgabe 6.1 so um, dass Typendeklarationen die Strukturangaben überdecken.

Übung 6.4: Wrap it all up: eine Bibliothek

An dieser Stelle ist es günstig, Werkzeugfunktionalität und Programm konsequent zu trennen:

Der komplexe Datentyp und seine darauf abgestimmten Funktionen verhalten sich wie ein vorgefertigter Werkzeugkasten. Sie sind eigentlich nur Tools, um Hauptprogramme zu realisieren, die sich der Fragestellung, zu der die Tools generiert wurden, widmen wollen.

Verlagere daher die Struktur- und Funktionsdeklarationen sowie die zugehörigen Definitionen in eine Bibliothek, die du aus einem Header- und einem Soucefile realisierst. Erstelle dann eine Reihe von Testprogrammen, die sich dieser To ls bedienen und teste deine Funktionen (und die deiner Mitschüler) ausführlich.