C-Kurs E3: Vertiefung

Zielsetzung:
- Speicherklassen, Lebensdauer & Sichtbarkeit
- Makros mit **`#define`**
- **`#include`** Präprozessoranweisung

Aufgabenblock 1: Speicherklassen, Lebensdauer & Sichtbarkeit
Hier sind einige Aufgaben aufgeführt, welche die verschiedenen Speicherklassen untersuchen, die die Sprache C zur Verfügung stellt.

A 29: Speicherklasse **`auto`**
Warum wird beim untenstehenden Code für die Zahl **`23`** auf der Konsole ausgegeben, obwohl die Variable **`a`** in der Funktion **`increment`** um eins erhöht wird? Korrigieren Sie das Programm so, dass die gewünschte Funktionalität erfüllt wird.

```
void increment(int x) { x = x + 1; }
int main() {
     int a=23;
     increment(a);
     printf("Die Zahl lautet: %d\n", a);
}
```

A 30: Speicherklasse **`static`**
Schreiben Sie eine Funktion, welche sich selbst merken kann, wie häufig sie aufgerufen wurde. Dabei sollen keine globalen Variablen eingesetzt werden. Der Zähler soll innerhalb der Funktion als statische Variable definiert werden. Rufen Sie in Ihrem **`main`**-Programm die Funktion einige Male auf, um sie zu testen.

A 31: Speicherklasse **`extern`**
Mit **`extern`** können Variablen oder Funktionen aus anderen Programmdateien in unser Programm eingebunden werden. Um den Einsatz von **`extern`** zu demonstrieren, brauchen wir also mindestens zwei Dateien:
- Schreiben Sie eine Datei **`a-31a.c`**, welche eine globale Variable **`HSR`** definiert, die den Namen der Hochschule festlegt (Datentyp **`char *`**). Fügen Sie der Datei ausserdem die Funktion **`double inverse(double value)`** hinzu, welche den Kehrwert (1/x) einer Zahl liefern soll.
- Schreiben Sie eine zweite Datei **`a-31b.c`**, welche die **`main`**-Funktion des Programms enthält. Die oben genannte Variable und Funktion sollen mit **`extern`** deklariert und aus der **`main`**-Funktion aufgerufen werden.
- Um das Programm zu übersetzen, müssen beide Quellcodedateien angegeben werden:
  `gcc a-31a.c a-31b.c`

A 32: **`static`** bei globalen Variablen und Funktionen: Datenkapselung

Wie wir in der Aufgabe A 31 gesehen haben, gibt es mit **`extern`** die Möglichkeit, auf globale Variablen oder Funktionen zuzugreifen, die nicht im "eigenen" Sourcecode implementiert sind. Dies kann aber das Prinzip der Datenkapselung verletzen, da bei einer Quellcodedatei in der Regel nicht auf alle Funktionen und Variablen zugegriffen werden darf. Um den Code vor unerlaubten **`extern`**-Zugriffen zu schützen, können globale Variablen oder Funktionen mit einem vorgestellten **`static`** abgesichert werden. Die Funktionen und Variablen können innerhalb der Datei verwendet werden, sind aber von ausserhalb nicht einsehbar. (Das **`static`** entspricht in diesem Fall dem Schlüsselwort **`private`** in Java).

- Erweitern Sie die Datei **`a-31a.c`** um eine weitere Variable und eine Funktion, welche Sie mit **`static`** absichern.
- Versuchen Sie im Programm **`a-31b.c`** die neue Variable und die Funktion mit **`extern`** einzubinden und das Programm zu übersetzen. Wenn alles klappt, sollte der Compiler eine Fehlermeldung bringen, die externen Symbole können nicht aufgelöst werden.

Aufgabenblock 2: Die `#define` Präprozessor-Anweisung

Die **`#define`** Anweisung kann für symbolische Konstanten oder Makros eingesetzt werden. Die **`#define`**-Anweisungen werden vom Precompiler vor der Compilierung des Programmcodes ausgeführt.

