HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Electronics4you

Übungen mit Anleitung

Hamster-Simulator

Nicola Ramagnano

23. Oktober 2011

1 Informationen über den Hamster-Simulator

Der Hamster-Simulator ist ein Programm, mit dem Hamster-Programme erstellt, ausgeführt und getestet werden können. Das Programm wird an der Universität Oldenburg in Deutschland entwickelt. Die offizielle Webseite lautet

`http://www.java-hamster-modell.de/`

Im Rahmen des Electronics4you wurde dieses Simulator-Programm mit eigenen Übungsaufgaben ergänzt und wird zusammen mit dieser Anleitung an unseren Teilnehmern weitergegeben.

2 Installation des Hamster-Simulators

Der Hamster-Simulator ist ein in Java geschriebenes Programm. Um es ausführen zu können, muss auf deinem Rechner eine Java-Laufzeitumgebung (JRE) installiert sein:

`http://www.java.com/de/download/`

Das Simulator-Programm mit den speziellen Übungen findest du auf der Electronics4you-Webseite

`http://www.electronics4you.cc/`

Dort befindet sich im Bereich "Projekt Mikrocontroller" eine Datei namens "hamstersimulator-e4u.zip". Diese Datei musst du auf deinem Rechner laden und anschliessend entpacken. Die Datei enthält eine Reihe von Dateien und Ordner. Die wichtigsten sind:

- `hamstersimulator.bat`: Dient zum Starten des Hamster-Simulators unter Windows.
- Ordner `Programme`: Enthält alle Hamster-Programme zu den Electronics4you-Übungen.

Führe ein Doppelklick auf die Datei `hamstersimulator.bat` aus, um den Hamster-Simulator zu starten. Anschliessend öffnen sich zwei Fenster, die **Editor** und **Simulation** heissen.

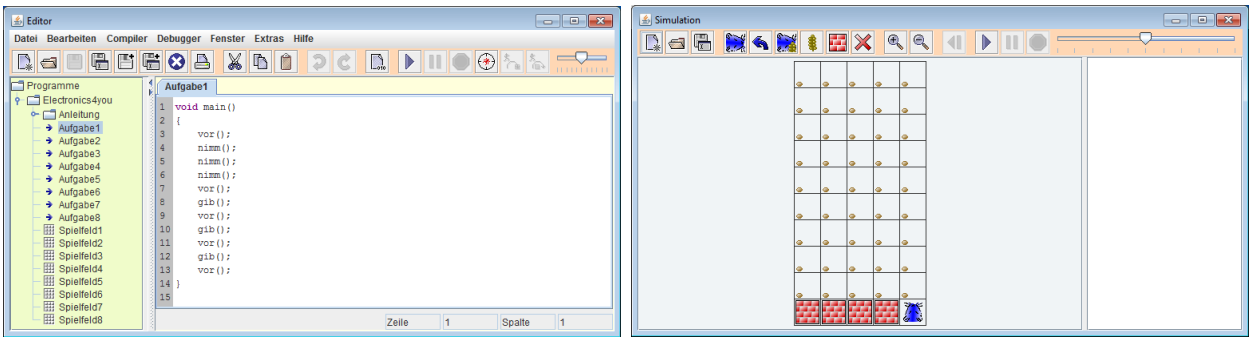


Abbildung 1: Editor- und Simulator-Fenster des Hamster-Simulators

3 Übungsaufgaben

Bei den folgenden Aufgaben geht es darum, Programme für den Hamster zu schreiben, damit dieser die vorgegebenen Ziele auf dem Spielfeld erreicht. Der Hamster versteht grundsätzlich nur sehr wenige Befehle (sog. Funktionen). Er kann folgende Anweisungen ausführen:

`vor();`	Bewegt den Hamster um ein Feld nach vorn. Diese Funktion darf nicht aufgerufen werden, falls der Weg versperrt ist, z.B. durch eine Wand oder den Spielfeldrand.
`linksUm();`	Dreht den Hamster an Ort um 90 Grad gegen den Uhrzeigersinn.
`nimm();`	Nimmt ein Korn vom aktuellen Feld auf. Diese Funktion darf nicht aufgerufen werden, falls sich keine Körner auf dem aktuellen Feld befinden.
`gib();`	Gibt ein Korn aus dem Maul des Hamsters ab. Diese Funktion darf nicht aufgerufen werden, falls sich keine Körner im Maul des Hamsters befinden.

