1. Einstieg in C (und die Bash)

Mit den ersten Aufgaben lernen Sie das Programmieren in C und die Benutzung der Linux-Standard-Shell `bash` kennen, dabei geht es um die Unterschiede zwischen C und C-basierten objektorientierten Sprachen (C++, C#, Java etc.).

Laden Sie von der Kurswebseite das Quellcode-Archiv `swf-sp-ws2023-ue01.tgz` herunter, öffnen Sie ein Terminalfenster, erzeugen Sie einen Ordner `sp-ws2023` , wechseln Sie hinein und entpacken Sie dort das Archiv:

```
$ mkdir sp-ws2023; cd sp-ws2023
$ tar xzf ~/Downloads/swf-sp-ws2023-ue01.tgz
```

(falls sich das Archiv in `~/Downloads` befindet). Sie finden dann einen neuen Unterordner `ue01`, der weitere Unterordner `ue01a`, `ue01b` etc. für die Teilaufgaben enthält. Sie wechseln mit

```
$ cd ue01a
```

in einen dieser Unterordner und mit

```
$ cd ..
```

wieder eine Ebene nach oben zurück.

a) Wechseln Sie in den ersten Unterordner (`ue01a`) und betrachten Sie die enthaltene C-Code-Datei `ue01a.c` im Editor oder Dateibetrachter (z. B. mit `vi ue01a.c`). Kompilieren Sie das Programm mit dem C-Compiler (`gcc -o ue01a ue01a.c`) und führen Sie es aus (`./ue01a`). Das ist das unvermeidliche Hello-World-Programm.

Das Programm sorgt mit `#include <stdio.h>` dafür, dass die Standard-I/O-Bibliothek verwendet wird, in der die Funktion `printf()` implementiert ist. Jedes C-Programm muss mindestens eine Funktion namens `main()` enthalten, die beim Programmstart aufgerufen wird.

b) Weiter geht es mit `ue01b.c` im Ordner `ue01b`. Hier lernen Sie ein paar der Standardtypen kennen, die Sie in C nutzen können. `int`, `long` und `char` sind Integertypen (mit Vorzeichen; für die nicht-negativen Varianten gibt es `unsigned int`, `unsigned long` und `unsigned char`), `float` und `double` sind Floating-Point-Typen (immer mit Vorzeichen), und Strings kann man auf zwei Weisen deklarieren. Es gibt keinen eigenständigen String-Typ in C, Strings werden immer als „Sammlung“ von Zeichen (`char`) behandelt.

Über `char s[20];` wird ein String der Länge 20 deklariert, genauer: ein `char`-Array mit 20 Einträgen. Da Strings in C immer 0-terminiert sein müssen, passen maximal 19 nutzbare Zeichen (und das abschließende `\0`, ASCII-Zeichen 0) hinein. `\n` steht für den Zeilenumbruch (newline). Mit `s[0]`, `s[1]` usw. bis `s[19]` können Sie die 20 einzelnen Elemente (vom Typ char) ansprechen, `s` (ohne Index) gibt die Speicheradresse zurück, an der das Array beginnt.

Die zweite Methode erlaubt Strings beliebiger Länge und arbeitet mit Pointern: Mit `char *s;` deklarieren Sie so einen Zeiger, `s` ist dann die Anfangsspeicheradresse des Strings. Es wird zunächst kein Speicher für den String deklariert (weil nicht klar ist, wie lang er wird); das müssen Sie selbst mit der Funktion `malloc()` (memory allocation) machen. Lesen Sie die Manpage zu `malloc()` (`man malloc`, nur Abschnitte *DESCRIPTION* und *RETURN VALUE*).

Am Anfang erzeugt das Programm mit `snprintf()` einen String, der dann mehrfach kopiert wird. Der Aufruf `snprintf (ziel, länge, format, var1, var2, …)` verwendet den Format-String format als Muster und baut die Inhalte der Variablen `var1`, `var2`, … an den passenden Stellen ein; das Ergebnis landet in `ziel`. Beispiel: `snprintf (z,10,"%d + %d ist %d\n",4,5,4+5)` schreibt in `z` den String `"4 + 5 ist 9\n"`.

