Praktikum C-Programmierung

Sommersemester 2021

Name:

Gopala Krishnan

Vorname:

Bala Krishnan

Matrikelnummer:

018330315

Studienrichtung: ○ EB ○ IB ◉ XB

Letzte bearbeitete Aufgabe:

8.2

Termin:

4

Bemerkung:

Hinweise zur Durchführung:

- Sie arbeiten in diesem Praktikum mit einem GCC in einer Ubuntu-Server-VM (Linux). Die VM finden Sie auf der Seite https://fbevirt.hs-bochum.de
- Speichern Sie die von Ihnen erstellten Projekte in der VM unter dem Pfad **`Documents/`**. Legen Sie dabei für jede Aufgabe ein eigenes Verzeichnis an.
- Geben Sie in jeder Datei Namen, Matrikelnummer und Aufgabe als Kommentar an.
- Verschwenden Sie keine Zeit mit endlosen Schönheitsoperationen an Ihren Programmen. Im Zweifelsfall fragen Sie, ob Sie die Aufgabe ausreichend bearbeitet haben.
- Die Beantwortung der Fragen in der Aufgabenstellung und eine ordentliche Quelltextformatierung sind Bestandteil der Aufgabenstellung!
- Bearbeiten Sie die Aufgaben vollständig und in der vorgegebenen Reihenfolge. **Überspringen Sie keine Aufgaben!**

1 Einleitung und Grundlagen

1.1 Umgang mit der virtuellen Maschine

Für die Aufgaben dieses Praktikums verwenden Sie eine virtuelle Maschine mit einer LTS-Version von **Ubuntu Server**.
Melden Sie sich nach dem Start mit dem Benutzernamen **praktikum** und dem Passwort **Pa$$w0rd** bei der VM an. Achtung! Das Passwort wird bei der Eingabe nicht angezeigt!
Nach der Anmeldung befinden Sie sich im Kontext des Benutzers **praktikum** und im Home-Verzeichnis dieses Benutzers. (**`home/praktikum/`**)
Die für diese Versuche benötigten Dateien befinden sich entweder im Unterverzeichnis **`HS_Versuche/`** in diesem Basisverzeichnis oder auf dem GIT-Server.
Ihre Versuche speichern Sie in einem Verzeichnis **`Documents/.`** Dieses Verzeichnis ist in Ihrem Benutzerverzeichnis bereits angelegt.
Lassen Sie sich zur Übersicht zunächst einen Dateibaum (**tree**) und ein Inhaltsverzeichnis mit den vorgegebenen Dateien und Verzeichnissen anzeigen.

1.2 Arbeiten mit dem GCC-Compiler

Kontrollieren Sie zunächst die Version des installierten C-Compilers mit **`gcc -v`**
Neben einer Reihe von Detailinformationen sollte am Ende der Ausgabe die Compilerversion angezeigt werden.

```
gcc version 9.3.0
```

Die Version des installierten make-Tools können Sie sich mit **`make -v`** anzeigen lassen.

```
GNU Make 4.2.1
```

1.3 Arbeiten mit vorgegebenen Quellen und MAKE

Für diesen Versuchsteil wird die Datei **`hs_1.tar`** benötigt
Bei dieser Datei handelt es sich um ein (im Gegensatz zu z.B. Zip) nicht komprimiertes Dateiformat in dem mehrere Dateien und Verzeichnisse zusammengefasst sein können.
Diese Datei soll vor dem Entpacken aus dem Verzeichnis **`HS_Versuche/`** in das Arbeitsverzeichnis **`Documents/`** kopiert werden. Dabei sollen ***relative*** *Pfadangaben* benutzt werden.
Lassen Sie sich zunächst mit der Anweisung *tree* den Verzeichnisbaum anzeigen.

Wie lautet die Kopieranweisung, wenn Sie sich beim Kopieren in dem Verzeichnis **`HS_Versuche/`** befinden?

```
cp hs_1.tar ../Documents
```

Löschen Sie die kopierte Datei. Wie lautet **die Kopieranweisung**, wenn Sie sich beim Kopieren in dem Verzeichnis **`Documents/`** befinden?

```
cp ../HS_Versuche/hs_1.tar .
```

Entpacken Sie nach dem Kopieren die Datei dort mit der Anweisung.

```
tar xfv hs_1.tar
```

nach dem Entpacken mit **tar** sollten Sie ein Verzeichnis **`Documents/hs_1/`** vorfinden.

Starten Sie aus diesem neuen Verzeichnis mit **`make`** die Übersetzung und führen Sie anschließend die erzeugte Datei **`hs_1`** aus.

Beobachten Sie die Ausgabe.

Vergleichen Sie die in Ihrem Projektverzeichnis nach dem Übersetzen und Ausführen des Projekts vorhandenen Dateien mit den ursprünglich vorhandenen Dateien.
Welche Datei wurde zusätzlich erzeugt?
Wo ist der Unterschied zwischen der hier erzeugten ausführbaren Datei und einer *.class*-Datei von Java?

hs_1.c wurde zusätzlich erzeugt. Der Unterschied liegt daran, dass die erzeugte Datei in byte code compiliert ist und kann direkt in Terminal ausgeführt werden. .class-Datei von Java braucht JVM für das Ausführen.

1.4 Verwendung von GIT

Ein übliches Hilfsmittel zur Verwaltung von Quellcodes ist die verteilte Versionsverwaltung GIT. Selbst dann wenn Programme nur lokal auf einem Rechner entwickelt werden kann GIT als Versionsverwaltung eine wertvolle Arbeitserleichterung sein.
Überprüfen Sie zunächst mit der Anweisung **git config --list** , dass es auf dem lokalen System noch keine Voreinstellungen gibt.

Setzen Sie jetzt Ihre Benutzerdaten mit den Anweisungen
```
git config --global user.name "Username"
git config --global user.email johndoe@stud.hs-bochum.de
```

Welche Anzeige ergibt sich jetzt bei **git config --list** ?

user.name=bago9654
user.email=bala.gopala-krishnan@stud.hs-bochum.de

Wechseln Sie in Ihr Arbeitsverzeichnis **`Documents/`** und laden Sie sich mit der Anweisung
`git clone http://10.102.50.40:30000/AID-Labor/C-Programmierung.git`
das Repository mit dem Testprogramm auf Ihren Rechner.
Welche Verzeichnisse und Dateien werden importiert?

Das Verzeichnis C-Programmierung und da drin das Verzeichnis ".git" und die Dateien ".gitignore" ,"hs_git.c" und einen makefile werden importiert.

Wechseln Sie in das importierte Verzeichnis. Die Anweisung **git log** zeigt Ihnen die Änderungshistorie des Repositories an.
Wie viele commits werden für das aktuelle Repository angezeigt?

1 Commit von dem Autor

Übersetzen Sie das Testprogramm und führen Sie das Programm aus. Lassen Sie sich ein Inhaltsverzeichnis anzeigen und beachten Sie die Veränderungen zum Ursprungszustand. Sehen Sie sich die Ausgabe des Befehls **git status** an.