A 33: Symbolische Konstanten mit **`#define`**

Makros ohne Parameter werden als symbolische Konstanten bezeichnet und haben die Form
`#define Bezeichner Ersatztext`

- Schauen Sie in der Datei **`math.h`** nach, unter welchem Bezeichner und auf wieviele Stellen genau die Zahl PI definiert ist. Sie finden die Header-Datei auf den Übungsrechnern im Übungsverzeichnis **`bereitgestellt`**.
- Schreiben Sie ein Programm, welches den Sinus von 0 ... PI bestimmt, wobei insgesammt **`buf_size`** Teilwerte berechnet und in einem Vektor abgelegt werden sollen. Bei einer **`buf_size`** von **`5`** sollen also die Sinuswerte für die Winkel 0, PI/4, PI/2, 3PI/4 und PI bestimmt werden. Die Grösse von **`buf_size`** soll mit einer **`#define`**-Anweisung festgelegt werden.
  Hinweis: Unter Linux muss der Compiler bei dieser Aufgabe mit der zusätzlichen Option **`-lm`** aufgerufen werden (damit wird die Mathematik-Programmbibliothek referenziert).

A 34: Funktionsweise des Precompilers visualisieren

Bei der GCC gibt es mit der Option **`-E`** die Möglichkeit, ausschliesslich den Präprozessor ohne Compiler und Linker auszuführen. Tippen Sie den untenstehenden Code ab und führen Sie den Precompiler mit **`gcc -E a-34.c`** aus. Ist die symbolische Konstante **`KONSTANT`** im Programmcode anschliessend noch zu finden?

```
#define KONSTANT 23
int main() {
     int a = KONSTANT;
     int b = a + KONSTANT;
     return 0;
}
```

A 35: Makro mit Parameter

Ein Makro mit Parametern kann aus dem Programmcode wie eine Funktion aufgerufen werden. Beim Ausführen des Makros werden die Parameter im Ersatztext durch den(die) angegebenen Wert(e) ersetzt. Bei der Notation des **`define`**-Befehls ist zu achten, dass zwischen **`Bezeichner`** und **`(`** kein Leerzeichen steht:

```
#define Bezeichner(Parameter1, ..., ParameterN) Ersatztext
```

- Schreiben Sie ein Makro **`print`**, welches den übergebenen Parameter auf der Konsole ausgibt. Das Makro soll im Programm z.B. mit **`print("HSR");`** aufgerufen werden können.

A 36: Makro mit Parametern II

Es soll ein Makro geschrieben werden, welches das Quadrat einer Zahl bestimmt. Die Definition des Markos lautet demzufolge:

```
#define square(x) x*x
```

Das untenstehende Programm setzt das Makro ein:

```
int main() {
      double y = 3.0;
      printf("Das Quadrat von %.2f ist %.2f\n", y, square(y));
      printf("Das Quadrat von %.2f ist %.2f\n", y+1, square(y+1));
      return 0;
}
```

Wenn Sie das Programm ausführen, werden Sie feststellen, dass das Resultat der ersten Ausgabe korrekt ist, bei der zweiten Ausgabe hingegen nicht. Wo könnte das Problem liegen? Untersuchen Sie die ersetzten Makros mit der GCC **`-E`** Option und korrigieren Sie das Makro, indem Sie Klammern an den erforderlichen Stellen setzen.

Aufgabenblock 3: Header-Dateien

Mit der **`#include`**-Anweisung werden Dateien vom Precompiler in den Sourcecode kopiert.

A 37: Wie wir bereits mehrfach gesehen haben, werden in C die Funktionen am Anfang des Programmcodes deklariert. Diese Deklarationen werden normalerweise in einer separaten Datei, der Header-Datei gespeichert. Im Programmcode selbst wird die Header-Datei mit der **`#include`** Anweisung eingebunden:

`#include <dateiname>` oder **`#include "dateiname"`**

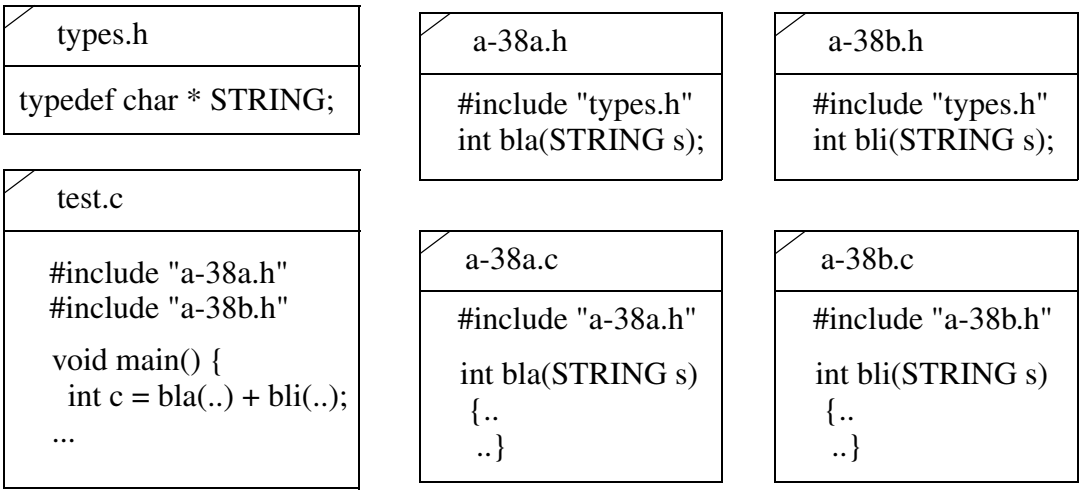
Ist der Dateiname in spitzigen Klammern **`< >`** eingeschlossen, sucht der Compiler in den typischen include-Verzeichnissen. Bei **`""`**-Zeichen sucht der Compiler zuerst im aktuellen Verzeichnis.

- Schreiben Sie ein Programm, welches neben der **`main`**-Funktion zwei Funktionen implementiert, die in einer separaten Header-Datei deklariert sind. Die beiden Funktionen sollen **`bla`** und **`bli`** heissen und einen **`int`** als Resultat liefern.

A 38: Programme mit mehreren Quellcodedateien

In der Regel setzen sich Programme aus vielen Quellcodedateien zusammen. Untenstehend ist ein einfaches Beispiel schematisch dargestellt. Die Funktion der Datei **`a-38a.c`** wird in der Header-Datei **`a-38a.h`** deklariert. Entsprechend die Funktion der Datei **`a-38b.c`** in der Header-Datei **`a-38b.h`**. Zusätzlich dient die Header-Datei **`types.h`** dazu den sowohl in **`a-38a.c`** und **`a-38b.c`** verwendeten Datentyp **`STRING`** zu vereinbaren.

Das Programm **`test.c`** enthält die **`main`**-Funktion und verwendet Funktionen der Datei **`a-38a.c`** und **`a-38b.c`**. Hierzu werden in der Datei **`test.c`** mit **`include`**-Anweisungen die Header-Dateien **`a-38a.h`** und **`a-38b-h`** eingebunden.



Um das Programm zu übersetzen, müssen alle Quellcodedateien angegeben werden:

```
gcc test.c a-38a.c a-38b.c
```

- Implementieren Sie das obenstehende Beispiel. Übersetzen Sie das Programm in der Konsole.
- Wie sie sehen, reklamiert der Compiler einen Fehler im Zusammenhang mit der Include-Datei **`types.h`**. Es wird ungewollt die Typvereinbarung für **`STRING`** zweimal eingebunden: Führen Sie den Precompiler aus, um den Effekt sichtbarzumachen:
  `gcc -E test.c`
  (Am besten sieht man den Effekt, wenn Sie kein **`#include <stdio.h>`** verwenden, da sonst alle Funktionsdeklarationen von **`stdio.h`** aufgelistet werden.)
- Wir können die "Mehrfachinkludierung" von Anweisungen aus Header-Dateien verhindern, indem wir in den betroffenen Headerdateien eine neue symbolische Konstante einführen. Für die Datei **`types.h`** sieht das folgendermassen aus:

```
#ifndef _TYPES_H
#define _TYPES_H
typedef char * STRING;
#endif
```

  Ändern Sie die Datei **`types.h`** entsprechend ab und führen Sie den Precompiler erneut aus. Wie Sie an der Ausgabe erkennen können, werden die Funktionsdeklarationen nur noch einmal aufgelistet. Wie lautet die symbolische Konstante in der Datei **`math.h`**, welche für diesen Zweck eingesetzt wird?