Tabelle 1: Anweisungen, die der Hamster selber ausführen kann.

Wie man sieht, darf man die obigen Anweisungen nicht in jedem Fall aufrufen. Falls dies missachtet wird, z.B. wenn man die `nimm()`-Funktion auf einem leeren Feld aufruft, dann meldet der Simulator dies als Fehler und bricht die Programmausführung ab.

Aus diesem Grunde, kennt der Hamster drei Überprüfungen, die vor den obigen Anweisungen benutzt werden können um die aktuelle Situation abzuklären. Je nach Fall geben diese Funktionen entweder den Wert `wahr` oder `falsch` zurück:

`vornFrei()`	Überprüft ob das Feld vor dem Hamster frei ist.
`kornDa()`	Überprüft ob auf dem aktuellen Feld mindestens ein Korn liegt.
`maulLeer()`	Überprüft ob es im Maul des Hamsters mindestens ein Korn hat.

Tabelle 2: Überprüfungen, die der Hamster selber ausführen kann.

3.1 Aufgabe 1

In der ersten Aufgabe startet der Hamster auf einem Spielfeld mit 1×6 Feldern, wie dies in der Abbildung links gezeigt wird. Auf dem nächsten Feld vor dem Hamster liegen drei Körner, die zuerst aufgenommen und dann auf den nachfolgenden drei Felder verteilt werden sollen. Die Abbildung rechts zeigt wie das Spielfeld am Schluss aussehen soll.

(a) Ausgangslage (b) Ziel

Abbildung 2: Spielfeld der Aufgabe 1

➥ Ergänze das folgende Programm mit den passenden Befehlen aus Tabelle 1 und starte die Simulation, indem du auf den Startknopf drückst. Vergleiche am Schluss, ob dein Spielfeld dem obigen Ziel gleicht.

```
1  void main()
2  {
3      ... füge deine Befehle hier ein ...
4  }
```

Musterlösung

```
1  void main()
2  {
3      vor();        // gehe zum nächsten Feld
4
5      nimm();       // nimm die drei Körner auf
6      nimm();
7      nimm();
8
9      vor();        // gehe zum nächsten Feld
10     gib();        // gib ein Korn ab
11
12     vor();        // gehe zum nächsten Feld
13     gib();        // gib ein Korn ab
14
15     vor();        // gehe zum nächsten Feld
16     gib();        // gib ein Korn ab
17
18     vor();        // gehe zum letzten Feld
19 }
```

Listing 1: Musterlösung der Aufgabe 1

3.2 Aufgabe 2

Bei dieser Aufgabe geht es darum alle Körner auf dem Spielfeld einzusammeln, wie dies in den untenstehenden Abbildungen gezeigt wird.

(a) Ausgangslage (b) Ziel

Abbildung 3: Spielfeld der Aufgabe 2

Man könnte dies ähnlich wie in der vorherigen Aufgabe mit einer Kombination aus `vor()` und `nimm()` Anweisungen tun. Diesmal wollen wir aber nicht die `nimm()`-Funktion selber mehrmals schreiben müssen, um alle Körner auf einem Feld aufzunehmen. Das Programm soll "intelligent" genug sein und solange ein Korn auf einem Feld aufnehmen bis keine Körner auf diesem Feld sind. Somit dürfte es praktisch beliebig viele Körner auf einem einzelnen Feldern haben und das Programm würde trotzdem alle Körner "automatisch" einsammeln.

Zu diesem Zweck benutzen wir die Überprüfung `kornDa()`, um zu wissen ob wir nochmals die `nimm()`-Funktion auf dem gleichen Feld ausführen müssen. Zudem benötigen wir eine neue Anweisung, die uns erlaubt die `nimm()`-Funktion zu wiederholen.

