Programmieren für Ingenieure – Übungs-Klausur

2014-xx-xx

Name: _____

Matrikelnummer: _____

Studiengang: _____ **seit** _____

Dauer: 120 Minuten (2 Stunden)

Zugelassene Hilfsmittel: Schreibgeräte. Zusätzliches Papier erhalten Sie vom Aufsichtspersonal.

Hilfe: Bei Fragen wenden Sie sich an das Aufsichtspersonal.

Diese Klausur hat **11 Seiten.** Bitte prüfen Sie, ob alle Seiten vorhanden sind.

In dieser Klausur können Sie bis zu **60 Punkte** erreichen.
Die Klausur ist mit **30 Punkten** oder mehr bestanden.

Aufgabe		Max. Punkte	Erreichte Punkte
1	Algorithmen	12	
2	Board-Programmierung	20	
3	Datenstrukturen	15	
4	Programmverständnis	8	
5	Wundertüte	5	
Summe		60	

Punkte

Note

Notizen

1 Algorithmen [12 Punkte]

Die Fakultätsfunktion $n!$ ist für natürliche Zahlen $n \in \mathbb{N}$ wie folgt definiert:

$$n! = n \cdot (n-1)! = 1 \cdot 2 \cdot \cdots \cdot n$$

wobei $0! = 1! = 1$ gilt. Die Fakultät von $n = 4$ ist somit $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$.
Ihre Aufgabe ist es, eine Funktion `fak(n)` zu implementieren, die für das ganzzahlige Argument $n \geq 0$ dessen Fakultät liefert:

```
int fak(int n) {
    // Code hier einfügen
}
```

a) [4 Punkte] Implementieren Sie `fak()` mit Hilfe einer *Schleife*. Geben Sie den Code an.

Beispiellösung.

```
int fak(int n) {
  if (n <= -1) return -1;  // Guter Stil

  int result = 1;

  for (int i = 1; i <= n; i++) {
    result *= i;  // Entspricht:  result = result * i;
  }

  return result;
}
```

b) [4 Punkte] Implementieren Sie `fak()` *rekursiv*, indem Sie `fak()` sich selbst aufrufen lassen. Geben Sie den Code an.

Beispiellösung.

```
int fakRec(int n) {
  if (n <= -1) return -1;  // Guter Stil
  if (n <= +1) return +1;

  return n * fakRec(n - 1);
}
```

c) [4 Punkte] Legen Sie ein *Feld* an, in dem Sie die Fakultätswerte 0! bis 5! ablegen. Wird `fak()` mit $0 \leq n \leq 5$ aufgerufen, geben Sie das Ergebnis direkt aus dem Feld zurück. Geben Sie den ergänzenden Code an.

Beispiellösung.

```
int fakAry[] = { 1, 1, 2, 6, 24, 120 };

int fakCache(int n) {
  if (n >= 0 && n <= 5) {
    return fakAry[n];
  }

  return fak(n);
}
```

2 Board-Programmierung [20 Punkte]

An ein Arduino/Galileo-Board sind angeschlossen:

- Ein *prellfreier Taster*, angeschlossen an Pin 8 und verbunden mit −(GND). Wird der Taster gedrückt, liegt an Pin 8 LOW an, sonst HIGH.
- Ein *LCD-Display*, angeschlossen an Pins 2–7.
- Eine LED, angeschlossen an Pin 13.

Der Code zum Einrichten der Bauteile liegt bereits vor:

```
#include <LiquidCrystal.h>

// Taster
int buttonPin = 8;

// LED
int ledPin = 13;

// LCD-Anzeige
int rsPin = 2;
int ePin  = 3;
int d4Pin = 4;
int d5Pin = 5;
int d6Pin = 6;
int d7Pin = 7;

LiquidCrystal lcd(rsPin, ePin, d4Pin, d5Pin, d6Pin, d7Pin);

void setup() {
  // put your setup code here, to run once:
  pinMode(rsPin, OUTPUT);
  pinMode(ePin,  OUTPUT);
  pinMode(d4Pin, OUTPUT);
  pinMode(d5Pin, OUTPUT);
  pinMode(d6Pin, OUTPUT);
  pinMode(d7Pin, OUTPUT);

  pinMode(ledPin, OUTPUT);
  pinMode(buttonPin, INPUT_PULLUP);

  lcd.begin(16, 2);
  lcd.clear();
}

void loop() {
  // put your main code here, to run repeatedly:
}
```

