Lehrstuhl für Connected Mobility
Fakultät für Informatik
Technische Universität München

Grundlagen - Betriebssysteme und Systemsoftware

IN0009, WiSe 2019/20

Hausaufgabe 1

30.10.2019 23:59 Uhr über Artemis(`https://artemis.ase.in.tum.de`)

Aufgabe 1 Einführung C

Keine Abgabe für diese Aufgabe!

a) Sanitizer
Da wir Sie dazu zwingen möchten „guten“ C-Code zu schreiben, wird ihr Programm mehrere Male kompiliert und ausgeführt. Dies geschieht jedes mal mit anderen *Compiler Flags* aktiviert.
Im folgenden werden die verschiedenen Kompiliertests erläutert:

Standard

```
$ make
$ make run
```

Hier wird Ihr Programm im C11 Standard kompiliert. Zudem werden zusätzliche Warnungen beim kompilieren angezeigt, falls vorhanden.

Adress Sanitizer

```
$ make asan
$ ./asan.out
```

Hier werden beim kompilieren Adress Sanitizer aktiviert. Diese weisen Sie beispielsweise daraufhin, wenn sie auf Speicher zugreifen, den Sie nicht zuvor reserviert haben.
Beispiel:

```
1  #include <stdlib.h>
2
3  int main() {
4      char* buf = malloc(100 * sizeof(char));
5      buf[200] = 'b';
6      free(buf);
7      return 0;
8  }
```

```
1  $ make asan
2  $ ./asan.out
3  =================================================================
4  ==11426==ERROR: AddressSanitizer:
5  heap-buffer-overflow on address 0x60b0000001b8
6  at pc 0x0000004011bb bp 0x7ffcba7d7c20 sp 0x7ffcba7d7c10
7  WRITE of size 1 at 0x60b0000001b8 thread T0
8      #0 0x4011ba in main /home/user/fak.c:5
9      #1 0x7fe6599a4f42 in __libc_start_main (/lib64/libc.so.6+0x23f42)
10     #2 0x4010ad in _start (/home/user/asan.out+0x4010ad)
11
12 Address 0x60b0000001b8 is a wild pointer.
13 SUMMARY: AddressSanitizer: heap-buffer-overflow /home/user/fak.c:5 in main
14 [...]
15 ==11426==ABORTING
```

Das Interessante hier ist Zeile 8 in der Fehlermeldung beim Ausführen. `#0 0x4011ba in main /home/user/fak.c:5` sagt uns, das wir in Zeile 5 unseres Programmes den Fehler verursacht haben.

Undefined Behavior Sanitizer

```
$ make ubsan
$ ./ubsan.out
```

Prof. Dr.-Ing. Jörg Ott ott@in.tum.de
Martin Uhl gbs@cm.in.tum.de

Analog zum Adress Sanitizer funktioniert der Undefined Behavior Sanitizer. Hierbei wird nach Code gesucht, welcher Funktionen beinhaltet, die nicht im C-Standard definiert sind.
Beispiel:

```c
#include <stdio.h>

int main() {
    int i;
    if(i % 2) {
        printf("p ist ungerade\n");
    }
    else {
        printf("p ist gerade\n");
    }
    return 0;
}
```

```
$ make ubsan
fak.c: In function 'main':
fak.c:5:4: error: 'i' is used uninitialized
    in this function [-Werror=uninitialized]
    5 |  if(i % 2) {
      |    ^
cc1: all warnings being treated as errors
make: *** [Makefile:58: ubsan] Error 1
```

Da `i` nie einen Wert zugewiesen wurde, kann `i` alles sein. In C werden Variablen **nicht** für Sie automatisch bei der Deklaration initialisiert.

Leak Sanitizer

```
$ make lsan
$ ./lsan.out
```

Ebenfalls analog zum Adress Sanitizer funktioniert der Leak Sanitizer. Hierbei wird nach Code gesucht, welcher Speicher „leakt“. Dabei gehen Pointer auf Speicher verloren. Dieser Speicher kann somit mittels `free(...)` nicht mehr freigegeben werden.
Beispiel:

```c
#include <stdlib.h>

void *p;
int main() {
  p = malloc(7);
  p = NULL;
  return 0;
}
```

```
$ make lsan
$ ./lsan.out
=================================================================
==14005==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 7 byte(s) in 1 object(s) allocated from:
    #0 0x7fb087f9e362 in __interceptor_malloc (/lib64/liblsan.so.0+0xf362)
    #1 0x401133 in main /home/user/fak.c:5
    #2 0x7fb087decf42 in __libc_start_main (/lib64/libc.so.6+0x23f42)

SUMMARY: LeakSanitizer: 7 byte(s) leaked in 1 allocation(s).
```

Da wir `p` in Zeile 6 überschreiben, verlieren wir alle unsere Pointer auf die in Zeile 5 reservierten 7 Byte. Wir können sie somit nicht wieder freigeben.

b) Mehr Quellcode Dateien
Es ist möglich zusätzliche Quellcode Dateien hinzuzufügen. Hierfür müssen Sie nur die **Makefile** anpassen. Sollen die 3 Dateien: *venit.c*, *videns.c* und *vicit.c* hinzugefüget werden, so muss das **Makefile** wiefolgt angepasst werden:

```
1  .PHONY: all run
2
3  all: helloWorld
4
5  ############################################################
6  # DON'T change the variable names of INCLUDEDIRS and SOURCE
7  ############################################################
8  # A list of include directories
9  INCLUDEDIRS =
10 # A list of source files
11 SOURCE = helloWorld.c venit.c videns.c vicit.c
12 [...]
```

Wichtig: Nur die Zeilen 9 und 11 werden vom Tester später beachtet. Alle anderen Änderungen in Ihrer **Makefile** werden ignoriert!

