

Übungspaket 5

Objektmodellierung

Übungsziele:

1. Verstehen von Funktionszeigern.
2. Anwenden von Funktionszeigern.

Literatur:

C-Skript¹, Kapitel: 85

Semester:

Wintersemester 2020/21

Betreuer:

Theo und Ralf

Synopsis:

Zeiger haben wir in den Grundlagen oft genug geübt. Insbesondere haben wir mit Zeigern Variablenparameter in Funktionen sowie dynamische Datenstrukturen realisiert. Das war ein hartes Stück Arbeit. In der Programmiersprache C gibt es aber auch Zeiger auf Funktionen. Dieses Konzept ist anfangs wieder einmal recht verwirrend aber am Ende echt cool. Funktionszeiger erhalten als Werte die Namen konkreter Funktionen, was den Anfangsadressen dieser Funktionen im Arbeitsspeicher entspricht. Konsequenterweise entspricht dann das Dereferenzieren eines Funktionszeigers dem Aufruf der referenzierten Funktion, sofern noch die Klammern und Parameter angefügt werden. Dieser Mechanismus ist sehr flexibel und eröffnet völlig neue Möglichkeiten. Auf technischer Ebene können Funktionszeiger als ein wesentlicher Schritt in Richtung objektorientierte Programmierung angesehen werden.

¹www.amd.e-technik.uni-rostock.de/ma/rs/lv/hoqt/script.pdf

Teil I: Stoffwiederholung

Teil II: Quiz

Teil III: Fehlersuche

Teil IV: Anwendungen

In diesem Anwendungsteil versuchen wir eine oft auftretende Aufgabenstellung mittels eines objektorientierten Ansatzes zu lösen. Dazu werden wir eine erste Klasse entwickeln und diese in Standard C (also *nicht* C++) realisieren. Bei der Realisierung werden wir in zwei Schritten vorgehen. Zunächst entwickeln wir ein Modul. Anschließend „transformieren“ wir es in unser C-basiertes Objektmodell, das auf Funktionszeigern basiert.

Aufgabe 1: Eingabepuffer beliebiger Länge

1. Aufgabenstellung

Nehmen wir einmal an, wir entwickeln ein größeres Programm, das mit dem Nutzer interaktiv interagiert. Als Beispiel könnte es sich um ein Studentenverwaltungssystem handeln, das verschiedene Kommandos entgegennimmt. So ein Kommando könnte wie folgt aussehen: `set student "Peter Musterfrau" iq 130`. Für diese Aufgabe verwendet man üblicherweise eine Eingabefunktion der folgenden Art:

```
1 int readln( FILE *fp, char *buf, int size )
2     {
3         int c;
4         char *end = buf + size - 1;
5         while( buf < end && (c=getc(fp)) != '\n' && c!=EOF )
6             *buf++ = c;
7         *buf = '\0';
8         return c != EOF;
9     }
```

Ein typischer Anwendungsfall könnte wie folgt aussehen:

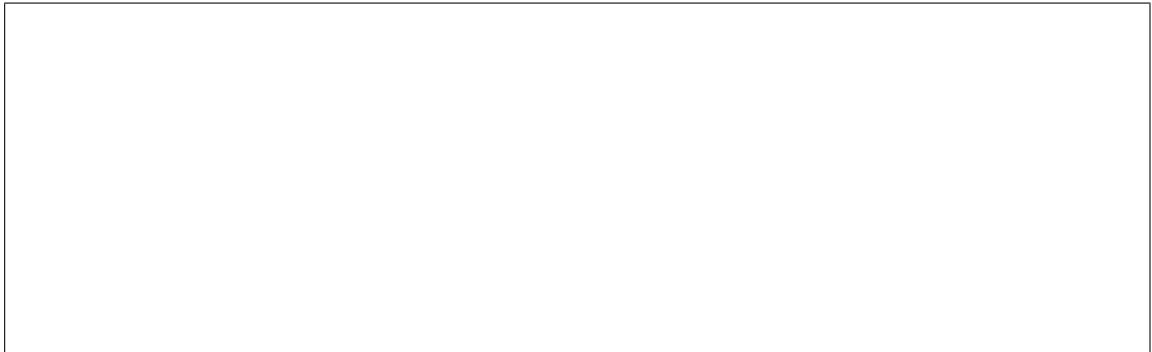
```
1     char buf[ SIZE ];
2     while( readln( stdin, buf, sizeof( buf ) ) )
3         do_something( buf );
```

Die Frage ist nun, welchen Wert man der Konstanten `SIZE` geben soll, die die Länge des eigentlichen Eingabepuffers definiert. Typische Werte hierfür sind heutzutage 128 und 256. Unabhängig von der gewählten Puffergröße kann es aber immer zu einem „Überlauf“ kommen. In diesem Fall müssen wir uns entscheiden, was wir machen wollen. Obige Beispielimplementierung „schneidet“ den den Rest der Zeile ab, um ihn einfach als nächste Zeile zu verwenden, wofür wir eine kleine Handsimulation empfehlen. Je nach Einzelfall kann dieses Vorgehen ok oder problematisch sein. Das allgemeine Problem ist, dass sich die Frage nach der *richtigen* Puffergröße nicht allgemein beantworten läßt. Sie ist entweder doch zu klein oder so groß, dass zu viel reservierter Arbeitsspeicher ungenutzt bleibt.