Ihre Aufgabe ist es, eine elektronische *Stoppuhr* zu implementieren:

1. Wird der Taster das erste Mal gedrückt, beginnen Sie mit der Zeitmessung. Auf dem LCD-Display soll der Text „Running“ erscheinen.
2. Wird der Taster das zweite Mal gedrückt, soll auf dem LCD-Display die zwischen dem ersten und dem zeiten Tastendruck abgelaufene Zeit (in Millisekunden) angezeigt werden.
3. Anschließend fahren Sie bei Schritt 1 fort (bei erneutem Drücken erfolgt eine neue Zeitmessung).

Nutzen Sie die folgenden Funktionen:

```
int digitalRead(int pin);                 // Pin auslesen (LOW oder HIGH)
void digitalWrite(int pin, int state); // Pin auf state setzen (LOW oder HIGH)
unsigned long millis();                   // Millisekunden seit Programmstart
lcd.print(x);                        // Zahl oder Zeichenkette x auf LCD ausgeben
lcd.clear();                              // LCD-Anzeige löschen
```

a) [10 Punkte] Implementieren Sie die Stoppuhr mit Hilfe der angegebenen Funktionen. (Hinweis: Lesen Sie zunächst den Rest der Aufgabenstellung, damit Sie mögliche Ergänzungen planen können.)

Beispiellösung.

```
long runningSince = -1;

// Prinzipiell liesse sich der Zustand (ob aufgezeichnet
// wird oder nicht) auch als  runningSince == -1
// repräsentieren. Der Autor der Beispiellösung ist der
// Meinung, dass der Code sich mit einer expliziten
// Variable besser liest.
int isRecording = 0;

void loop() {
  int buttonDown = digitalRead(buttonPin) == LOW;

  if (buttonDown && !isRecording) {
    isRecording = 1; runningSince = millis();

    lcd.clear();
    lcd.print("Running");
  } else if (buttonDown && isRecording) {
    isRecording = 0;
    printTime(millis() - runningSince);
  }

  if (buttonDown) {
    // Warten bis der Button nicht mehr gedrückt ist
    while (digitalRead(but) == LOW) {}
  }
}

void printTime(long delta) {
  lcd.clear(); lcd.print(delta);
}
```

b) [6 Punkte] Ergänzen Sie die Stoppuhr so, dass während des Messvorgangs die LED blinkt.

Beispiellösung. *Wir definieren eine Reihe neuer Variablen und eine neue Funktion:*

```
long lastBlink = -1;
int blinkDuration = 1000 / 4; // 4 times per second
int ledOn = 0;

void handleBlink() {
  if (isRecording) {
    if (millis() - lastBlink > blinkDuration) {
      ledOn = !ledOn;
      digitalWrite(ledPin, ledOn);
      lastBlink = millis();
    }
  }
}
```

Und müssen beachten, dass wir `handleBlink()` *entweder am Ende oder Anfang der* `loop()`*-Funktion aufrufen müssen.*

c) [2 Punkte] Ergänzen Sie die Stoppuhr so, dass die Zeit als Sekunden mit drei Nachkommastellen ausgegeben wird. (Tip: Geben Sie zunächst den ganzzahligen Teil aus, gefolgt von „.“ und den Nachkommastellen.)

Beispiellösung. *Diese Teilaufgabe war schwerer als gedacht. Es gibt verschiedene Möglichkeiten, sie zu lösen, indem wir die von uns in Teilaufgabe a) definierte* `printTime()`*-Funktion ersetzen:*

```
void printTime(long delta) {
  lcd.clear();

  int deltaSeconds   = delta / 1000;
  int deltaRemMillis = delta % 1000;

  lcd.print(deltaSeconds);
  lcd.print(".");

  if (deltaRemMillis < 10) { lcd.print("0"); }
  if (deltaRemMillis < 100) { lcd.print("0"); }

  lcd.print(deltaRemMillis);
}
```

```
void printTime(long delta) {
  lcd.clear();

  float deltaF1000 = delta;
  float deltaF = deltaF1000 / 1000.0;
  lcd.print(deltaF, 3);
}
```

```
void printTime(long delta) {
  lcd.clear();

  float deltaF1000 = delta;
  float deltaF = deltaF1000 / 1000.0;

  char buf[256];
  sprintf(buf, "%.03f", deltaF);
  lcd.print(buf);
}
```