Aufgabe 2 Hello World!

Traditionsgemäß wollen wir die Einführung der Programmiersprache C mit einem ”Hello World”-Programm beginnen. Schreiben Sie dazu ein C-Programm, das den Text ”`Hello␣World!`” (ohne Anführungszeichen) gefolgt von einem Zeilenumbruch auf stdout (auf dem Terminal) ausgibt.

Abzugebende Dateien:

- helloWorld.c
- Makefile

Hinweis:

- Um die Anzahl von Leerzeichen sichtbar zu machen werden diese mit dem folgendem Symbol gekennzeichnet: ␣
- Geben Sie den Text exakt wie angegeben aus. Achten Sie insbesondere auch auf nicht sichtbare Steuerzeichen, wie Leerzeichen und Zeilenumbrüche.

Hintergrundwissen:

- Das Anfügen eines Zeilenumbruches hat mehrere Gründe. Wenn Sie eine Textausgabe in Ihrem Programm vornehmen, so wird diese nicht unmittelbar ausgegeben, sondern in einem Puffer zwischengespeichert. Dieser Puffer wird bei verschiedenen Ereignissen „geflusht“. Das bedeutet, der Inhalt des Puffers wird ausgegeben und anschließend aus dem Puffer gelöscht. Dies ist u.a. der Fall, wenn der auszugebende Text einen Zeilenumbruch enthält. Wenn Sie Text ausgeben wollen ohne in die nächste Zeile zu springen, können Sie den Puffer manuell „flushen“. Dies geschieht mit der Funktion `fflush` aus `stdio.h`. Bei Programmende wird der Inhalt des Puffers automatisch geflusht. Abgesehen von der direkten Ausgabe, hat der Zeilenumbruch auch den Vorteil, dass Ihr Programm mit einer leeren Zeile endet. Andernfalls würden weitere Ausgaben nach Programmende direkt an Ihre Ausgabe angehängt. Dies erzeugt einen unschönen Versatz in der Textausgabe des nachfolgenden Programms.

Aufgabe 3 Elementare Ein-/Ausgabe: ASCII-Tabelle *

Der American Standard Code for Information Interchange (ASCII) ist eine weitverbreitete Zeichencodierung. Standardmäßig wird hierbei jeder Zahl von 0 bis 127 ein *character* zugeordnet. Die Großbuchstaben von A bis Z sind bspw. durch die Zahlen 65 bis 90 im Dezimalsystem codiert. Um einen Überblick über die enthaltenen Zeichen zu bekommen, erstellen Sie ein Programm, dass folgende Tabelle für alle ASCII-Codes in verschiedenen Zahlensystemen ausgibt:

```
Oct     Dec     Hex     Char
000     0       00
...
012     10      0a

013     11      0b

...
033     27      1b
034     28      1c
...
060     48      30      0
061     49      31      1
...
101     65      41      A
102     66      42      B
...
```

Nutzen Sie für die Formatierung der Zeilen `printf` mit geeigneten Formatstring-Platzhaltern (`man 3 printf`). Achten Sie auf die verschieden formatierten Zahlen: **oktal** dreistellig mit führender 0, **dezimal** linksbündig,

hexadezimal zweistellig mit führender 0 (a-f in Kleinbuchstaben) dargestellt. Die letzte Spalte soll den jeweiligen Character ausgeben, unabhängig davon, ob dieser darstellbar ist oder nicht (s. unten), eine spezielle Behandlung solcher Fälle ist nicht gefragt. Vn nicht-darstellbaren Zeichen als `char` wird hierbei (erwünschte) Nebeneffekte haben, wie im obigen Beispiel angedeutet. Die jeweiligen Spalten sollen durch Tabulatoren getrennt werden.

Beantworten Sie sich die Frage, warum die Ausgabe die ersten 32 Zeichen des ASCII-Codes nicht darstellbar sind. Erklären Sie Teile Ihrer Antwort mithilfe des Terminalbefehls `man ascii` (keine Abgabe).

Aufgabe 4 Binary Search Tree

In der Vorlesung *Grundlagen Algorithmen und Datenstrukturen* haben Sie die Baum- (Tree-) Datenstruktur kennengelernt. Bäume sind nichts Anderes als zusammenhängende, kreisfreie Graphen mit Knoten und Kanten. Sie werden im Betriebssystem-Kernel zur Verwaltung von unterschiedlichen Informationen eingesetzt. Insbesondere ermöglichen Bäume einen bequemen und schnellen Zugriff auf die gespeicherten Daten.