Die Aufgabe besteht nun darin, einen abstrakten Datentyp **CBUF** zu entwickeln, bei dem die aktuelle Puffergröße dynamisch an die vorhandene Eingabesituation angepasst wird. Wir wollen also eine Klasse **CBUF** entwickeln, die einen Puffer verwaltet, in den wir „beliebig“ viele Zeichen eintragen können, da dieser je nach Anforderung wachsen kann. Im Kern brauchen wir eine Methode **add_ch(buf, c)** die wir „beliebig“ oft aufrufen können. Es sollte klar sein, dass wir den eigentlichen Puffer **buf** mittels **malloc()** und **free()** dynamisch verwalten.

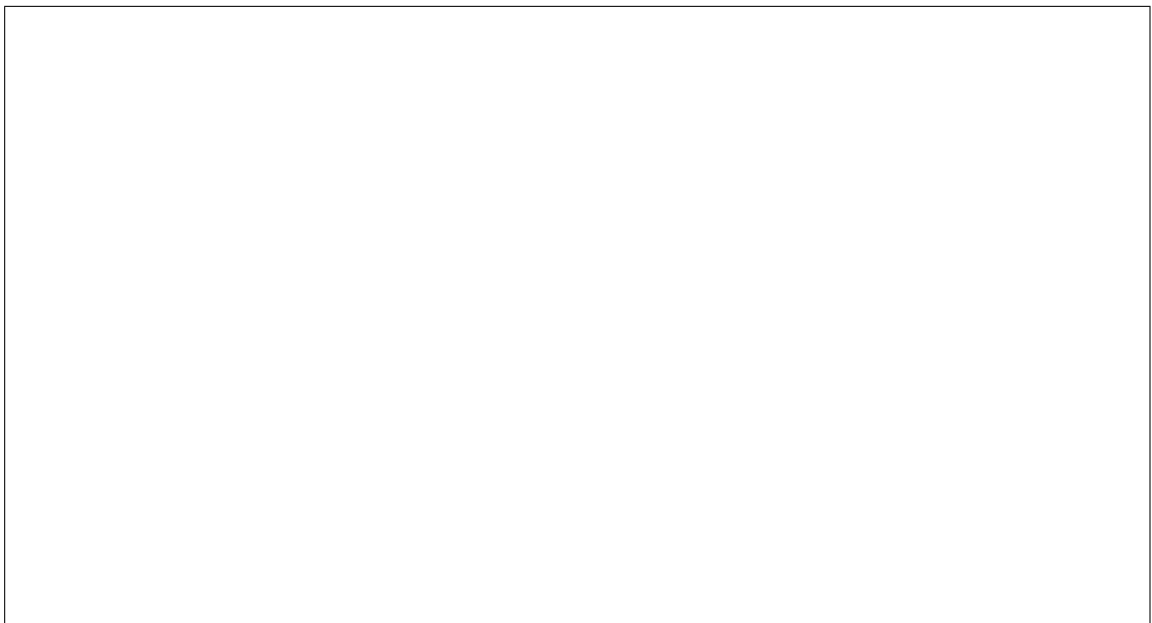
2. Entwurf

Im ersten Entwurfsschritt müssen wir definieren, welche Methoden unsere Klasse (unser Datentyp) **CBUF** haben soll. Diese Methoden definieren das Funktionsspektrum.