Diese Anweisung heisst `while()` und ist eine sogenannte Wiederholungsschlaufe. Zwischen den runden Klammern wird die Bedingung für eine Wiederholung eingefügt, z.B. `kornDa()`. Falls die Bedingung `wahr` ist, werden alle Anweisungen, die zwischen den geschweiften Klammern stehen, wiederholt.

Eine mögliche Lösung wäre also:

```
1  void main()
2  {
3      vor();                // gehe zum nächsten Feld
4      while(kornDa()) {     // wiederhole solange Körner da sind
5          nimm();           // nimm ein Korn auf
6      }
7
8      vor();                // gehe zum nächsten Feld
9      while(kornDa()) {     // wiederhole solange Körner da sind
10         nimm();           // nimm ein Korn auf
11     }
12
13     ...
14 }
```

➥ Ändere nun den Lösungsvorschlag so, dass auch die Länge des Spielfelds beliebig gross sein kann, d.h das Programm soll also am Ende des Spielfelds "automatisch" stoppen.
Tipp: Benutze dafür die Überprüfung `vornFrei()`.

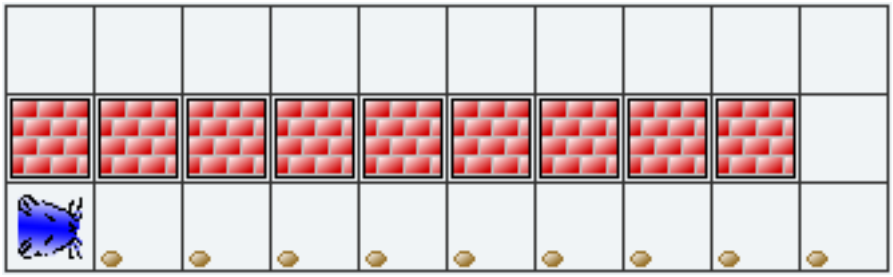
Musterlösung

```
void main()
{
    while(vornFrei()) {      // solange vorne frei ist
        vor();               // gehe zum nächsten Feld
        while(kornDa()) {    // solange Körner da sind
            nimm();          // nimm ein Korn auf
        }
    }
}
```

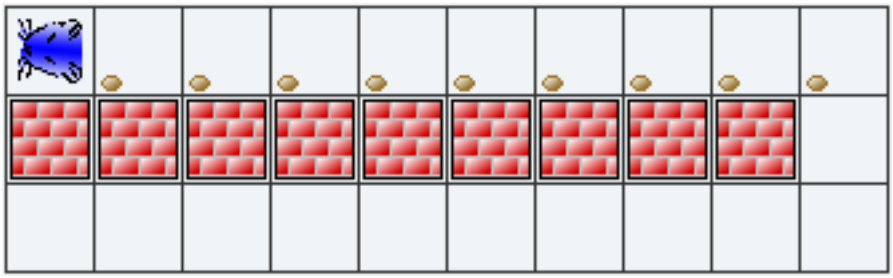
Listing 2: Musterlösung der Aufgabe 2

3.3 Aufgabe 3

Bei dieser Aufgabe soll der Hamster die Körner auf der untersten Reihe einsammeln und auf der obersten Reihe wieder ablegen.



(a) Ausgangslage



(b) Ziel

Abbildung 4: Spielfeld der Aufgabe 3

➥ Schreibe das entsprechende Programm, das diese Aufgabe löst. Verwende die Befehle aus der vorherigen Aufgabe um die Körner automatisch einzusammeln und setze wieder eine `while()`-Anweisung ein, um die Körner auf die oberen Felder zu verteilen.