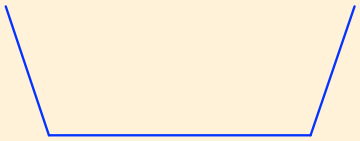
Und müssen beachten, dass wir `handleBlink()` *entweder am Ende oder Anfang der* `loop()`*-Funktion aufrufen müssen.*

d) [2 Punkte] Was müssen Sie berücksichtigen, wenn Ihr Taster *nicht* prellfrei ist? (Eine Beschreibung als Text genügt.)

Beispiellösung. *Prellende Taster federn mehrmals zwischen 0 und 1 hin und her.*

Wir können uns merken, wann der Taster zum letzten Mal den Zustand gewechselt hat und einen Zustandswechsel nur dann berücksichtigen, wenn er länger als einen bestimmten Schwellwert her ist.

Damit erkennen wir langes Drücken:



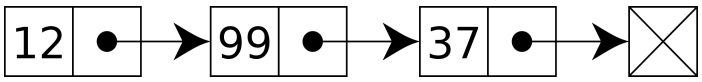
Aber ignorieren kurzes Drücken wie es beim Prellen auftritt:

3 Datenstrukturen [15 Punkte]

Eine *verkettete Liste* ist eine dynamische Datenstruktur, die eine Speicherung von miteinander in Beziehung stehenden Objekten erlaubt. Die Anzahl der Objekte ist im Vorhinein nicht bestimmt. Die Liste wird durch *Zeiger* auf das jeweils folgende Element realisiert.

Das folgende Bild zeigt eine Liste, bestehend aus drei Elementen. Jedes Element ist definiert als

```
struct Elem {
    int value;          // Der Wert
    struct Elem *next; // Zeiger auf das nächste Element
};
```



Der Zeiger des letzten Elements (hier 37) hat den Wert NULL.

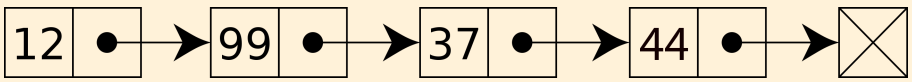
Um auf eine Liste zuzugreifen, fängt man beim ersten Element (hier 12) an, und folgt dann den Zeigern auf das jeweils nächste Element. Die folgende Funktion prüft, ob ein Element mit dem Wert x in der Liste e enthalten ist. Wenn ja, gibt sie einen Zeiger auf das Element zurück; wenn nicht, gibt sie NULL zurück.

```
// Erstes Element der Liste LIST mit Wert X zurückgeben
// (oder NULL, wenn nicht gefunden)
struct Elem *search(struct Elem *list, int x) {
   struct Elem *e = list; // Erstes Element
   while (e != NULL && e->value != x)
       e = e->next;
   return e;
}
```

Ihre Aufgabe ist es, eine Funktion zu schreiben, die ein gegebenes Element e als letztes Element an eine nicht-leere Liste *list* anhängt.

a) [3 Punkte] Nehmen wir an, Sie möchten ein Element mit Wert `44` an die Liste anhängen. Zeichnen Sie (ähnlich zu obigem Diagramm), wie die Liste nach dem Anhängen aussieht.

Beispiellösung.



b) [8 Punkte] Um ein Element anzuhängen, müssen Sie zunächst das letzte Element finden. Schreiben Sie eine Funktion `last()`, die (ähnlich wie `search()` oben) durch die Liste geht, und das letzte Element liefert.

```
// Letztes Element der Liste LIST zurückgeben
struct Elem *last(struct Elem *list) {
    // Ihr Code hier
}
```

Beispiellösung.

```
struct Elem *last(struct Elem *list) {
  struct Elem *result = NULL;

  while (list != NULL) {
    result = list;
    list = list->next;
  }

  return result;
}
```

c) [4 Punkte] Gegeben sei nun ein existierendes Element E. Nutzen Sie Ihre Funktion `last()`, um an das letzte Listenelement das Element E anzuhängen. Implementieren Sie die Funktion `append()` entsprechend; achten Sie darauf, dass der Zeiger des (neuen) letzten Elements anschließend NULL sein muss.

```
// Element E an die Liste LIST anhängen
void append(struct Elem *list, struct Elem *e) {
    // Ihr Code hier
}
```