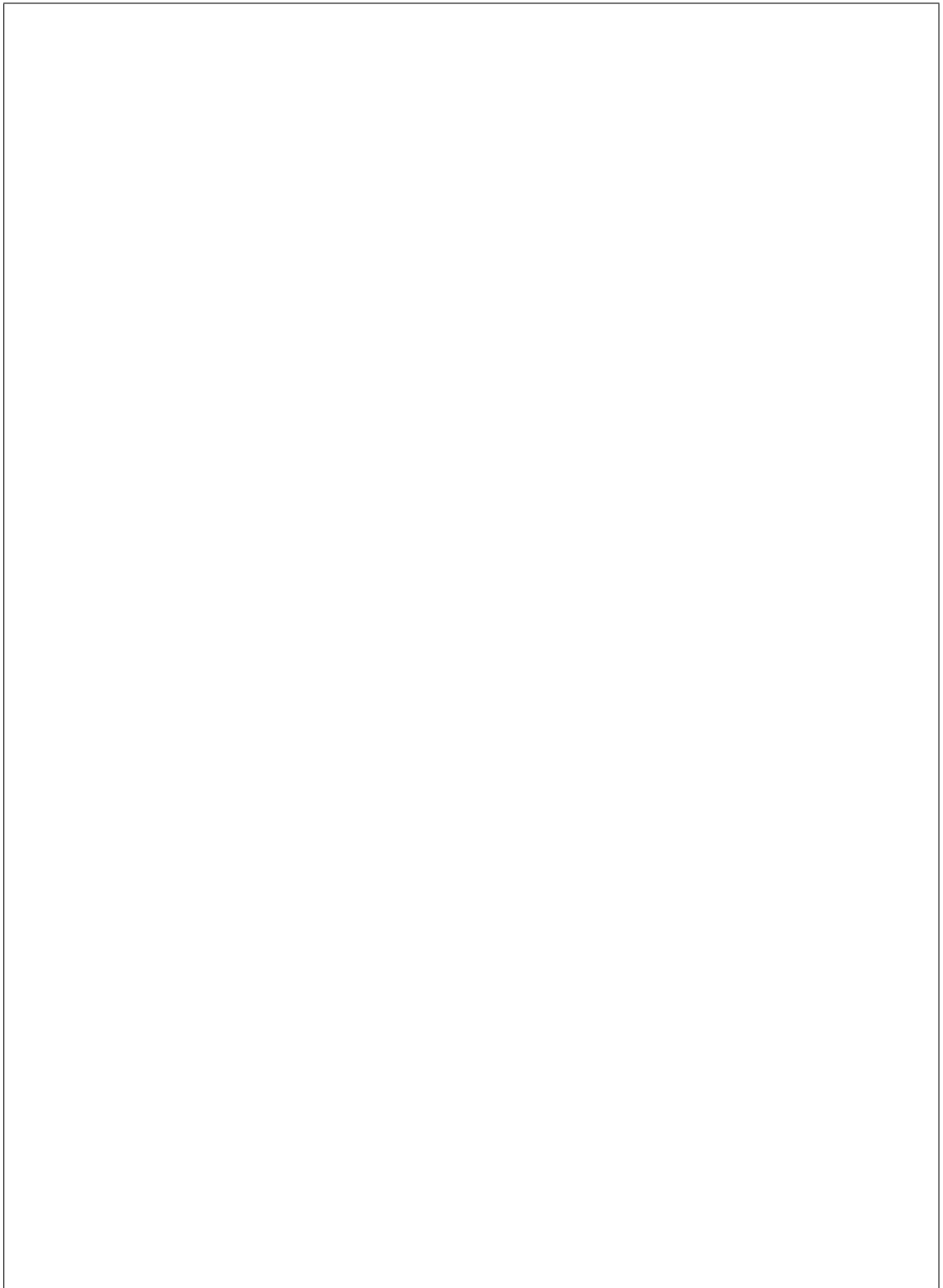


3. Kodierung

Der erste Schritt besteht in der Definition des unseres Datentyps **CBUF**:



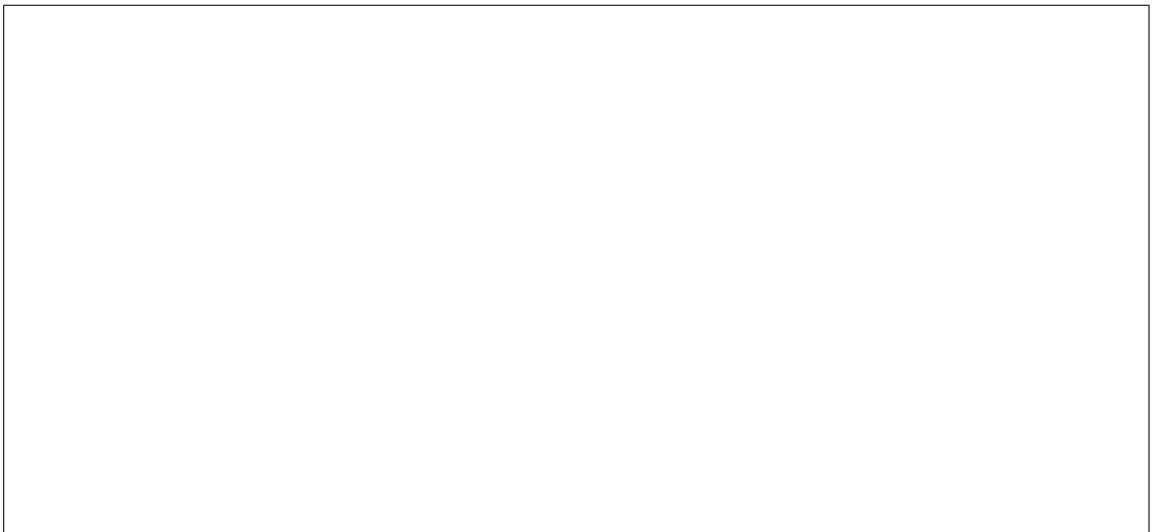
Implementierung in `cbuf.c`:



Implementierung in `cbuf.c` (Fortsetzung):

A large, empty rectangular box with a thin black border, intended for the implementation details of the `cbuf.c` file.

Ein Testprogramm `main.c`:

A large, empty rectangular box with a thin black border, intended for the details of the test program `main.c`.