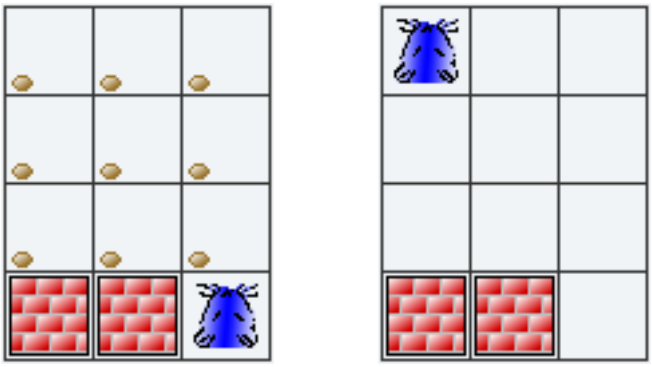
Musterlösung

```
void main()
{
    while(vornFrei()) {      // solange vorne frei ist
        vor();               // gehe zum nächsten Feld
        nimm();              // nimm das Korn
    }
    linksUm();               // gehe auf die andere Seite
    vor();                   // ...
    vor();                   // ...
    linksUm();               // ...
    while(vornFrei()) {      // solange vorne frei ist
        gib();               // gib ein Korn ab
        vor();               // gehe zum nächsten Feld
    }
}
```

Listing 3: Musterlösung der Aufgabe 3

3.4 Aufgabe 4

Bei dieser Aufgabe soll der Hamster alle Körner auf dem Spielfeld einsammeln.



(a) Ausgangslage (b) Ziel

Abbildung 5: Spielfeld der Aufgabe 4

➥ Schreibe das entsprechende Programm und benutze dabei nur die Funktionen aus Tabelle 1 ohne jegliche `while()`-Schlaufen. Tipp: Lasse den Hamster zeilenweise hinaufsteigen.

Musterlösung

```
void main()
{
    vor();
    nimm();

    linksUm();

    vor();
    nimm();
    vor();
    nimm();

    linksUm();
    linksUm();
    linksUm();

    vor();
    nimm();

    linksUm();
    linksUm();
    linksUm();

    vor();
    nimm();
    vor();
    nimm();

    linksUm();

    vor();
    nimm();

    linksUm();

    vor();
    nimm();
    vor();
    nimm();

    linksUm();
    linksUm();
    linksUm();
}
```

Listing 4: Musterlösung der Aufgabe 4

3.5 Aufgabe 5

Nimm nun die Musterlösung der vorherigen Aufgabe 4 als Vorlage. Da der Hamster den Befehl für "rechts um" nicht kennt, muss immer dreimal die Funktion `linksUm()` geschrieben werden. Um uns die Arbeit zu vereinfachen, definieren wir jetzt selber eine Funktion, die diese Arbeit übernimmt.

Eine Funktion kann unter anderem benutzt werden um mehrere andere Funktionen zusammen zu fassen. So wird ein ganzer Funktionsblock über einen einzigen Namen ausgeführt.

Unsere neue Funktion soll `rechtsUm()` heissen und dreimal die `linksUm()`-Funktion ausführen. Die Definition dieser neuen Funktion muss übrigens ausserhalb der `main()`-Funktion stehen:

```
1 void rechtsUm()
2 {
3     linksUm();
4     linksUm();
5     linksUm();
6 }
```

Zudem kann eine Funktion definiert werden, die alle Körner auf dem Weg aufnimmt, ähnlich wie in Aufgabe 3:

```
1 void sammle_geradeaus()
2 {
3     while(vornFrei()) {
4         vor();
5         nimm();
6     }
7 }
```

➥ Schreibe nun die Musterlösung der vorherigen Aufgabe 4 neu. Setze dabei die zwei neu erstellten Funktionen `rechtsUm()` und `sammle_geradeaus()` ein.

