PR3 - Übungsblatt A.6

(Stand 2021-11-03 11:40)

Prof. Dr. Holger Peine
Hochschule Hannover
Fakultät IV – Abteilung Informatik
Raum 1H.2.60, Tel. 0511-9296-1880
Holger.Peine@hs-hannover.de

Thema

Funktionen

Termin

Ihre Arbeitsergebnisse zu diesem Übungsblatt führen Sie bitte bis zum 26.11.2021 vor.

1 Implizite Funktionsdeklaration (1 Punkt)

basiert auf Vorlesung bis einschl. Abschnitt 6.a

Betrachten Sie das folgende Programm (Quelltext auch in den Anlagen als PR3_U_A.6.2_implicit.c). Dieses verdeutlicht die Gefahr impliziter Funktionsdeklarationen:

```
#include <stdio.h>
#include <string.h>

int main(void) {
  char text[]= "HALLO WELT!"; /* lokal im main-Stackframe */
  printf("main(): text= %s\n", text);
  printf("main(): Adressbereich= %p-%p\n", (void*)text, (void*)&text[strlen(text)]);
  sub();
  printf("main(): text= %s\n", text);
  printf("main(): Adressbereich= %p-%p\n", (void*)text, (void*)&text[strlen(text)]);
  printf("Ende main\n");
  return 0;
}

struct S {
  char c[64];
};

int sub(struct S s) {
  int k;
  for (k=63; k>=0; k--) {
    printf("sub(): %3d (%p): %02X %c\n",
         k, s.c+k, (unsigned char)s.c[k], s.c[k]);
  }
  strcpy(s.c, "Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed di");
  return 0;
}
```

Die Funktion `sub` nimmt als Parameter einen Struct in Kopie entgegen (call by value) und gibt den Inhalt des Structs (ein Array von 64 Zeichen) aus. Dann befüllt `sub` das Array mit neuem Inhalt (exakt 63 Zeichen plus 0-Terminator) und kehrt zurück. Da die Befüllung lediglich im `sub`-Stackframe stattfindet, sollte die Befüllung keine Außenwirkung haben, d. h. die aufrufende `main`-Funktion sollte den Text „Lorem ipsum…“ nicht zu Gesicht bekommen.

Nun ist die Funktion `sub` **unterhalb** von `main` deklariert, so dass der C-Compiler zunächst eine implizite Funktionsdeklaration ("`int`-Funktion mit unbekannten Parametern") vornimmt. Die `main`-Funktion ruft `sub` auf, allerdings ohne Parameter, weil der Parameter schlicht vergessen wurde – der Compiler erkennt dies nicht als Fehler, weil der Aufruf zur implizit angenommen Funktionsdeklaration von `sub` passt (denn "unbekannte Paraeter" umfasst ja auch "keine Parameter"). Vor dem Aufruf von `sub` geben wir den Inhalt eines kurzen

Zeichenarrays `text` aus und danach ebenfalls. Da der `sub`-Aufruf in keinem Zusammenhang mit dem `text`-Array steht, sollten beide Ausgaben identisch sein.

Wir übersetzen das Programm wie folgt:

```
gcc -std=c99 implicit.c
```

Wir erhalten keine Fehler oder Warnungen. Es handelt sich also um ein gültiges C-Programm.