Führen Sie jetzt eine Änderung am Quellcode durch und überprüfen Sie nochmals die Ausgabe von **git status**.
Mit der Anweisung **git add .** können Sie die Änderungen des Projekts für die Erfassung vorbereiten. Testen Sie anschließend ein weiteres Mal die Ausgabe von **git status**.

Die so vorbereiteten Änderungen können mit der Anweisung
`git commit -m "Kommentar"` im Repository gesichert werden. Lassen Sie sich zum Abschluss noch einmal die Informationen von **git status** und **git log** anzeigen.

Welche Datei ist bei diesen Änderungen nicht im Repository erfasst und gespeichert worden?

Die Veränderung zur Datei "hs_git.c" ist nicht im ausführbarer Datei gespeichert, deshalb wird die Datei "hs_git" nicht im Repository erfasst und gespeichert.

Jeder der von **git log** angezeigten commits ist durch einen 40-Stelligen Hashwert eindeutig gekennzeichnet.
Dabei sind normalerweise für die eindeutige Identifikation eines commit nicht alle 40 Stellen erforderlich, sondern es kann nach den ersten, eindeutigen Stellen mit **..** abgekürzt werden.
Unter Verwendung dieses Hashwertes können Sie sich den Unterschied zwischen dem aktuellen Stand des Repositories und dem mit diesem commit gespeicherten Zustand anzeigen lassen.
Testen Sie dies für die Datei **hs_git.c** mit dem Aufruf von
`git diff hashwert hs_git.c`

Welche Informationen werden jetzt angezeigt?

Die Veränderungen in dem Inhalt der Datei hs_git.c wird jetzt angezeigt.

1.5 Ein eigenes Programm

Legen Sie in Ihrem Arbeitsverzeichnis (`Documents/`) ein Projektverzeichnis `hs_2` an.
Erzeugen Sie in diesem neuen Verzeichnis mit einem Editor Ihrer Wahl (z.B. vi/vim oder nano) eine Quelltextdatei `hs_2.c` mit folgendem Inhalt:

```
/* Aufgabe hs_2
   Mister Muster – Matr.Nr.: 575123456789 */

#include <stdio.h>

int main(void)
{
  printf("Hallo, Welt!");

  return 0;
}
```

Starten Sie anschließend, jetzt ohne Verwendung von **make**, den C-Compiler mit der Anweisung `cc hs_2.c` oder `gcc hs_2.c`

Beobachten Sie, welche Dateien jetzt erzeugt wurden und starten Sie das erzeugte ausführbare Programm.
Mit dem Zusatzparameter **-o** können Sie den Namen der zu erzeugenden ausführbaren Datei vorgeben.
Übersetzen Sie den Quelltext ein zweites Mal und lassen Sie jetzt die ausführbare Datei **hs_2** erzeugen.

Kopieren Sie jetzt aus den Vorlesungsunterlagen oder aus dem Projekt **hs_1** ein makefile in Ihr Projektverzeichnis **/hs_2**. Passen Sie die Datei an Ihre Erfordernisse an und übersetzen Sie anschließend Ihr Projekt unter Verwendung von **make**.
Um das neu angelegte Projekt in einem git-Repository zu verwalten können Sie, wenn Sie sich im Projektverzeichnis befinden, mit der Anweisung **git init** die benötigten Verwaltungsinformationen erzeugen.
Die ersten auf git init folgenden Schritte sind meist ein Zufügen der bereits vorhandenen Dateien (git add .) und ein erster commit.
Überprüfen Sie den Erfolg dieser Schitte mit **git status** bzw. **git log**.

Verfahren Sie so auch mit den folgenden Projekten so. Arbeiten Sie immer mit *make* und führen Sie nach jeder größeren Änderung oder Ergänzung in Ihrem Projekt ein commit durch.

Eine gute Einführung in GIT finden Sie auf der Seite
http://www-cs-students.stanford.edu/~blynn/gitmagic/intl/de/index.html

2 Bildschirm Aus- und Eingabe

2.1 Bildschirmausgabe

Zur Ausgabe von Texten am Bildschirm wird der Befehl *printf* verwendet.

In seiner einfachsten Version gibt dieser Befehl mit *printf("Text");* einen vorgegebenen Text am Bildschirm aus.

- Ergänzen Sie das Programm **hs_2** so, dass mit zwei *printf*-Anweisungen Ihr Name und Ihre Matrikelnummer ausgegeben werden.

Wenn Sie das so veränderte Programm starten werden Sie feststellen, dass die beiden Texte der *printf*-Anweisungen direkt hintereinander ausgegeben werden. Um einen Zeilenumbruch zu erreichen können Sie an einer beliebigen Stelle im Text die Zeichenfolge **\n** einfügen.

- Verändern Sie das Programm **hs_2** jetzt so, dass die beiden mit *printf* ausgegebenen Zeilen durch eine Leerzeile getrennt werden. Ergänzen Sie zudem eine Leerzeile vor und nach dem Text.

Weitere Steuerzeichen zur Textformatierung werden über sogenannte „Escape-Sequenzen“ ausgegeben.

Escape-Seq.	Ausgabe
\a	Bell (akustisches Signal)
\b	Backspace
\n	Neue Zeile
\r	Zeilenrücklauf (carriage return)
\t	Tabulator
\\	\ (backslash)
\"	" (Anführungszeichen)
\'	' (Hochkomma)
\nnn	Zeichen mit dem angegebenen Oktalwert ausgeben
\xhh	Zeichen mit dem angegebenen Hexadezimalwert ausgeben

2.2 Verwendung von Variablen

C kennt, wie Sie es schon aus Java kennen, eine Reihe von Datentypen zur Verwendung als Variablen.
Einige dieser Datentypen sind *char, int, long, float* und *double*.

- Erzeugen Sie ein neues Projekt **hs_3** (Verzeichnis, Makefile, C-Datei mit Basisprogramm)
- Definieren Sie in diesem Programm drei Variablen vom Typ int.
- Weisen Sie zwei dieser Variablen beliebige (sinnvolle) Werte zu.
- Speichern Sie das Produkt der beiden ersten Variablen in der dritten Variablen.

Wann spricht man in diesem Zusammenhang von deklarieren, definieren und wann von initialisieren?

Declaration: Wann wir die drei Variablen zum Program vorstellen.

Definition: Wann Speicherplatz zu den Variablen zugewiesen wird, indem man diese Variablen als ein Typ int definiert.

Initialisieren: Wann wir Werte für die drei Variablen zuweisen.

Wenn Sie Ihr Projekt bis zu dieser Stelle bearbeitet haben können Sie es mit *make* Übersetzen.
Der Compiler sollte Ihnen jetzt keine Fehlermeldung, sondern eine Warnung anzeigen. Diese Warnungen dienen dazu Sie auf mögliche Probleme in Ihrem Quellcode hinzuweisen.
Warnungen führen nicht dazu, dass der Quellcode nicht übersetzt werden kann. Trotzdem sollten Sie Warnungen nicht einfach ignorieren (oder sogar abschalten)
In diesem speziellen Fall hat der Compiler festgestellt, dass die Variable in der Sie das Resultat der Multiplikation speichern niemals verwendet wird.