`%d` (bzw. `%ld`) ist dabei Platzhalter für einen Integer-Wert, einen Floating-Point-Wert könnten Sie über `%f` einbauen, und einen anderen String mit `%s`.

Das Programm verwendet auch eine Precompiler-Direktive (`#define STRLEN 25`), die `STRLEN` auf `25` setzt. Beim Kompilieren wird vom Precompiler jedes Auftreten von `STRLEN` durch `25` ersetzt, bevor der Compiler seine Arbeit beginnt. Die meisten Bibliotheksfunktionen für String-Behandlung erwarten einen Pointer to `char` als Argument.

Kompilieren Sie das Programm und führen Sie es aus.

Schauen Sie noch einmal in den Quellcode. Strings (egal ob als `char`-Array fester Länge oder als Pointer to `char` deklariert) können Sie nicht direkt einander zuweisen (mit `s1 = s2;`), Sie müssen stattdessen die Funktion `strcpy()` oder `strncpy()` verwenden. Lesen Sie in der Manpage zu `strncpy` (`man strncpy` in der Shell) nach, wie diese beiden Funktionen sich unterscheiden. `strcpy()` ist die Quelle vieler Buffer-Overflow-Sicherheitslücken (warum?).

Das Programm hat drei `#include`-Anweisungen, weil gleich mehrere Funktionen aus Bibliotheken benötigt werden. Man kann `#include` in zwei Varianten verwenden, als

```
#include <name.h>
```

und als

```
#include "name.h" bzw. #include "pfad/pfad/.../name.h"
```

Die erste Variante sucht nach Header-Dateien (`*.h`) von bekannten Standardbibliotheken, die zweite bindet eine `.h`-Datei an, deren Ort Sie exakt angeben (z. B. für selbst geschriebene `.h`-Dateien).

c) In `ue01c.c` sehen Sie, wie Sie Strings (in ihrer Pointer-Darstellung) zeichenweise manipulieren können. Wenn Sie einen String mit `char *s;` deklariert haben, können Sie mit `*s` auf das `char`-Element zugreifen, auf das der Zeiger zeigt.

Pointer erlauben (in eingeschränktem Umfang) Addition und Subtraktion, Sie können also Ausdrücke der Form `s + 2`, `s - 1` etc. verwenden. Das Ergebnis der Berechnung *pointer* + *integer* ist wieder ein Pointer, der in einem Array auf folgende oder vorangehende Einträge zeigt. Dieses Feature nutzt die Funktion `copy_string()` im Programm. Die Operatoren `++` und `--` sind in C Abkürzungen für das Inkrementieren und das Dekrementieren. (Der Befehl `i++;` entspricht der Zuweisung `i=i+1;`.)

Wollen Sie einen String beim Deklarieren auch gleich mit einem Wert vorbelegen, müssen Sie die Syntax `char s[] = "Inhalt";` verwenden. Die leeren Klammern deuten darauf hin, dass es sich um ein `char`-Array von (zunächst) unbestimmter Länge handelt, durch die folgende Initialisierung wird dann aber klar, wie lang der String ist.

(i) Warum sehen Sie in der Programmausgabe sowohl bei `s1` als auch `s2` vorne „ABC“, obwohl das Programm nur in `s1` die ersten drei Zeichen verändert?

(ii) Warum müssen Sie im `malloc()`-Aufruf jeweils `(len+1)` Zeichen und nicht nur `len` Zeichen für `s3` und `s4` reservieren?

(iii) Was macht die Funktion `sub_string()`, die einen ähnlichen Aufbau wie die Funktion `copy_string()` hat, und wie funktioniert das?

(iv) Ersetzen Sie die vorhandenen Deklarationen von `s3` und `s4` durch `char s3[10]; char s4[10];` (also Strings mit fester Länge 10) und entfernen Sie die `malloc()`-Aufrufe für `s3`, `s4`. Was geht schief, wenn Sie das so veränderte Programm ausführen? Haben Sie eine Idee, woran das liegt?

(v) Schreiben Sie eine Funktion `odd_copy_string()`, die wie `copy_string()` arbeitet, aber beim Kopieren jedes zweite Zeichen des Quell-Strings überspringt. (Aus ABCDEFG wird ACEG.) Testen Sie Ihre Funktion mit Strings, die gerade oder ungerade Länge haben.