Wir starten das Programm und erhalten (je nach Plattform) etwa folgende Ausgabe:

```
main(): text= HALLO WELT!
main(): Adressbereich= 0x28ac54-0x28ac5f
sub():  63 (0x28ac7f): 61 a
sub():  62 (0x28ac7e): 27 '
...
sub():  34 (0x28ac62): 00
sub():  33 (0x28ac61): 00
sub():  32 (0x28ac60): 2F /
sub():  31 (0x28ac5f): 00
sub():  30 (0x28ac5e): 21 !
sub():  29 (0x28ac5d): 54 T
sub():  28 (0x28ac5c): 4C L
sub():  27 (0x28ac5b): 45 E
sub():  26 (0x28ac5a): 57 W
sub():  25 (0x28ac59): 20
sub():  24 (0x28ac58): 4F O
sub():  23 (0x28ac57): 4C L
sub():  22 (0x28ac56): 4C L
sub():  21 (0x28ac55): 41 A
sub():  20 (0x28ac54): 48 H
sub():  19 (0x28ac53): 61 a
...
sub():   1 (0x28ac41): 20
sub():   0 (0x28ac40): 92 ▒
main(): text= t amet, consetetur sadipscing elitr, sed di
main(): Adressbereich= 0x28ac54-0x28ac7f
Ende main
```

Wir können zweierlei beobachten:

- Die Funktion `sub` gibt den Text `HALLO WELT!` zeichenweise zwischen Position[1] 20 und 30 aus, obwohl sie eigentlich keinen Zugriff auf den `main`-Stackframe haben dürfte.
- Die Ausgabe des Textes am Ende von `main` offenbart, dass `sub` schreibend auf den `main`-Stackframe zugegriffen hat! Die Funktion `sub` hat einfach den Text `HALLO WELT!` mit einem eigenen Text überschrieben.

Beide Beoabachtungen sind besorgniserregend. Nur weil wir

- die Deklaration von `sub` unterhalb von `main` erstellt haben und
- den Parameter beim Aufruf vergessen haben,

ist der Stack offenbar vollkommen durcheinander geraten. Auf einem Windows XP-PC erhalte ich sonst keinen Hinweis auf ein Fehlverhalten des Programms. Es endet ganz normal. Die Fehlersuche kann hier sehr aufwändig werden. Auf einem Linux-PC und einem Windows 7-PC (oder neuer) erhalte ich wenigstens noch einen Absturz:

```
*** stack smashing detected ***: ./a.out terminated
Segmentation fault
```

der mich hellhörig werden lässt.

Ihre Aufgabe: Zeichnen Sie ein Abbild des Stacks, so wie er sich auf Ihrem System darstellt (mit genau den Speicheradressen, die Sie in Ihrer Ausgabe beobachten). Erläutern Sie

[1] Auf Ihrem System können das auch andere Positionen sein, z. B. direkt ab Position 0.

anhand des Bildes, was das Problem an diesem Stack ist und wie dieses Problem enstanden ist.

Um einen solchen Fehler gar nicht erst entstehen zu lassen, rate ich dringend, dass Sie Ihre Programme grundsätzlich mit der Option `-Wall` übersetzen. In unserem Fall ergibt sich dann:

```
$ gcc -std=c99 -pedantic-errors -Wall implicit.c
implicit.c: In function 'main':
implicit.c:8: warning: implicit declaration of function 'sub'
```

Wir erhalten also bereits zur Compilezeit einen Hinweis auf den Fehler. Wenn wir nun versuchen, den Fehler durch Verschieben der Funktion `sub` nach oberhalb von `main` zu korrigieren, erhalten wir beruhigenderweise auch den nächsten Compilerfehler:

```
$ gcc -std=c99 -pedantic-errors -Wall implicit.c
implicit.c: In function 'main':
implicit.c:22: error: too few arguments to function 'sub'
```

Wir sehen also, wie wichtig es ist, keine impliziten Funktionsdeklarationen zu verwenden (sondern nur explizite) und generell Warnungen immer als Fehler zu behandeln.

2 Kommandozeilenargumente in Zahlen umwandeln (1 Punkt)

basiert auf Vorlesung bis einschl. Abschnitt 6.b

L.6 Lesen Sie zunächst das Dokument PR3_06_L.pdf im Vorlesungsordner. Es enthält Informationen zu wichtigen Funktionen der C-Standardbibliothek.