Um das Resultat der Berechnung auszugeben (und damit auch die Warnung zu erledigen) verwenden wir eine erweitere Version des *printf*-Befehls.

Vorsicht! Die, vielleicht naheliegende, Verwendung von *printf* in der Form

```
int x = 3;
printf(x);
```

ist falsch!

Die Funktion *printf* ist definiert als

```
int printf (const char *format, ...);
```

Der erste Parameter der *printf*-Anweisung ist also immer eine Zeichenkette *(genauer: const char*).*
Der zurückgegebene Wert gibt dabei die Zahl der ausgegebenen Zeichen an.

Indem wir innerhalb dieser Zeichenkette eine entsprechende Kennung einfügen können wir Platz zum Einfügen einer Variablen reservieren. Einige Beispiele für solche Kennungen sind:

Kennung	Ausgabe	Beispiel
%c	Character	A
%d oder %i	Integer mit Vorzeichen	-392
%e oder %E	double im Format [-]d.ddd e±dd bzw. [-]d.ddd E±dd	3.9265e+2
%f	float im Format [-]ddd.ddd	392.65
%lf	double im Format [-]ddd.ddd	392.65
%g oder %G	Automatische Auswahl von %f oder %e	
%o	int als Oktalzahl ausgeben	610
%p	Adresse eines Pointers ausgeben	0028FF1C
%s	String ausgeben	Abc
%u	unsigned integer	7235
%lu	unsigned long integer	99999999
%x oder %X	int als Hexadezimalzahl ausgeben	3a
%%	Prozentzeichen ausgeben	%

Die in den Text einzufügende(n) Variable(n) werden nach dem Text, getrennt durch ein Kommata, angefügt.

Beispiel:

```
printf("Der Wert von x ist %i!",n);
```

würde für eine Variable *n* mit dem Inhalt 10 ausgeben

Der Wert von x ist 10!

- Ergänzen Sie das Programm **hs_3** so, dass das Resultat der Multiplikation ausgegeben wird in der Form „Das Produkt von X und Y ist Z“.

- Ersetzen Sie das Produkt durch einen Quotienten – was passiert mit Nachkommastellen?

- Ersetzen Sie den Variablentyp *int* bei allen drei Variablen durch den Typen *float* – wie werden Nachkommastellen jetzt behandelt? Was passiert wenn nur das Resultat vom Typ *float* ist?
- Wie verhält das Programm sich im letzten Fall bei den Schreibweisen x/y, (float)x/y, (float)(x/y) und (float)x/(float)y?

Um eine gewünschte Zahl von Nachkommastellen zu erreichen kann die Kennung %f entsprechend erweitert werden. %.4f würde zu einer Ausgabe von 4 Nachkommastellen führen.

- Verändern Sie Ihr Programm so, dass nur noch zwei Nachkommastellen ausgegeben werden.

2.3 Eingabe mit scanf

Zur Eingabe von Informationen zur Laufzeit kann in C die *scanf*-Anweisung benutzt werden.

VORSICHT! Die Funktion *scanf* ist nicht gegen Buffer-Overflows abgesichert. Aus sicherheitstechnischer Sicht ist die Verwendung von *scanf* daher bedenklich.

Die Funktion *scanf* ist definiert als

```
int scanf (const char *format, ...);
```

Der erste Parameter von *scanf* ist dementsprechend eine Zeichenkette mit einer Formatangabe zu den einzulesenden Informationen. Mögliche Formatangaben sind:

Kennung	Ausgabe
%d, %i	Integer mit Vorzeichen
%u	Integer ohne Vorzeichen
%e, %f, %g, %a oder %E, %F, %G, %A	Fliesskommazahl
%lf	Double-Fließkommazahl
%o	Oktalzahl
%s	String
%x	Hexadezimalzahl

Der (oder die) folgenden Parameter sind DIE ADRESSEN (Pointer) der einzulesenden Variablen.

Beispiel:

```
int  n;
scanf("%i",&n);
printf("Der eingelesene Wert war %i\n",n);
```

Dabei wird die Adresse einer Variablen (der Pointer) durch die Verwendung des Adressoperators & gebildet.

VORSICHT! Aus Gründen die später erläutert werden entfällt bei Variablen vom Typ String (array of char) dieser Operator.

Erweitern Sie in einem neuen Projekt **hs_4** das Programm zur Multiplikation von zwei Zahlen jetzt so, dass die zu multiplizierenden Zahlen vorher vom Benutzer eingegeben werden können. Programmieren Sie die Eingabe mit einer **scanf**-Anweisung und ignorieren Sie dabei zunächst den Rückgabewert von **scanf**.

Beobachten Sie, welche Meldung bei einer Übersetzung des Programms durch einen direkten Aufruf des C-Compilers ausgegeben wird. Vergleichen Sie die Ausgabe bei einem Übersetzen mit dem angepassten **makefile**.
Untersuchen Sie die Compiler-Optionen im Makefile (z.B. einzeln ausprobieren). Welche ist für die Ausgabe dieser Warnung verantwortlich?

-O3

Untersuchen Sie jetzt den Rückgabewert der **scanf**-Funktion bei richtigen und bei falschen Eingaben. Welche Information enthält dieser Rückgabewert?

Die Rückgabenwerte enthalten Information darüber, wie viele eingegeben Werte richtig eingelesen wurden. Das heißt, wenn beide Eingaben von Typ int (in diesem Fall) sind, wird "2" als Rückgabewert ausgegeben.

Warum wird bei der (falschen) Eingabe eines *String* oder *char* statt der ersten Zahl die Eingabe der zweiten Zahl übersprungen?

Scanf sucht nach einem Wert von dem Typ int(in diesem Fall) im EIngang-Buffer und wenn ein Charakter stattdessen kommt, wird der Charakter im Buffer hinterlassen. Ungewünschte Werte sind dann von dem Scanf in allen weiteren initialisierten Variablen in der Scanf Anweisung zugewiesen.

Werten Sie jetzt die Rückgabewerte der **scanf**-Anweisungen aus und ändern Sie das Programm so, dass das Resultat nur noch bei korrekten Eingaben angezeigt wird.

3 Variablen

3.1 Speicherbedarf und Genauigkeit von Variablen

Um den Speicherbedarf einer Variablen zu ermitteln kennt C die Anweisung sizeof().

- Erzeugen Sie ein neues Projekt **hs_5** in dem Sie den Speicherbedarf von Variablen der Typen *char, int, long, float* und *double* ermitteln. Geben Sie die Resultate in Form einer Tabelle am Bildschirm aus.

Die interne Darstellung von Variablen der Typen float und double (also von Fließkommazahlen) erfolgt nach dem IEEE754-Verfahren. Bei der Speicherung von Werten in diesem Verfahren können Ungenauigkeiten auftreten.

- Definieren Sie in Ihrem Programm zwei Variablen vom Typ *int* und eine Variable vom Typ *float*.
- Weisen Sie der ersten *int*-Variablen einen großen Wert zu (z.B. 1234567890)
- Kopieren Sie den Wert der *int*-Variablen in die *float*-Variable.
- Kopieren Sie den Wert der *float*-Variablen in die zweite *int*-Variable.
- Vergleichen Sie den ursprünglichen Wert in der ersten *int*-Variablen mit dem Wert der zweiten *int*-Variablen. Geben Sie beide Werte und die Differenz der Werte am Bildschirm aus.
- Beobachten Sie welche Warnungen/Fehlermeldungen der Compiler bei der Umwandlung mit und ohne Typecast am Bildschirm ausgibt.

