Übungspaket 4

Objektmodellierung in C

Übungsziele:

1. Modelliern eines Objektes als abstrakter Datentyp
2. Implementieren einer Klasse in C
3. Anwenden von Funktionszeigern

Literatur:

C++-Skript[1], Kapitel 15 bis 18

Semester:

Wintersemester 2021/22

Betreuer:

Theo und Ralf

Synopsis:

In diesem Übungspaket werden wir noch einmal rekapitulieren, was ein Objekt ist und wie man dies in der Programmiersprache C realisiert. Im Wesentlichen besteht ein Objekt aus verschiedenen Attributen und Methoden. Methoden sind eine spezielle Formen von gewöhnlichen Funktionen, die sich in er Programmiersprache C besonders gut mittels Funktionszeigern (Skript, Kapitel 6 und Übungspaket 2). realisieren lasssen. Als konkreten Anwendungsfall nehmen wir einen „Eingabepuffer“, dessen Länge auf Anforderung wächst und somit nicht überlaufen kann.

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

Teil I: Stoffwiederholung

Aufgabe 1: Bestandteile eines Objektes

Was ist eine Klasse?	Ein abstrakter Datentyp
Wie definiert man eine Klasse?	Vorzugsweise mittels `typedef`
Was ist ein Objekt?	Eine Variable eines (Klassen-) Typs
Aus welchen Elementen bestehen Objekte?	Attribute und Methoden
Wie implementiert man Objekte?	Mittels `structs`
Zu wem gehört ein Methodenaufruf?	Zu einem Objekt
Wie nennt man dieses Objekt in `Smalltalk`?	Empfänger der Nachricht
Was ist das Charakteristikum einer Methode?	Der erste Parameter ist ein Zeiger
Worauf zeigt der erste Parameter einer Methode?	Zum (Empfangs-) Objekt
Wie realisiert man eine dynamische Bindung?	Mittels Funktionszeigern
Ist es sinnvoll, einen Konstruktor dynamisch an ein Objekt zu binden?	Nein
Warum ist das so?	Kein Objekt kein Konstruktor, kein Konstruktor kein Objekt

Aufgabe 2: Statische vs. dynamische Bindung

Erkläre mit eigenen Worten, was man unter *statischer Bindung* versteht. Was sind die Vor- und Nachteil dieses Konzeptes?

Bei der statischen Bindung ruft man eine Methode oder Funktion direkt über ihren Namen auf. Dazu muss der Methoden- bzw. Funktionsname bekannt bekannt sein. Die statische Bindung läßt sich einfach realisieren hat aber den Nachteil, dass man beim *Verwechseln* des Namens die falsche Methode aufruft.

Erkläre mit eigenen Worten, was man unter *dynamischer Bindung* versteht. Was sind die Vor- und Nachteil dieses Konzeptes?

Bei der dynamischen Bindung ruft man eine Methode oder Funktion *nicht* direkt über ihren Namen sondern einen Funktkionszeiger auf. In diesem Fall kann eine Methode bzw. Funktion als `static` spezifiziert sein, da ihr Name nicht außerhalb einer Datei bekannt sein muss. Die dynamische Bindung erfordert etwas mehr Programmierarbeit, hat aber den Vorteil, dass unterschiedlichen Objekten einer Klasse unterschiedliche Methoden zugeordnet werden können.

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 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 ohne weiteres 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.

```
1 CBUF *cb_init  ( CBUF *p );           // constructor
2 void  cb_close ( CBUF *p );           // destructor
3 CBUF *cb_malloc();                    // new CBUF
4 void  cb_free  ( CBUF *p );           // free
5 CBUF *cb_reset ( CBUF *p );           // clear the content
6
7 char  *cb_addc  ( CBUF *p, char c ); // add one char
8 char  *cb_addstr( CBUF *p, char *s );// add a string
9 char  *cb_buf   ( CBUF *p );          // get the content
```

3. Kodierung

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

Ohne Funktionszeiger sieht unsere Typdefinition `CBUF` in `cbuf.h` wie folgt aus:

```
1 typedef struct {
2           int len;                    // length incl. '\0'
3           int size;                   // total size of buf
4           char *buf;                  // the acutal buffer
5         } CBUF, *CB_PTR;
```

Unsere Idee ist wie folgt: Wir verwalten einen Puffer `buf` vom Type `char *`, in den wir die Zeichen einfüllen. Seine aktuelle Belegung merken wir uns im Attribut `len`, seine tatsächliche Größe im Attribut `size`.