Schreiben Sie ein Programm, das zwei als Kommandozeilenargumente gegebene Zahlen addiert und die Summe ausgibt. Wenn die Argumente fehlerhaft sind, sollen Fehlermeldungen ausgegeben werden.

Beispielablauf:

```
$ ./plus 5
Benutzung: ./plus <zahl> <zahl>

$ ./plus 5 7
5 + 7 = 12

$ ./plus -5 7
-5 + 7 = 2

$ ./plus x 8
Kann 'x' nicht in Zahl umwandeln: Falsches Format

$ ./plus 10x 8
Kann '10x' nicht in Zahl umwandeln: Falsches Format

$ ./plus 99999999999999999 9
Kann '99999999999999999' nicht in Zahl umwandeln: Numerical result
out of range
```

3 Probleme mit Zeiger-Parametern (0 Punkte)

basiert auf Vorlesung bis einschl. Abschnitt 6.b

a) Die folgenden Funktionen geben einen Zeiger zurück. Betrachten Sie die Funktionen und erklären Sie die Fehler/Probleme jeder Funktion.

```c
int* func1(void)
{
  int a = 42;
  return &a;
}
```

```c
int* func2(void)
{
  int* a;
  *a = 42;
  return a;
}
```

```c
int* func3(void)
{
  int *a;
  a = (int*) malloc(sizeof(int));
  *a = 42;
  return a;
}
```

b) Die folgende Funktion hat einen Zeiger als Parameter. Was ist der Fehler in dem Programm?

```c
# include<stdio.h>
# include<stdlib.h>

void func(int* p)
{
  p = (int*)malloc(sizeof(int));
}

int main()
{
  int *a;
  func(a);
  *a = 42;
  printf("%d\n", *a);
  return 0;
}
```

c) Was ist die Ausgabe des folgenden Programms? Warum?

```c
#include<stdio.h>
void f(int *p, int *q)
{
  p = q;
  *p = 2;
}
int i = 0, j = 1;
int main()
{
  f(&i, &j);
  printf("%d %d \n", i, j);
  return 0;
}
```

4 Stackframes (0 Punkte)

basiert auf Vorlesung bis einschl. Abschnitt 6.b

Betrachten Sie folgendes Programm:

```c
#include <string.h>

struct bsp {
  char c[3];
  short i;
};
```

```c
void g(short* p) {
  int i = 79;
  /* C */
  *p = i;
}

void f(struct bsp* s) {
  s->i = 99;
  /* B */
  g(&(s->i));
}

int main(void) {
  struct bsp s[2];
  strcpy(s[0].c, "Hi");
  s[0].i = 52;
  /* A */
  strcpy(s[1].c, "Ho");
  s[1].i = 42;
  f(&s[1]);
  /* D */
  return 0;
}
```

Wie sieht der Stack an den Punkten A, B, C und D aus? Verwenden Sie für die Beschreibung wie in der Vorlesung Tabellen mit Adresse, Name und Wert. Der Stackframe der main-Funktion soll an der Adresse 2000 beginnen. Beachten Sie, dass der Stack nach unten wächst, und berücksichtigen Sie ggf. Padding Bytes.

5 Speicherorte (0 Punkte)

basiert auf Vorlesung bis einschl. Abschnitt 6.d

Betrachten Sie folgendes Programm. In welchen Speicherbereichen befinden sich i, j und k?

```c
int i;
int main()
{
  int j;
  int *k = (int *) malloc(sizeof(int));
}
```

6 Zeiger auf Funktionen (1 Punkt)

basiert auf Vorlesung bis einschl. Abschnitt 6.f

Schreiben Sie zunächst drei Funktionen, alle mit einem `double`-Parameter und -Ergebnis:

- Berechnung der Mehrwertsteuer (19%) für einen Nettobetrag
  - Parameter: Nettobetrag
  - Rückgabewert: entsprechender Steuerbetrag
- Berechnung des Bruttobetrags für einen Nettobetrag
  - Parameter: Nettobetrag
  - Rückgabewert: entsprechender Bruttobetrag (= Nettobetrag plus Steuerbetrag)
- Berechnung des Nettobetrags für einen Bruttobetrag
  - Parameter: Bruttobetrag
  - Rückgabewert: entsprechender Nettobetrag (= Bruttobetrag ohne Steuer)

Das Hauptprogramm soll dem Benutzer die Möglichkeit bieten, eine der drei Funktionen auszuwählen und dann einen Wert und einen entsprechenden Betrag einzugeben. Es soll die gewählte Funktion mit dem Betrag aufrufen und das Funktionsergebnis auf den Bildschirm

ausgeben. Auswahl und Aufruf der Funktionen sollen beliebig oft wiederholt werden können, bis der Benutzer ein Ende der Programmausführung wünscht.

Beispielablauf (Benutzereingaben sind **unterstrichen**):

```
Ihre Eingabe
      <funktion> [<betrag>]
Bedeutung von <funktion>: 0=Mwst. vom Netto, 1=Brutto vom Netto, 2=Netto vom Brutto, 3=Ende
z. B. 0 99.95      (für die Berechnung der Mehrwertsteuer von 99.95 netto)
> 0 99.95
Mwst. vom Netto: 18.99


Ihre Eingabe
      <funktion> [<betrag>]
Bedeutung von <funktion>: 0=Mwst. vom Netto, 1=Brutto vom Netto, 2=Netto vom Brutto, 3=Ende
z. B. 0 99.95      (für die Berechnung der Mehrwertsteuer von 99.95 netto)
> 1 99.95
Brutto vom Netto: 118.94


Ihre Eingabe
      <funktion> [<betrag>]
Bedeutung von <funktion>: 0=Mwst. vom Netto, 1=Brutto vom Netto, 2=Netto vom Brutto, 3=Ende
z. B. 0 99.95      (für die Berechnung der Mehrwertsteuer von 99.95 netto)
> 2 118.94
Netto vom Brutto: 99.95


Ihre Eingabe
      <funktion> [<betrag>]
Bedeutung von <funktion>: 0=Mwst. vom Netto, 1=Brutto vom Netto, 2=Netto vom Brutto, 3=Ende
z. B. 0 99.95      (für die Berechnung der Mehrwertsteuer von 99.95 netto)
> 3
```

Wegen der Gleichartigkeit der oben genannten Funktionen setzen Sie bitte Funktionszeiger ein. Ein Array von Funktionszeigern deklarieren Sie wie folgt:

```
Rückgabetyp (*varname[Arraygröße])(Parameterliste);
```

Der Aufruf eines Elements des Arrays erfolgt so:

```
Ergebnis= (*varname[Index])(Argumente);
```

Verzichten Sie auf `switch` oder ein kaskadiertes `if/else` zur Auswahl der gewünschten Berechnungsfunktion, sondern verwenden Sie die Eingabe direkt als Index des Funktionszeigerarrays.

7 qsort-Bibliotheksfunktion benutzen (0 Punkte)

basiert auf Vorlesung bis einschl. Abschnitt 6.f

Die C-Standardbibliothek enthält eine Funktion `qsort`, die einen Array von beliebigem Elementtyp mit dem Quicksort-Algorithmus sortiert (deklariert in `stdlib.h`, siehe `man qsort` für die genauen Parameter-Typen: Sie müssen `qsort` mit Argumenten von exakt den dort angegebenen Typen aufrufen, d.h. meistens casten).

a) Schreiben Sie ein Programm, das einen Array von 100 zufälligen `int`-Zahlen zwischen 1 und 1000 (siehe Leseaufgabe `PR3_06_L.pdf` für die Erzeugung von Zufallszahlen in C) mit `qsort` sortiert und danach ausgibt.