Untersuchen Sie wieder die Compiler-Optionen im Makefile.
Welcher Parameter im *makefile* ist für die Ausgabe der Warnungen verantwortlich?

-03 -Wconversion

Um zu sehen, dass das Problem von Wertverlusten nicht nur bei extrem großen oder extrem kleinen Werten auftritt, weisen Sie einer Variablen vom Typ *float* den Wert 0.4 zu und geben Sie die Variable mit 12 Nachkommastellen am Bildschirm aus.

Wenn Sie die Ursache für dieses Problem verstehen wollen suchen Sie in Google nach *float applet* und sehen sich die Seite *IEEE754 Umrechner* an.

3.2 Ein vollständiges Programm

Schreiben Sie ein Programm **hs_6** das den Benutzer mit sinnvollen Meldungen zur Eingabe von folgenden Werten auffordert:

- Einem Startkapital (double)
- Einem jährlichen Zinssatz (double)
- Einer Zahl von Jahren (int)

Berechnen Sie anschließend in einer Schleife die Zinsen und Zinseszinsen und geben Sie die Resultate in Form einer Tabelle aus.
Dabei sollen in der ersten Spalte die Jahre, in der zweiten Spalte die Zinsen und in der dritten Spalte das aufsummierte Kapital ausgegeben werden.

3.3 Verwendung von statischen Variablen

Schreiben Sie ein Programm **hs_7** (oder ergänzen Sie das Zinsprogramm) mit einem Unterprogramm *zaehlen()*. Dieses Unterprogramm hat keine Übergabe- und keine Rückgabeparameter.

- Definieren Sie innerhalb von `zaehlen()` eine int-Variable und initialisieren Sie diese Variable mit dem Wert 0.
- Inkrementieren Sie den Wert der Variablen.
- Geben Sie jetzt innerhalb des Unterprogramms den Wert der Variablen aus.
- Rufen Sie dieses Unterprogramm mehrfach (2-3 mal) aus `main()` auf.

Welche Beobachtung machen Sie bezüglich der ausgegebenen Werte?

Wenn die Funktion angerufen wird, wird jedes Mal die int-Variable 'a' mit dem Wert 0 initialisiert und deshalb ist der Wert der Variable nur einmal inkrementiert. Als Resultat wird der Wert '1' jedes Mal ausgegeben.

Ergänzen Sie jetzt die Definition der Variablen um die Angabe ***static*** und wiederholen Sie den Versuch.
Welche Beobachtung machen Sie jetzt?

Jetzt gibt die 'Zaehlen' Funktion die Werte 1,2,3 usw, wann die Funktion angerufen wird. Das heißt, der Wert der int -Variable nach jedem Anruf der Funktion wird in diesem Fall nicht von dem Speicherplatz gelöscht.

4 Pointer

4.1 Pointer (Zeiger) in C

Ein Zeiger (oder eine Zeigervariable) ist eine Variable, die eine Speicheradresse enthält. Dabei handelt es sich um die Adresse einer anderen Variablen, einer Struktur oder einer Funktion.
Zum Verständnis von Zeigervariablen ist es wichtig ein Verständnis für die Speicherverwaltung eines C-Programms zu haben. Normalerweise wird der von einem C-Programm verwendete Speicher auf drei Arten verwaltet:

- Static/Global

In diesem Speicherbereich werden global definierte und statische Variablen gespeichert. Dieser Bereich wird beim Start des Programms belegt und bleibt über die gesamte Laufzeit des Programms erhalten. Der Zugriff auf globale Variablen ist uneingeschränkt aus dem gesamten Programm möglich. Auf als static definierte Variablen kann nur aus dem Kontext der Definition zugegriffen werden.

- Automatic

In diesem Bereich werden Variablen gespeichert, die zur Programmlaufzeit innerhalb eines Gültigkeitsbereichs (Funktion, Klammerebene) definiert werden. Die Lebensdauer dieser Variablen ist auf den Gültigkeitsbereich ihrer Definition beschränkt. Beim Verlassen des Gültigkeitsbereichs werden solche Variablen „automatisch“ gelöscht.

- Dynamic

Speicher dieses Typs wird zur Programmlaufzeit explizit alloziert und gelöscht. Ein Zeiger dient zum Zugriff auf Speicher in diesem Bereich.

4.2 Deklaration von Zeigervariablen (Pointern)

Ein Pointer in C wird normalerweise als sogenannter „typisierter Pointer“ deklariert. Die Deklaration enthält neben der eigentlichen Adresse auch eine Information über den Datentyp. (z.B. „Zeiger auf int“)
Die Deklaration erfolgt unter Verwendung des gewünschten Datentyps mit einem * (Asterisk). Die Verwendung des Leerzeichens zur Trennung ist dabei beliebig.

Beispiel:

```
int* pA;        // Zeiger auf int mit Namen pA
double *pB;     // Zeiger auf double mit Namen pB
char * pC;      // Zeiger auf char mit Namen pC
```

Legen Sie in Ihrem Arbeitsverzeichnis (`Documents/`) ein Projektverzeichnis `hs_8` an.
Erzeugen Sie in diesem Programm

- Eine Zeigervariable vom Typ „Zeiger auf *char*“.
- Eine Zeigervariable vom Typ „Zeiger auf *float*“.
- Eine Zeigervariable vom Typ „Zeiger auf *double*“.
- Eine Zeigervariable vom Typ „Zeiger auf *long double*“.

Ermitteln Sie mit Hilfe der Funktion *sizeof()* den Speicherbedarf dieser Zeigervariablen.
Vergleichen Sie das Resultat mit dem Speicherbedarf der Variablen aus Versuch 3.1.

Ermittelter Speicherbedarf:

	Variable	Zeiger auf Variable
char	1	8
float	4	8
double	8	8
long double	16	8

4.3 Zuweisung von Werten an Zeigervariablen

Eine Zeigervariable ist nach der Definition zunächst uninitialisiert. Zur Zuweisung eines Wertes (einer Adresse) kann der Adressoperator *&* verwendet werden.

Beispiel:

```
int i = 10;     // Definition und Wertzuweisung von int
int * pi;       // Definition eines Zeigers auf int
pi = &i;        // Zuweisung der Adresse von i an pi!
```

- Legen Sie ein neues Projekt `hs_9` an und definieren Sie innerhalb von *main()* drei *int*-Variablen *i1, i2* und *i3* und lassen Sie sich mit *printf()* die Adressen der Variablen anzeigen.
- Ergänzen Sie eine vierte Variable *i4* vom Typ *static int* und lassen Sie sich die Adresse mit *printf()* anzeigen.
- Ergänzen Sie eine globale Variable *i5* gleichen Typs und zeigen Sie die Adresse an.