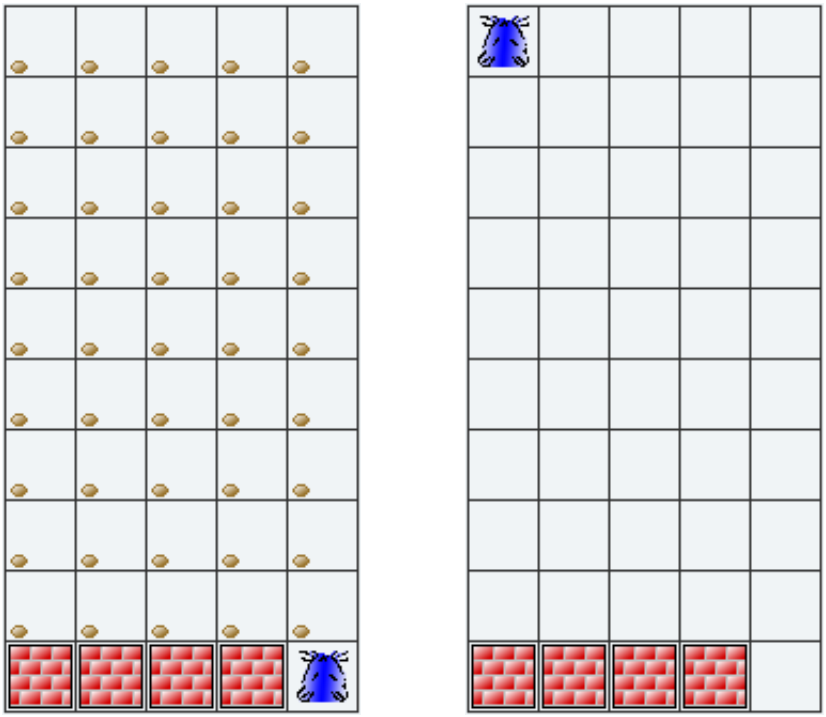
Musterlösung

```
1  void main()
2  {
3      vor();                // gehe ersten Zeile
4      nimm();               // nimm das erste Korn
5      linksUm();            // drehe nach links um
6      sammle_geradeaus();   // sammle die Körner auf dieser Zeile
7      rechtsUm();           // drehe nach oben um
8
9      vor();                // gehe zur nächsten Zeile
10     nimm();               // nimm das erste Korn
11     rechtsUm();           // drehe nach rechts um
12     sammle_geradeaus();   // sammle die Körner auf dieser Zeile
13     linksUm();            // drehe nach oben um
14
15     vor();                // gehe zur nächsten Zeile
16     nimm();               // nimm das erste Korn
17     linksUm();            // drehe nach links um
18     sammle_geradeaus();   // sammle die Körner auf dieser Zeile
19     rechtsUm();           // drehe nach oben um
20 }
21
22 void rechtsUm()
23 {
24     linksUm();
25     linksUm();
26     linksUm();
27 }
28
29 void sammle_geradeaus()
30 {
31     while(vornFrei()) {
32         vor();
33         nimm();
34     }
35 }
```

Listing 5: Musterlösung der Aufgabe 5

3.6 Aufgabe 6

Nun ist das Spielfeld um einiges grösser geworden!



(a) Ausgangslage (b) Ziel

Abbildung 6: Spielfeld der Aufgabe 6

Man könnte hier z.B. das Programm aus der vorherigen Aufgabe einfach aufblasen. Das würde aber auch dreimal mehr Programmzeilen benötigen. Stattdessen soll das Programm wieder einmal "intelligenter" gemacht werden, damit es "automatisch" durch die Zeilen läuft und alle Körner einsammelt.

Wenn man bei der letzten Musterlösung genau hinschaut, sieht man, dass das Programm aus drei ähnlichen Gruppen von Programmzeilen besteht. Der einzige Unterschied liegt in der Reihenfolge wie die Funktionen `linksUm()` und `rechtsUm()` aufgerufen werden:

```
1 vor();
2 nimm();
3 rechtsUm();
4 sammle_geradeaus();
5 linksUm();
```

```
1 vor();
2 nimm();
3 linksUm();
4 sammle_geradeaus();
5 rechtsUm();
```

Je nachdem auf welcher Zeile des Spielfelds sich der Hamster gerade befindet, muss er zuerst nach links oder zuerst nach rechts abbiegen. Am Ende der Zeile muss er dann entsprechend in die andere Richtung abbiegen.

Ein “intelligentes” Programm führt einfach diese zwei Programmblöcke abwechslungsweise aus, bis der Hamster am oberen Spielfeldende angekommen ist.