b) Schreiben Sie ein Programm, das einen Array von 100 Angestellten mit zufälligem `int`-Gehalt zwischen 2000 und 6000 mit `qsort` nach ihrem Gehalt sortiert und danach ausgibt; benutzen Sie dabei folgenden Typ für Angestellte:

```
typedef struct {
  char name[15]; /* Platz für Schmidt_1234 */
  int gehalt;
} angestellter;
```

Der Einfachheit halber sollen die Angestellten alle den Namen Schmidt_*n*" mit zufälligem *n* zwischen 1 und 1000 haben. Sie können das *n* z.B. so an den Namen anhängen, wenn Sie die Angestellten initialisieren:

```
char nAsString[] = "0000";
strcpy(angs[i].name, "Schmidt_");
sprintf(nAsString, "%d", randomNumber(1000));
strcat(angs[i].name, nAsString);
```

8 Opaker Typ für Vektoren (0 Punkte)

basiert auf Vorlesung bis einschl. Abschnitt 6.f

a) Implementieren Sie einen opaken Typ `DVektor`, der einen Vektor von `double`-Werten realisiert (intern als Array variabler Länge realisiert). `DVektor` sollte folgende Funktionen bieten:
   - `DVektor create(double data[], int nElems);`
     - Erzeugt einen `DVektor` mit `nElems` Elementen aus den gegebenen Daten
   - `DVektor copy(DVektor original);`
     - Erzeugt einen neuen `DVektor` als unabhängige ("tiefe") Kopie von `original`
   - `void delete(DVektor v);`
     - Löscht `v`
   - `int add(DVektor destination, DVektor source);`
     - Addiert die Werte aus `source` zu den entsprechenden Werten von `destination`. `source` bleibt dabei unverändert.
     - Rückgabwert ist 1, außer wenn `source` und `destination` unterschiedliche Länge habe: In dem Fall ist der Rückgabewert 0, und `destination` bleibt unverändert.
   - `void process(DVektor v, void (*f)(double* elemPtr))`
     - Ruft die Funktion `f` für jedes Element von `v` auf, wobei das Element als call-by-reference-Argument an `f` übergeben wird.

   Sie können Ihren `DVektor` mit dem Programm `DVektor_test.c` in den Anlagen zu diesem Übungsblatt testen.

b) Wie könnte ein Nutzer von `DVektor` eine Funktion realisieren, die alle Elemente des Vektors aufsummiert (also etwa `double sum(DVektor v)`), ohne die Implementierung von `DVektor` zu kennen?

c) Verallgemeinern Sie `DVektor` (ohne `add()`) auf beliebige Elementtypen statt `double`, (nennen Sie den neuen Typ einfach `Vektor`), indem Sie die Elementgröße als zusätzlicher Parameter `int elemSize` an `create()` übergeben und die Elemente als Teilarray von `elemSize` Bytes innerhalb eines einzigen `char`-Arrays aus `nElems * elemSize` Bytes (1 `char` = 1 Byte) realisiert werden. `process` bekommt dann folgenden Prototyp:

```
void process(Vektor v, void (*f)(char* elemPtr, int elemSize))
```

d.h. `f` bekommt neben der Adresse eines Elements auch seine (in `v` gespeicherte) Elementgröße übergeben.

Sie können Ihren `Vektor` mit dem Programm `Vektor_test.c` in den Anlagen zu diesem Übungsblatt testen.

d) `Vektor` unterscheidet sich von `DVektor` (abgesehen von der Umstellung von `double` auf `char[]` und vom fehlenden `add()`, das für allgemeine Elementtypen nicht mehr sinnvoll ist) in einem weiteren Punkt: Die Benutzung von `process` ist weniger typsicher – warum? (Dieser Unterschied ist übrigens auch der Grund, warum man - auch wenn man `add()` nicht benötigt – trotzdem nicht auf `DVektor` verzichten sollte, indem man stattdessen `Vektor` (mit `sizeof(double)` als Elementgröße) benutzt.)