In dieser Aufgabe implementieren Sie einen einfachen Baum, nämlich den *Binary Search Tree* (BST).[1] Im Folgenden finden Sie einen Beispiel-Knoten eines BST. Dieser enthält einen Pointer (Kante) zum jeweiligen linken und rechten Kind, sowie ein Feld für zusätzliche Informationen in Form einer Integer-Variable:

```
struct node {
        struct node *left;
        struct node *right;
        int val;
};
```

Die folgenden Funktions-Prototypen sind dafür zuständig den Baum zu verwalten:

```
/* (Dynamically) allocate a new node */
struct node *new_node(int val);

/* (Recursively) insert a new value into the tree */
struct node *insert_node(struct node *root, struct node *node);

/* Remove a node with value val from the tree */
struct node *remove_node(struct node *root, int val);
```

Dabei allokiert die Funktion `new\_node` dynamisch Speicher für einen Knoten vom Typ `struct node`, initialisiert diesen mit dem übergebenen Wert `val`, und gibt einen Pointer zum Knoten zurück an die aufrufende Funktion. Bei der Knoten-Initialisierung werden auch weitere Felder des Knotens initialisiert, um zufällige Werte in den betroffenen Feldern zu vermeiden.

Die Funktion `insert_node` fügt eine bereits existierenden Knoten `node` in den Baum definiert durch `root` ein. Dafür muss diese Funktion den Tree (*iterativ* oder *rekursiv*) durchlaufen und dabei entscheiden, wohin im Baum der neue Wert in Form eines Knotens eingefügt werden soll. Bedenken Sie, dass es sich bei der Datenstruktur um einen BST handeln soll. Für einen BST sind verschiedene Invarianten möglich. Um Ihren Baum testen zu können muss für Ihren Baum gelten, dass der **linke Kindknoten einen kleineren oder gleichen Wert enthält** als der Elternknoten. Der rechte Kindknoten darf dementsprechend nur einen größeren Wert enthalten. **Jeder neu eingefügte Knoten muss ein Blatt des Baumes sein.**

Die Funktion `remove_node` entfernt den Knoten mit dem Wert `val` aus dem Baum. Gibt es mehrere Knoten mit dem Wert `val`, so ist der Knoten mit der kleinsten Pfadlänge zur Wurzel zu löschen. Beachten Sie, dass es mehrere Fälle beim Entfernen eines Knoten gibt. Für den Fall, dass der zur entfernende Knoten zwei Kindknoten hat, so muss er durch den Knoten mit dem nächstgrößeren Wert ersetzt werden. Nach der Operation muss es sich stets um einen validen BST handeln.

a) Implementierung

Implementieren Sie einen Binary Search Tree, indem Sie die obigen Funktions-Prototypen in einer Datei `bst.c` implementieren. Die Funktionssignaturen finden Sie in `bst.h`. Sie dürfen weitere Dateien erstellen, diese müssen Sie allerdings in das Makefile einfügen. Bitte beachten Sie, dass der Tester die Dateinamen aus dem Makefile ausliest und in ein internes Testmakefile einfügt. Somit ist es unabdingbar, dass sie die Kommentare im Makefile beachten.

[1] Binary Search Tree: `https://en.wikipedia.org/wiki/Binary_search_tree`

b) Higher-Order-Functions on Trees

In dieser Aufgabe werden Sie sogenannte *Higher-Order-Functions* kennen lernen. Higher-Order-Functions sind Funktionen, die eine weitere Funktion als Argument erhalten. Diese können sehr hilfreich sein, insbesondere wenn man eine bestimmte Operation auf allen Knoten des Baumes ausführen möchte (z.B. Ausgabe der Knotenwerte). Dabei erhält die Higher-Order-Function die Wurzel der BST-Datenstruktur und eine Funktion `f` als Parameter. Die Higher-Order-Function durchläuft den Baum und wendet die übergebene Funktion `f` auf jedem Baumknoten an. Was Higher-Order-Functions bequem macht, ist, dass sie nur einmal implementiert werden müssen, dafür aber unterschiedliche Operationen auf den Baum anwenden können.

1. Erweitern Sie die BST-Datenstruktur der letzten Aufgabe um Higher-Order-Functions mit den folgenden Prototypen:

```
/* Recursively iterate the tree in in-order */
void iterate_in_order(struct node *node,
        void (*f)(struct node *n);

/* Recursively iterate the tree in pre-order */
void iterate_pre_order(struct node *node,
        void (*f)(struct node *n));

/* Recursively iterate the tree in post-order */
void iterate_post_order(struct node *node,
        void (*f)(struct node *n));
```

2. Überlegen Sie sich, in welcher Reihenfolge die Funktion `free_node` auf die Knoten des Baumes angewandt werden muss, sodass alle Knoten ohne Probleme freigegeben werden können (keine Abgabe).

Prof. Dr.-Ing. Jörg Ott
ott@in.tum.de
Martin Uhl
gbs@cm.in.tum.de