Dafür benötigen wir zwei neue Werkzeuge: eine Variable und die `if()`-Anweisung. Variablen sind wie “Behälter” und können Daten (z.B. Zahlen) speichern. In unserem Fall wollen wir mit einer Variable uns merken, welcher Block als nächstes ausgeführt werden soll.

Die Variable muss am Anfang der `main()`-Funktion definiert werden. Wir setzen hier auch gleichzeitig den Startwert der Variable auf 0:

```
1  void main()
2  {
3      int Richtung = 0;
4      ...
5  }
```

Um den Variablenwert weiter unten im Programm zu ändern (z.B. auf 1), kann man dies mit dem folgenden Befehl machen: `Richtung = 1;`

Die `if()`-Anweisung funktioniert ähnlich wie die `while()`-Anweisung. Zwischen den runden Klammern wird die Bedingung geschrieben. Ist die Bedingung `wahr`, dann werden die Befehle zwischen den geschweiften Klammern nach dem “`if`” ausgeführt. Ist die Bedingung `falsch`, dann werden die Befehle zwischen den geschweiften Klammern nach dem “`else`” ausgeführt.

```
1  if( Bedingung ) {
2      ...
3      diese Befehle werden ausgeführt, falls Bedingung=wahr
4      ...
5  }
6  else {
7      ...
8      diese Befehle werden ausgeführt, falls Bedingung=falsch
9      ...
10 }
```

➥ Schreibe nun die Musterlösung aus Aufgabe 5 neu mit der `if()`-Anweisung. Die Bedingung soll “`Richtung==0`” heissen, d.h. sie ist `wahr`, falls die Variable `Richtung` den Wert `0` hat und sie ist `falsch`, falls die Variable irgend einen anderen Wert hat, z.B. 1. Füge die zwei Programmblöcke zwischen den richtigen geschweiften Klammern der `if()`-Anweisung.

Musterlösung

```
void main()
{
    int Richtung = 0;            // definiere die Variable
    while(vornFrei()) {          // solange es noch Zeilen gibt
        vor();                   // gehe zur nächsten Zeile
        nimm();                  // nimm erstes Korn der Zeile
        if(Richtung == 0) {      // falls Richtung gleich 0 ist
            linksUm();           // drehe nach links um
            sammle_geradeaus(); // sammle Körner dieser Zeile
            rechtsUm();          // drehe nach oben um
            Richtung = 1;        // wechsle die Richtung
        }
        else {                   // falls Richtung nicht 0 ist
            rechtsUm();          // drehe nach rechts um
            sammle_geradeaus(); // sammle Körner dieser Zeile
            linksUm();           // drehe nach oben um
            Richtung = 0;        // wechsle die Richtung
        }
    }
}

void sammle_geradeaus()
{
    while(vornFrei()) {
        vor();
        nimm();
    }
}

void rechtsUm()
{
    linksUm();
    linksUm();
    linksUm();
}
```

Listing 6: Musterlösung der Aufgabe 6

3.7 Aufgabe 7

Hier muss der Hamster die zwölf Körner im nächsten Feld aufnehmen und diese gemäss untenstehendem Bild auf den folgenden Feldern verteilen.

(a) Ausgangslage (b) Ziel

Abbildung 7: Spielfeld der Aufgabe 7

Eine mögliche Lösung wäre z.B.:

```
1  void main() {
2      vor();                  // gehe zum nächsten Feld
3
4      while(kornDa()) {       // solange Körner da sind
5          nimm();             // nimm ein Korn auf
6      }
7
8      vor();                  // gehe zum nächsten Feld
9      gib();                  // gib drei Körner ab
10     gib();                  // ...
11     gib();                  // ...
12
13     vor();                  // gehe zum nächsten Feld
14     gib();                  // gib vier Körner ab
15     gib();                  // ...
16     gib();                  // ...
17     gib();                  // ...
18
19     vor();                  // gehe zum nächsten Feld
20     gib();                  // gib fünf Körner ab
21     gib();                  // ...
22     gib();                  // ...
23     gib();                  // ...
24     gib();                  // ...
25
26     vor();                  // gehe zum letzen Feld
27 }
```

Nun wollen wir dieses Programm so ergänzen, dass wir eine Reihe von `gib()`-Funktionen nicht von Hand aufschreiben müssen, sondern eine entsprechende Funktion soll die gewünschte Anzahl an `gib()`-Funktionen selber aufrufen.