Beispiellösung.

```
void append(struct Elem *list, struct Elem *e) {
  struct Elem *last = last(list);

  if (last == NULL) {
    Serial.println("Could␣not␣find␣last␣element␣of␣list␣
       in␣append()");
    exit(-1);
  }

  last->next = e;
  e->next = NULL;
}
```

4 Programmverständnis [8 Punkte]

Manche Programmfehler sind offensichtlich, andere sehr subtil. Jede der folgenden Funktionen soll den Wert 42 zurückgeben, hat aber einen Fehler.

a) Was tut die jeweilige Funktion stattdessen? („Syntax-Fehler“, „Gibt 41 zurück“, etc.)

b) Geben Sie eine möglichst kleine Korrektur an, die den Fehler behebt.

```
int f1() {
    return 43;
}

int f2() {
    return 42
}

int f3() {
    if (5 < 2);
        return 43;
    return 42;
}

int f4() {
    int n = 0;
    while (n <= 42)
        n++;
    return n;
}

int f5() {
    int a[] = {40, 1, 1};
    int sum = 0;
    for (int i = 1; i < 3; i++)
        sum = sum + a[i];
    return sum;
}

int f6() {
    int n = 21;
    for (int n = 0; n < 1; n++)
       n = n * 2;
    return n;
}

int f7() {
    int x = 0;
    while (x < 1000) {
        if (x == 42)
            return x;
    }
    return 42;
}

int f8() {
    int n = 42;
    if (n = 43)
        n = 44;
    return n;
}
```

Beispiellösung.

Sollte 42 sein.

Fehlendes Semikolon.

Das fälschlich gesetzte Semikolon hinter dem `if` führt dazu, dass das `if` schon hier endet...
...und diese Zeile un-bedingt ausgeführt wird.

Off by one, korrigiert:

```
while (n < 42)
```

Das erste Element wird ignoriert. Korrigiert:

```
for (int i = 0; i < 3; i++)
```

Der Variablename `n` wird doppelt verwendet, was hier zu Problemen führt. Korrigiert:

```
for (int i = 0; i < 1; i++)
```

Die Variable `x` wird niemals hochgezählt, was zu einer Endlosschleife führt. Korrigiert:

```
x++; }
```

Irrtümlicherweise wird `n` 43 zugewiesen statt mit 43 zu vergleichen. Korrigiert:

```
if (n == 43)
```

5 Wundertüte [5 Punkte]

Prüfen Sie die folgenden Aussagen. Kreuzen Sie „W“ an, wenn sie wahr sind, oder „F“, wenn sie falsch sind.

Bewertung:

- $+\frac{1}{2}$ Punkt pro korrekte Antwort,
- ± 0 Punkte für keine Antwort,
- $-\frac{1}{2}$ Punkt für eine falsche Antwort.

Für die Gesamtaufgabe werden mindestens 0 Punkte vergeben.

1. **W** **F** *[W]* Ein *Algorithmus* besteht aus einer Folge von Schritten.
2. **W** **F** *[F]* Eine `int`-Variable kann beliebig große ganzzahlige Werte annehmen.
3. **W** **F** *[W]* Eine *Schleife* kann Anweisungen wiederholt ausführen.
4. **W** **F** *[F]* Die *von-Neumann-Architektur* trennt Programmspeicher und Datenspeicher.
5. **W** **F** *[W]* Eine nicht konstante *globale Variable* kann von jeder Funktion geändert werden.
6. **W** **F** *[F]* Greift ein C-Programm außerhalb der Grenzen auf ein Feld zu, bricht es sofort ab.
7. **W** **F** *[F]* Ein digitaler Eingang unterscheidet `LOW`, `HIGH`, und `PULLUP_HIGH`.
8. **W** **F** *[F]* Alan Turing gilt als der erste Programmierer.
9. **W** **F** *[F]* Eine Funktion vom Typ `void` gibt stets NULL zurück.
10. **W** **F** *[W]* Eine *Bibliothek* stellt zusätzliche Funktionalität zur Verfügung.