Wie verhalten sich die Adressen unter Berücksichtigung des Speicherbedarfs einer *int*-Variable und des Speichertyps (automatic/static).

Die Adressen der Zeigervariablen vom Speichertyp 'automatic' verändern sich komplett mit jeder Ausführung des Programms, aber die letzte zwei Stellen der Adressen vom Typ 'static' bleiben gleich bei jeder Ausführung.

Als Kennzeichen für einen „leeren“ oder uninitialisierten Pointer wird üblicherweise der Wert NULL verwendet.

- Ergänzen Sie eine Zeigervariable *i6* vom Typ „Zeiger auf int“ und weisen Sie dieser Variablen den Wert *NULL* zu.

Welche Informationen (nicht welche Werte!) werden durch die folgenden Anweisungen ausgegeben?

```
printf("I6: %p",&i6);
```

Die Adresse, die der Zeigervariable selbst zugewiesen wird, wird angezeigt.

```
printf("I6: %p",i6);
```

Die Adresse, die die Zeigervariable enthält(in diesem Fall ein Null),wird angezeigt.

4.4 Dereferenzieren von Zeigervariablen

Um unter Verwendung eines Pointers auf den Inhalt des adressierten Speicherbereichs zuzugreifen ist es nötig den Zeiger zu „dereferenzieren“
Der dafür erforderliche *Dereferenzierungsoperator* ist ebenfalls der * (Asterisk) (Bitte nicht verwechseln!)

Beispiel:

```
int i;          // neue (uninitialisierte) int-Variable
int * pi = &i; // neuer Zeiger und Zuweisen der Adresse von i
*pi = 10;       // Wertzuweisung unter Verwendung des Zeigers
```

5 Funktionsparameter

5.1 Übergabeparameter von Funktionen

Ohne weitere Angaben werden die Übergabeparameter einer Funktion *„by Value“* übergeben. Unter Verwendung von Zeigern als Übergabeparameter ist eine Parameterübergabe *„by Reference“* realisierbar.

- Starten Sie ein neues Projekt **`hs_10.`** Definieren Sie in diesem Programm zwei Variablen vom Typ *int* und weisen Sie beliebige, verschiedene Werte zu.
- Ergänzen Sie eine Funktion *tausche()* mit dem Rückgabetyp *void*. Dieser Funktion sollen die beiden definierten Variablen (mit Zeigern) übergeben und innerhalb der Funktion getauscht werden.
- Rufen Sie die Funktion *tausche()* in *main()* auf und zeigen Sie anschließend mit *printf()* in main den Inhalt der Variablen an.

6 Arrays

6.1 Zeiger auf Arrays

Bei einem Array handelt es sich um eine Sammlung von Elementen gleichen Typs die im Speicher zusammenhängend abgelegt sind. Arrays werden unter Verwendung von eckigen Klammern [ ] definiert und können mehrere Dimensionen haben.

Beispiel:

```
int a1[5];     // Definition eines int-Arrays mit 5 Elementen
int a2[3][3];  // Zweidimensionales Array
int a3[3] = {1,2,3};     // Definition mit Wertzuweisung
int a4[2][3]={{7,6,5},{1,2,3}};   // Mehrdimensional mit Wert
int a5[100] = {0};  // Initialisierung aller Elemente mit 0
```

Arrays und Pointer sind zwar nicht identisch, haben aber viele Gemeinsamkeiten. Um unter Verwendung eines Pointers auf den Inhalt eines Arrays zuzugreifen kann die Array-Variable direkt, ohne Verwendung des Adressoperators, einer Zeigervariablen zugewiesen werden. Alternativ dazu kann dem Zeiger die Adresse des ersten Array-Elements zugewiesen werden.

Beispiel:

```
int a[3];           // Array anlegen
int* pa = a;        // Zeiger auf das Array
int* pa2 = &a[0];   // Zweiter Zeiger
```

Unter Verwendung von Zeigerarithmetik kann so über einen Zeiger auf die Elemente des Arrays zugegriffen werden.
Die umgekehrte Zuordnung

```
int b[3] = pa;
```

ist allerdings **nicht zulässig**!
Trotzdem kann über den Zeiger auch in Array-Schreibweise auf den Speicher zugegriffen werden.

```
int i = pa[2];
```

6.2 Anordnung der Arrayelemente im Speicher

Versuchen Sie die Anordnung der Elemente bei mehrdimensionalen Arrays im Speicher zu ermitteln.

- Starten Sie ein neues Projekt **hs_11**.
- Definieren Sie ein dreidimensionales Array mit 3x3x3 Elementen.
- Weisen Sie den Elementen des Arrays durch Zugriff **über einen Zeiger** in einer for-Schleife über alle Array-Elemente fortlaufende Werte zu.
- Greifen Sie anschließend in drei geschachtelten for-Schleifen **in Array-Schreibweise** auf das Array zu und geben Sie die Arrayelemente aus.

Wie ist die Anordnung der Elemente im Speicher?

a[0][0][0] = 0; a[0][0][1] = 1; a[0][0][2] = 2; usw. Alle dieser Elemente sind sequentiell im Speicher angeordnet.

- Geben Sie den Inhalt der Arrayelemente [0][0][5] und [3][3][10] aus.
- Ändern Sie die äußere for-Schleife auf 4 Schleifendurchläufe ab.

Welche Meldungen gibt der Compiler in diesen beiden Fällen?

Fall 1: Die Array-Subscripte 3, 5 und 10 liegen über den Array-Grenzen.
Fall 2: Interation 3 ruft undefiniertes Verhalten auf [-Waggressive-loop-optimizations]

Was passiert beim Überschreiten einer der definierten Arraygrenzen?

Überlauf von Werten in den Stack, Werte werden zu mehreren Elementen als 3x3x3 zugewiesen.

6.3 Typecasting von Zeigertypen

Durch *Typecasting* ist es möglich die Typinformation eines typisierten Zeigers zu ändern. Das folgende Programm macht Gebrauch von dieser Möglichkeit. Untersuchen Sie das Unterprogramm *Ausgabe* und versuchen Sie herauszufinden welche Funktion dieses Unterprogramm hat.

```
void ausgabe(unsigned int z)
{
  int n;
  for (n=3;n>=0;n--)
  {
    printf("%u",*(((unsigned char*)&z)+n));
    if (n>0) printf(".");
  }
}

ausgabe(3232235779);
```

Welche Aussage können Sie anhand dieses Programms über die Anordnung der Bytes eines *unsigned int* im Speicher treffen?

Maximal 255 Bytes können in jeder Addresse für einen unsigned Int gespeichert werden. Zum Beispiel wird bei einem Übergabewert von dem maximalen Wert von einem unsigned Int (0 - 2^32 = 0 - 4294967295) das Ergebnis 255.255.255.255 angezeigt.

6.4 Dynamische Speicherverwaltung

Die dynamische Reservierung von Speicher wird in C durch die Funktion *malloc()* aus <*stdlib.h*> ermöglicht. Diese Funktion ist deklariert als

```
void *malloc(size_t size);
```

Durch die Verwendung von *malloc()* kann der Programmierer zur Laufzeit des Programms einen zusammenhängenden Speicherbereich vom System anfordern. Dabei wird *malloc()* als Übergabewert die Zahl der zu reservierenden Bytes angegeben. Die Zahl der erforderlichen Bytes kann mit Hilfe der Funktion *sizeof()* aus den bekannten Typen ermittelt werden.