Dafür benötigen wir wiederum zwei neue Werkzeuge: eine Funktion mit Parameter und die `for()`-Schlaufe.

Bisher haben wir Funktionen nur zum Gruppieren von anderen Funktionen benutzt. Wir können eine Funktion aber auch "parametrisieren", d.h. beim Aufruf der Funktion können Parameter (z.B. Zahlen) mitgegeben werden, die das Verhalten der Funktion steuern können.

In unserem Fall wollen wir eine Funktion mit einem Parameter, das die Anzahl der `gib()`-Aufrufe angeben soll:

```
1 void gib_koerner(int anzahl)
2 {
3     ...
4 }
```

Der Parameter `anzahl` ist nichts anderes als eine Variable, deren Wert innerhalb der Funktion gelesen und auch überschrieben werden kann.

Innerhalb der Funktion benötigen wir nun die `for()`-Schlaufe um die Anzahl der `gib()`-Aufrufe zu zählen. Die `for()`-Schlaufe benötigt in der Regel eine zusätzliche Variable (sog. Zählvariable), die am Anfang der Funktion definiert werden muss. Das folgende Beispiel zeigt den typischen Aufbau einer `for()`-Schlaufe:

```
1 int i;
2 for(i=0; i<10; i=i+1) {
3     vor();
4 }
```

In diesem Fall wird die `vor()`-Funktion zehnmal aufgerufen, d.h. die Variable `i` zählt von 0 bis 9. Zwischen den runden Klammern werden drei Bereiche definiert, die mit Semikolons (;) getrennt werden. Im linken Bereich wird die Initialisierung der Zählvariable (`i=0`) gemacht. Der mittlere Bereich enthält die Bedingung (`i<10`, `i` kleiner als `10`), die `wahr` sein muss, damit die Befehle zwischen den geschweiften Klammern ausgeführt werden. Der rechte Bereich wird nach jeder Ausführung der geschweiften Klammern ausgeführt. Hier wird die Zählvariable um eins hochgezählt.

➥ Ändere nun den Lösungsvorschlag am Anfang dieser Aufgabe so, dass die `gib()`-Befehle mittels der `gib_koerner()`-Funktion aufgerufen werden können, z.B. `gib_koerner(3);` Vergiss nicht, dass vor jedem `gib()`-Aufruf zuerst geprüft werden muss ob es noch Körner im Maul des Hamsters hat. Tipp: Benutze dafür die `if()`-Anweisung. Ein Ausrufezeichen (!) vor einer Bedingung invertiert die Aussage einer Funktion, z.B. `if(!maulLeer())`.

Musterlösung

```
1  void main() {
2      vor();                // gehe zum nächsten Feld
3
4      while(kornDa()) {     // solange Körner da sind
5          nimm();           // nimm ein Korn auf
6      }
7
8      vor();                // gehe zum nächsten Feld
9      gib_koerner(3);       // gib drei Körner ab
10
11     vor();                // gehe zum nächsten Feld
12     gib_koerner(4);       // gib vier Körner ab
13
14     vor();                // gehe zum nächsten Feld
15     gib_koerner(5);       // gib fünf Körner ab
16
17     vor();                // gehe zum letzten Feld
18 }
19
20 void gib_koerner(int anzahl)
21 {
22     int i;
23     for(i=0;i<anzahl;i++) { //
24         if(!maulLeer()) {
25             gib();
26         }
27     }
28 }
```

Listing 7: Musterlösung der Aufgabe 7

3.8 Bonusaufgabe

➥ Schreibe ein Programm, das den Hamster durchs Labyrinth bis zum Korn führt. Benutze dabei das eben gelernte Wissen, um ein möglichst "intelligentes" und möglichst kurzes Programm zu schreiben.

