Rechnernetze
SoSe 2012

Nils Aschenbruck;
Jan Bauer, Alexander Bothe, Matthias Schwamborn

Übungsblatt Nr.1

Veröffentlichung: 23.04.2012
Besprechung: 07.05.2012

Bitte denken Sie daran, sich auf der Mailingliste einzutragen. Über diese Liste werden wichtige Ankündigungen gemacht! **Anmeldung auf:**
https://list.serv.uni-osnabrueck.de/mailman/listinfo/vl-rne.

Bitte helfen Sie Ihren Kommilitonen und nutzen Sie die Mailingliste bei aufkommenden Fragen!

Aufgabe 1: Dynamische Listen

Damit Sie sich mit den in der Vorlesung eingeführten Konzepten von C vertraut machen können, soll in dieser Aufgabe eine verkettete Liste implementiert werden, in der Personendaten gespeichert werden können. Die Daten sollen in alphabetischer Reihenfolge eingefügt werden. Die Sortierung erfolgt nach dem Nachnamen der Personen. Im Falle von gleichen Nachnamen wird der Vorname als zweites Kriterium hinzugezogen. Stimmen sowohl Nach- als auch Vornamen überein, ist die Reihenfolge der Speicherung irrelevant. Bei den Daten handelt es sich um Vor- und Nachnamen, die getrennt durch ein Leerzeichen in einer Textdatei vorliegen. Eine Zeile besteht aus einem Vornamen und einem Nachnamen. Sowohl Vor- als auch Nachnamen sind jeweils maximal 100 Zeichen lang. Eine solche Namensdatei könnte beispielsweise wie folgt aussehen:

Victor Hugo
Albert Einstein
Johann-Wolfgang Von-Goethe

Der Name der Datei soll während der Laufzeit abgefragt werden. Das Programm liest alle Daten aus der Datei ein und gibt diese anschließend in sortierter Form auf dem Bildschirm aus. Der vom Programm reservierte Speicher soll vor Beendigung wieder freigegeben werden.

Um nicht unnötig viel Speicherplatz für die einzelnen Einträge zu belegen, soll für die jeweiligen Einträge nur jeweils der wirklich benötigte Speicherplatz reserviert werden. Schauen Sie sich in diesem Zusammenhang die beiden Funktionen `malloc` und `calloc` an.

Ein Listenelement könnte beispielsweise folgende Struktur haben:

```
struct node {
  struct node *succ;
  char *vorname;
  char *nachname;
};
```

Bei der Implementierung sollen mindestens zwei Header-Dateien verwendet werden. Eine Header-Datei, z.B. `personen_liste.h`, definiert die benötigten Funktionsköpfe für die verkettete Liste und die andere, z.B. `mystring.h` definiert Funktionen für die Bearbeitung von Zeichenketten. Auf die Standard `string.h` soll hier bewusst verzichtet werden.

Überprüfen Sie mittels des Tools „Valgrind“ (http://valgrind.org/), ob der gesamte von Ihnen reservierte Speicher wieder freigegeben wurde („`valgrind --tool=memcheck --leak-check=yes ./myprog`“).

Aufgabe 2: Makefiles

Das Programm „make“ wird benutzt, um das Kompilieren eines Projekts zu vereinfachen. So werden beim Aufruf von „make“ nur diejenigen Dateien (z.B. .c Dateien) kompiliert, deren Änderungszeitpunkt jünger ist als der des Produkts (z.B. .o Dateien). Dabei arbeitet „make“ nach Regeln, die in einer Datei namens „Makefile“ definiert sind. Diese Regeln sind nach dem folgenden Schema aufgebaut:

```
Ziel: Abhängigkeiten
 <TAB>  Kommando
 <TAB>  ...
```

Beispiel:

```
ftpd: main.o commands.o
        gcc -o ftpd main.o commands.o

main.o:
        gcc -c main.c

commands.o:
        gcc -c commands.c
```

Das Programm „make“ liest das Makefile ein und startet bei der ersten Regel, die auch „default goal“ genannt wird.

Oft ist es sinnvoll, Variablen in Makefiles zu deklarieren, die später in den Regeln benutzt werden können. Dabei werden einfache Textersetzungsregeln benutzt, ähnlich dem C-Präprozessor.

Beispiel:

```
VORLESUNG = Rechnernetze
SEMESTER = Sommersemester
JAHR = 2012

default:
        echo $(VORLESUNG) $(SEMESTER) $(JAHR)
```

Trotz der Variablen kann die Anzahl der Regeln je nach Anzahl der zu kompilierenden Dateien sehr lang werden. GNU make unterstützt eine Reihe von automatischen Variablen, die zur Laufzeit gesetzt werden, z.B.:

`$@` : Dateiname des Ziels der Regel

`$<` : Name der ersten Abhängigkeit

Beispiel (vorher müssen die Dateien a.0, b.0 und c.0 angelegt werden, z.B. mit „`touch a.0`“...):

```
OK: a.1 b.1 c.1
	echo $@

%.1: %.0
	mv $< $@
```

Eine kurze Einführung ist auf http://www.student.cs.uwaterloo.ca/~isg/res/unix/make/tutorial/ zu finden.

Eine ausführliche Anleitung zu GNU make gibt es z.B. auf http://www.gnu.org/software/make/manual/make.html.

Aufgabe: Schreiben Sie ein Makefile für Aufgabe 1, das

a) folgende Variablen definiert und benutzt:
   - `OBJECTS` (alle .o Dateien),
   - `CPPFLAGS`, das jedem `gcc` Aufruf übergeben und auf `-g -Wall` gesetzt werden soll

b) alle .c Dateien in .o Dateien unter Verwendung von automatischen Variablen kompiliert (`gcc -c`).

Achten Sie darauf, dass alle .c Dateien, die eine .h Datei einbinden, neu kompiliert werden, falls die .h Datei geändert wird.

Aufgabe 3 (Kommunikationsanalyse mit Wireshark)

Die Analyse von Datenpaketen, die über ein Kommunikationsnetzwerk verschickt werden, ist eine essenzielle Fähigkeit für die Entwicklung von neuen und Analyse von bestehenden Netzwerkprotokollen und -anwendungen. Bei unverschlüsseltem Datenverkehr können mit einem Netzwerk-Sniffer-Tool wie Wireshark umfangreiche Studien durchgeführt werden.
Wireshark, erhältlich unter **www.wireshark.org**, ist ein mächtiges Tool und kann Datenverkehr mitschneiden sowie auf verschiedenste Arten darstellen. In dieser Aufgabe sollen Sie sich mit diesem Tool vertraut machen.

a) Laden Sie sich Wireshark von der angegebenen Quelle herunter und machen Sie sich damit vertraut.
b) Auf der Vorlesungs-Website finden Sie eine Datei, die mit Wireshark mitgeschnittenen Datenverkehr beinhaltet. Erklären Sie, was Sie in der Datei sehen können. Beantworten Sie dafür die folgenden Fragen:
    - Was genau passiert in den 45 Schritten in der Logdatei?
    - Welche Art von Verbindung wurde hergestellt?
    - Welche Protokolle von welcher OSI-Ebene können identifiziert werden?
c) Welche Arten von Geräten (Hersteller) haben auf Ebene 2 miteinander gesprochen? Wie kann Wireshark diese identifizieren?
d) Ist es möglich herauszufinden, welches Betriebssystem auf der Maschine mit IP 131.220.6.214 läuft?

Viel Erfolg und viel Spaß!