Alternativ dazu kann auch die Funktion *calloc()* verwendet werden. Diese Funktion ist deklariert als

```
void *calloc(size_t count, size_t size);
```

Bei der Verwendung von *calloc()* wird ein zusammenhängender Speicherbereich von *count* Elementen der Größe *size* angefordert.
Dieser Speicherbereich wird mit dem Wert 0 initialisiert!

Der Rückgabewert von *calloc()* oder *malloc()* ist ein Zeiger auf den reservierten Speicher. Dieser Zeiger ist untypisiert (void*), kann aber ohne ein Typecast in den gewünschten Typ umgewandelt werden. (das ist in ANSI C++ nicht so. Hier ist ein Typecast erforderlich.)
Falls die Reservierung von Speicher fehlschlägt wird von *calloc()* oder *malloc()* ein NULL-Zeiger zurückgegeben.

Der so reservierte Speicher muss unbedingt durch die Anweisung *free()* wieder freigegeben werden.

Ein „vergessenes" free() ist ein schwerwiegender Fehler und kann Folgen bis hin zu einem Systemabsturz haben!

Beispiel:

```
double* pa_d = (double*) malloc(1000 * sizeof(double));
int* pa_i = (int*) calloc(1000,sizeof(int));
free(pa_d);
free(pa_i);
```

Test verschiedener Schreibweisen beim Zugriff auf reservierten Speicher:

- Starten Sie ein neues Projekt `hs_12`.
- Reservieren Sie mit *malloc()* oder *calloc()* Platz für ein Array von 10000 Elementen vom Typ *double*.
- Überprüfen Sie anhand der Zeiger-Rückgabe ob die Reservierung erfolgreich war und geben Sie eine entsprechende Meldung aus.
- Speichern Sie in einer Schleife fortlaufende Werte in diesem Speicher. Greifen Sie dabei mit der Array-Schreibweise auf den Speicher zu.
- Lesen Sie über den Pointer einen beliebigen dieser Werte aus und zeigen Sie den Wert mit *printf()* an.

6.5 Auswirkung der Zeigertypisierung

Im letzten Versuch haben wir gesehen, dass Zeiger in C eine Typinformation beinhalten. Diese Typisierung erleichtert den Umgang mit Zeigern auf verschiedene Datentypen, hat jedoch erhebliche Auswirkungen auf das Rechnen mit Zeigern bzw. Zeigervariablen.
Legen Sie in Ihrem Arbeitsverzeichnis (`Documents/`) ein Projektverzeichnis `hs_13` an und beobachten Sie folgendes Verhalten.

- Definieren Sie eine Variable *var* vom Typ *int*.
- Definieren Sie eine Zeigervariable *pvar* vom Typ „Zeiger auf *int*“.
- Initialisieren Sie *pvar* mit der Adresse von *var*.
- Vergrößern Sie die Zeigervariable *pvar* um 1.
- Speichern Sie in einer Variablen vom Typ *long int* die Differenz der Adresse von *var* (*&var*) und *pvar*.
- Lassen Sie sich diese Differenz anzeigen (%ld).
- Lassen Sie sich in zwei weiteren Zeilen direkt (%p) die Adressen &var und pvar anzeigen.
- Wiederholen Sie dieses Experiment mit den Datentypen *long double* und *char*.

Welche Beobachtung machen Sie bezüglich der berechneten Differenz der beiden Adressen und der Differenz der beiden angezeigten Zeiger?

Die berechnete Differenz und die Differenz der beiden angezeigten Zeiger sind gleich für Typ 'char'. (Differenz = 1) Die berechnete Differenz und die Differenz der beiden angezeigten Zeiger sind unterschiedlich für Typ 'long double'. (berechnete Differenz = 1, aber die Differenz der beiden angezeigten Zeiger ist 10)

Sie können sich die Differenz der Adressen auch direkt anzeigen lassen indem Sie ein Typecast der beiden Zeiger auf den Datentyp *long int* durchführen und die Differenz berechnen.

Nach dem inkrementieren soll mit dem Zeiger *pvar* durch die Anweisung
`*pvar = 123;`
eine Integer-Zahl gespeichert werden. Erläutern Sie das hierbei auftretende Problem.

Wir haben nur eine Variable 'var' von Typ int definiert aber kein Array von Typ int. Daher wird bei der Anzeige von *pvar die Fehlermeldung angezeigt, dass Array-Index[1] außerhalb der Array-Grenzen liegt, weil 'var' kein Array ist.

6.6 Arrays als Funktionsparameter

Bei der Übergabe eines Arrays als Funktionsparameter sind einige Besonderheiten zu beachten.
Legen Sie in Ihrem Arbeitsverzeichnis (`Documents/`) ein Projektverzeichnis `hs_14` an.

- Definieren Sie ein Array *arr* mit 10 Elementen vom Typ *int*.
- Ermitteln Sie mit der *sizeof()*-Funktion die Größe des Arrays und zeigen Sie diese mit *printf()* an.
- Füllen Sie das Array *arr* mit fortlaufenden Werten ab 1. Verwenden Sie dazu eine *for*-Schleife und legen Sie das Ende der Schleife mit der sizeof()-Funktion fest.
- Programmieren Sie eine Funktion *ausgabe()*. Diese Funktion soll als Übergabeparameter das Array *arr* übergeben bekommen.
- Ermitteln Sie innerhalb dieser Funktion mit der *sizeof()*-Funktion die Größe des Übergabeparameters und zeigen Sie diese mit *printf()* an.
- Überprüfen Sie den Einfluss der verschiedenen Schreibweisen *int a[10], int a[], int * a* des Übergabeparameters auf das Resultat.
- Versuchen Sie die Funktion Ausgabe so zu erweitern, dass die Array-Elemente eines übergebenen *int*-Arrays beliebiger Länge in der Funktion ausgegeben werden.

Welche Informationen werden einer Funktion bei einem Array als Übergabeparameter übergeben, welche nicht?

Der Typ des Arrays wird einer Funktion übergeben. Die Funktion bekommt aber keine Information über die Zahl der Elemente in dem übergebenen Array.

7 Funktionszeiger

7.1 Zeiger auf Funktionen (function pointers)

Ein Zeiger auf eine Funktion ist, ähnlich wie ein Zeiger auf eine Variable, ein Zeiger der die Adresse einer Funktion im Arbeitsspeicher enthält. Entsprechend dem Datentyp eines typisierten Zeigers auf eine Variable enthält der Zeiger auf eine Funktion zusätzliche Informationen zum Rückgabewert und zu den Übergabeparametern der adressierten Funktion.

Um einen Zeiger auf eine Funktion zu definieren nehmen Sie die Deklaration der Funktion und ersetzen den Funktionsnamen durch einen Ausdruck der Form (*pf).