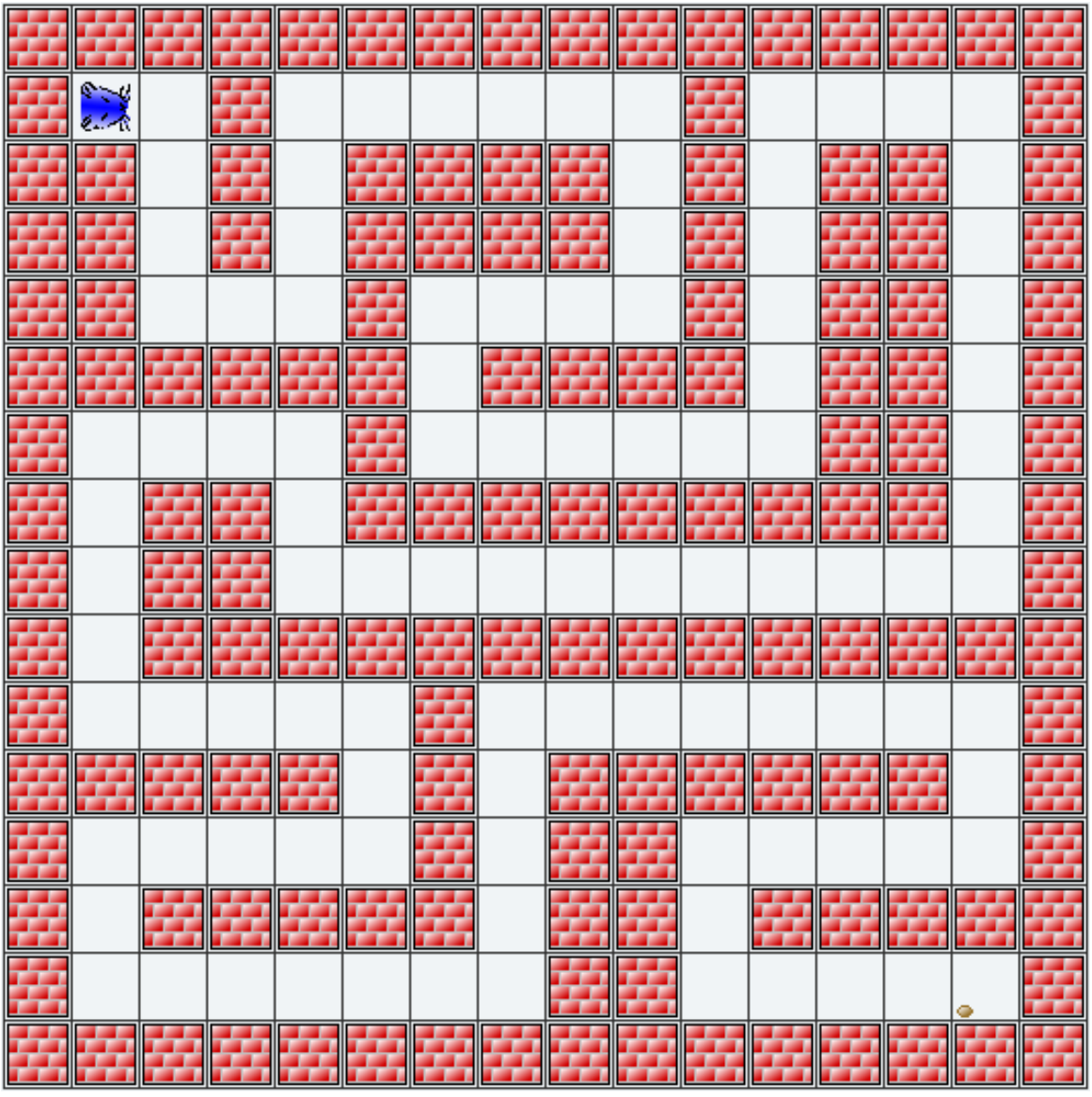


Abbildung 8: Spielfeld der Bonusaufgabe