Beispiel:

```
void test(char c){ printf("%c“,c);}     // Beispielfunktion 1
int summe(int a, int b) {return a+b;}  // Beispielfunktion 2

void (*pf1) (char); // Zeiger auf eine Funktion mit einem
                    // Übergabewert vom Typ char
                    // ohne Rückgabewert (wie Bsp.1)

int (*pf2) (int,int); // Zeiger auf eine Funktion mit zwei
                      // Übergabewerten vom Typ int und
                      // einem Rückgabewert vom Typ int (Bsp.2)

pf1 = test;    // Zuweisen der Adresse von test an pf1
pf2 = summe;   // Zuweisen der Adresse von summe an pf2
pf2 = &summe;  // Zweite, gleichwertige Schreibweise

(*pf1)('#');   // Aufrufen von test über den Zeiger pf1
               // mit dem Übergabewert '#'.
int s = (*pf2)(3,4);// Aufrufen von summe über den Zeiger pf2
                    // mit den Übergabewerten 3 und 4.
```

Clonen Sie von dem bekannten GIT-Server das Projekt **`AID-Labor/hs_15.git`**.
In dem hier vorgegebenen Programm *hs_15.c* finden Sie eine Sortierfunktion *bubble()*. Diese Funktion wird aus *main()* mit einem Array als Parameter aufgerufen, das vorher mit Zufallszahlen gefüllt wurde. Anschließend wird das Resultat der Sortierung ausgegeben.

- An einigen Stellen in diesem Programm finden Sie angefangene, leere Kommentare (//...). Ergänzen Sie zunächst an diesen Stellen sinnvolle Kommentare um zu zeigen, dass Sie den Programmablauf verstanden haben.
- Definieren Sie in *main()* einen Funktionszeiger passend zum Typ der vorgegebenen Vergleichsfunktion.
- Initialisieren Sie diese Zeigervariable mit der Adresse der vorhandenen Vergleichsfunktion.
- Ändern Sie die Funktion *bubble()* so ab, dass die Vergleichsfunktion über einen zusätzlichen dritten Übergabeparameter, den aus *main()* übergebenen Zeiger auf die Vergleichsfunktion, aufgerufen wird.
- Testen Sie die Funktionsfähigkeit des so veränderten Programms.

Nachdem Sie die Vergleichsfunktion über die Einführung des Zeigers aus der Sortierfunktion ausgelagert haben können Sie jetzt, ohne eine Änderung an der Funktion *bubble(),* das Vergleichskriterium der Sortierfunktion ändern.

Um diese Möglichkeit zu testen versuchen wir im nächsten Schritt die Zufallszahlen aufsteigend nach ihrer Quersumme zu sortieren.

Die Quersumme *qs(z)* einer positiven Zahl *z* berechnet sich (rekursiv) als:

qs(z) = z für z<10
qs(z) = z%10 + qs(z/10) für z >=10

- Erweitern Sie das vorhandene Programm um eine Funktion *qs()* zur Berechnung der Quersumme.
- Geben Sie in der Funktion *ausgabe()* in einer dritten Spalte die Quersumme der angezeigten Zahlen aus.
- Ergänzen Sie eine zweite Vergleichsfunktion *compare_qs()* zum Vergleich der Quersummen zweier übergebener Zahlen.
- Ändern Sie zum Test das Programm so ab, dass die neue Vergleichsfunktion über einen entsprechenden Funktionszeiger aufgerufen wird.
- Ergänzen Sie zum Abschluss in *main()*eine Abfrage (mit *scanf*) mit der die gewünschte Vergleichsfunktion ausgewählt werden kann. Dabei dürfen Sie die Warnung den Rückgabewert von *scanf* betreffend ausnahmsweise ignorieren.

Wie muss eine Zeigervariable mit dem Variablennamen *pf_cv* definiert werden, die einen Funktionszeiger auf die Funktion *compare_value()* speichern soll?

```
int (*pf_cv)(int ,int);
pf_cv = compare_value;
```

8 Strukturen

8.1 Der Datenverbund struct

Zusätzlich zu den vorgegebenen Datentypen besteht in C die Möglichkeit eigene kombinierte Datentypen mit dem Schlüsselwort *struct* anzulegen. Im Gegensatz zu Arrays, in denen nur Daten gleichen Typs zusammengefasst werden, besteht bei *struct* die Möglichkeit auch verschiedene Datentypen miteinander zu kombinieren.

Beispiel:

```
struct Geo
{
  double dLatitude;
  double dLongitude;
  float fAltitude;
};
```

Auch die Verwendung von Arrays und Strukturen innerhalb von *struct* sind erlaubt.

Beispiel:

```
struct Messwert
{
  int iMessungNummer;
  double dLuftdruck;
  float fTemperatur[3];
  struct Geo Position;
};
```

Um mit einer so definierten Struktur zu arbeiten ist es erforderlich ein Element dieser Struktur anzulegen. Dabei wird ähnlich wie bei der Definition einer Variablen vorgegangen:

```
struct Messwert mess1;     // definiert ein Element von
                           // Messwert mit dem Namen mess1
struct Messwert mess[5];  // 5 Elemente in einem Array

struct Geo g1 = {51.44786,7.27069,128.0f} // mit Wertzuweisung
```

Der Zugriff auf die einzelnen Elemente von Strukturen erfolgt mit der "Punkt-Schreibweise".

```
mess1.iMessungNummer = 1;
mess1.dLuftdruck = 2.03;
mess1.fTemperatur[1] = 18.3f;
mess1.Position.fAltitude = 123.45f;

mess[0].iMessungNummer = 0;
mess[1] = mess1;
```

Der einzige für Strukturen erlaubte Operator ist der Zuweisungsoperator. Mathematische und logische Operatoren und Vergleiche sind für Strukturen nicht definiert. Auch eine Wertzuweisung innerhalb der Strukturdeklaration ist nicht erlaubt.

Bei der Zuweisung wird eine vollständige Kopie der Struktur erzeugt. Darin sind gegebenenfalls auch die Elemente von in der Struktur vorhandenen Arrays enthalten.

Legen Sie in Ihrem Arbeitsverzeichnis (`Documents/`) ein Projektverzeichnis `hs16` an.

- Deklarieren Sie zunächst die hier beschriebenen Strukturen *Geo* und *Messwert*.
- Definieren Sie ein Element *messung* vom Typ *Messwert*. Versuchen Sie dabei dieses Element bei der Definition direkt mit (beliebigen) Werten zu initialisieren.
- Lassen Sie sich mit *sizeof()* den Platzbedarf von *messung* und den Platzbedarf von *messung.Position* anzeigen.
- Ergänzen Sie im Kopf Ihres Programms die Anweisung *#pragma pack(1)* und führen Sie das Programm erneut aus.

Wie ergibt sich der Speicherbedarf einer Struktur vom Typ *Messwert*? Welche Änderung wird durch das *packed* - Attribut erreicht?

Ohne das packed-Attribut:

sizeof(messung) = 56
sizeof(messung.Position) = 24

Mit dem packed-Attribut:
sizeof(messung) = 44
sizeof(messung.Position) = 20

Das packed-Attribut hat den Speicherbedarf von der Struktur reduziert.

- Ergänzen Sie ihr Programm um eine Funktion *ausgabe()*. Diese Funktion soll mit einem *Messwert* als Übergabeparameter aufgerufen werden und soll alle in diesem *Messwert* enthaltenen Daten anzeigen.
- Lassen Sie sich außerdem innerhalb dieser Funktion mit *sizeof()* den Speicherbedarf des in *Messwert* enthaltenen Arrays anzeigen.
- Verändern Sie den Inhalt eines der Array-Elemente von *fTemperatur* innerhalb der Funktion *ausgabe()*.
- Lassen Sie sich in *main()* nach dem Aufruf von *ausgabe()* den Wert des veränderten Array-Elements anzeigen.

Welche Beobachtung machen Sie? Wie unterscheidet sich dieses Verhalten von der Übergabe eines einzelnen Arrays als Funktionsparameter?

Die Veränderung eines Wertes in dem Array fTemperatur wird beim Anzeigen des Arrays in der main Funktion nicht reflektiert(der ursprungliche Wert wird in der main Funktion angezeigt). Bei einer einzelnen Array-Übergabe zu einer Funktion kann aber der veränderte Wert innerhalb der Funktion in der main Funktion angezeigt werden.

Auch das Anlegen von Zeigern auf eine Struktur ist möglich

```
Messwert* pMess = &mess1;
```

Der Zugriff auf die mit dem Zeiger *pMess* referenzierten Daten erfolgt dann durch

```
printf("%d\n",(*pMess).iMessungNummer);
```

oder einfacher durch die Pfeil-Schreibweise

```
printf("%d\n",pMess->iMessungNummer);
```

- Erzeugen Sie eine Kopie der Funktion *ausgabe()* mit dem neuen Namen *ausgabe_p().*
- Verändern Sie diese neue Funktion so, dass jetzt ein Zeiger auf eine *Messwert*-Struktur übergeben und verwendet wird. Dabei soll die Funktionalität von *ausgabe_p()* unverändert der von *ausgabe()* entsprechen.

Welche Auswirkung hat jetzt die Veränderung von Strukturelementen innerhalb der Funktion?

Die Veränderung von dem Array-Wert kann jetzt in der main Funktion angezeigt werden.

8.2 Strukturen als Rückgabewerte von Funktionen

Bisher konnten Sie als Rückgabewert einer Funktion nur einen einzelnen Wert oder einen Zeiger auf ein Array zurückgeben. Durch die Verwendung einer Struktur als Rückgabetyp wird es möglich, dass Sie auch komplexe Daten zurückgeben.

- Erweitern Sie das vorhandene Programm um eine Funktion *set_mw().* Diese Funktion soll eine Struktur vom Typ Messwert zurückgeben. Übergeben Sie der Funktion den Wert für *iMessungNummer* und ergänzen Sie innerhalb der Funktion beliebige restliche Werte.
- Testen Sie diese Funktion indem Sie innerhalb von *main()* einer neuen *Messwert*-Variablen den Rückgabewert zuweisen und diese Variable anschließend mit *ausgabe()* anzeigen.

9 Listen

9.1 Einfach verkettete Listen

Um eine große Zahl von Messungen zu speichern soll mit einer erweiterten Struktur vom Typ *Messwert* eine verkettete Liste aufgebaut werden.
Kopieren Sie dazu das bisher bearbeitete Projekt **hs16** in ein neues Projekt **hs17**.

- Ergänzen Sie zunächst die vorhandene Struktur *Messwert* um eine für die Verkettung der Liste benötigte Zeigervariable *pNext*.
- Definieren Sie innerhalb der Funktion *main()* eine Zeigervariable mit dem Namen *pStart* als „Anfang“ der verketteten Liste und initialisieren Sie diese Variable mit dem Wert *NULL*.

Zu der so vorbereiteten (aber noch leeren) verketteten Liste sollen jetzt Elemente hinzugefügt werden. Dazu muss zunächst für jedes neue Element der erforderliche Speicherplatz mit *malloc* oder *calloc* reserviert werden. Nachdem der reservierte Speicherplatz mit den entsprechenden Daten „gefüllt“ ist kann das Element zu der vorhandenen Liste hinzugefügt werden. Diese Arbeitsschritte sollen in einer neuen Funktion *addToList()* zusammengefasst werden.

- Ergänzen Sie eine Funktion *addToList()* ohne Rückgabewert. Diese Funktion soll als Übergabeparameter den „Anfang“ der verketteten Liste (*pStart*) und ein zuzufügendes Element übergeben bekommen.
- Der Rückgabewert der Funktion (vom Typ *int*) soll mit dem Wert 0 das erfolgreiche Zufügen zur Liste anzeigen. Der Rückgabewert -1 soll einen Fehler beim Zufügen signalisieren.
- Reservieren Sie mit *malloc* den für ein Listenelement benötigten Speicher. Falls kein Speicher reserviert werden konnte geben Sie -1 zurück und beenden Sie die Funktion.
- Kopieren Sie das übergebene Element in den mit *malloc* reservierten Speicher.
- Ändern Sie den Verkettungszeiger im neuen Element so, dass dieser Zeiger den (alten) Wert von *pStart* enthält.
- Speichern Sie den Zeiger auf das neu erzeugte Element in *pStart*.
- Geben Sie als Erfolgsmeldung den Wert 0 zurück.
- Testen Sie diese Funktion indem Sie aus *main()* eines der bisher erzeugten Elemente zur Liste zufügen.
- Lassen Sie sich zur Kontrolle nach dem Zufügen den Inhalt von *pStart* anzeigen.

Welchen Typ muss der Übergabeparameter für den Listenanfang haben? Warum?

Um zu überprüfen ob das Zufügen zur Liste erfolgreich war soll jetzt eine Funktion zum Ausgeben der Liste programmiert werden.

- Ergänzen Sie eine Funktion *printList().* Diese Funktion soll als Übergabeparameter den Listenanfang übergeben bekommen.
- Durchlaufen Sie innerhalb der Funktion mit einer geeigneten Schleife die gesamte Liste und geben Sie mit der Funktion *ausgabe_p()* die einzelnen Listenelemente aus.
- Zeigen Sie am Ende die Anzahl der Elemente in der Liste an.
- Kontrollieren Sie die Funktion indem Sie einige Listenelemente zufügen und dann die Liste anzeigen lassen.

Welchen Typ muss jetzt der Übergabeparameter für den Listenanfang haben?

In welcher Reihenfolge werden die Listenelemente ausgegeben? Warum?

Welche Ergänzung wäre nötig um nicht die gesamte Liste, sondern ein Element mit einer bestimmten *iMessungNummer* zu suchen und auszugeben?

Um die Arbeit mit der verketteten Liste abzuschließen soll noch eine Funktion zum Löschen der Liste programmiert werden.

- Ergänzen Sie eine Funktion *emptyList().* Diese Funktion soll als Übergabeparameter den Listenanfang übergeben bekommen.
- Durchlaufen Sie innerhalb der Funktion die gesamte Liste und geben Sie den für die einzelnen Listenelemente reservierten Speicher mit *free()* wieder frei.
- Setzen Sie am Ende den Zeiger auf den Listenanfang wieder auf den Wert *NULL*.