Sollte der Puffer zu klein werden, werden wir einen neuen alloziieren, den bestehenden Inhalt in den neuen Puffer kopieren und die Attribute `buf` und `size` aktualisieren.

Grundsätzlich werden wir am Ende immer das Endezeichen `'\0'` haben, sodass wir nur `size - 1` Zeichen im Puffer ablegen können.

Implementierung `cbuf.c`**:**

```
 1 #include <stdio.h>       // fprintf()
 2 #include <stdlib.h>      // malloc(), free(), exit()
 3 #include <string.h>      // memcpy()
 4 #include "cbuf.h"
 5
 6 #define INCR    128
 7
 8 static void *new_mem( int size, char *name )
 9        {
10           void *p;
11           if ((p = malloc( size )))
12              return p;           // everything all right
13           fprintf( stderr, "%s: out of memory\n", name );
14           exit( 1 );             // exit program
15        }
16
17 CB_PTR cb_reset( CB_PTR p )
18        {
19           p->buf[ p->len = 0 ] = '\0';
20           return p;
21        }
22
23 CB_PTR cb_init( CB_PTR p )
24        {
25           p->buf = new_mem( p->size=INCR, "cb_init()" );
26           return  cb_reset( p );
27        }
28
29 void cb_close( CB_PTR p )
30      {
31         free( p->buf );
32      }
33
34 CB_PTR cb_malloc()
35        {
36           return cb_init( new_mem( sizeof( CBUF ),
37                                    "cb_malloc()" ) );
38        }
39
40 void cb_free( CB_PTR p )
41      {
42         cb_close( p ); free( p );
43      }
```

Implementierung `cbuf.c` **(Fortsetzung):**

```
45 char *cb_addc( CB_PTR p, char c )
46      {
47         char *newbuf;
48         if ( p->len + 1 >= p->size )
49         {
50            newbuf = new_mem( p->size+INCR, "cb_addc()" );
51            memcpy( newbuf, p->buf, p->len + 1 ); // '\0'!
52            free( p->buf );
53            p->size += INCR;
54            p->buf   = newbuf;
55         }
56         p->buf[ p->len++ ] = c; p->buf[ p->len ] = '\0';
57         return p->buf;
58      }
59
60 char *cb_addstr( CB_PTR p, char *str )
61      {
62         while( *str )
63            cb_addc( p, *str++ );
64         return p->buf;
65      }
66
67 char *cb_buf( CB_PTR p )
68      {
69         return p->buf;
70      }
```

Ein Testprogramm `main.c`**:**

```
 1 #include <stdio.h>       // printf()
 2 #include "cbuf.h"
 3
 4 int main( int argc, char **argv )
 5     {
 6        CB_PTR p = cb_malloc();
 7        cb_addc( p, 'h' );
 8        cb_addc( p, 'i' );
 9        cb_addstr( p, " you" );
10        printf( "buf= '%s'\n", cb_addc( p, '!' ) );
11        cb_reset( p );
12        printf( "buf= '%s'\n", cb_addstr( p, "servus" ) );
13     }
Ausgabe: buf= 'hi you!'
         buf= 'servus'
```

4. Kodierung, vollständig objektorientiert

Um unsere Lösung vollständig objektorientiert zu machen, müssen wir unsere Typdefinition `CBUF` mit Funktionszeigern anreichern (siehe auch Skript, Kapitel 18) und die Funktionen `cb_init()` und `main()` entsprechend anpassen. Ferner können wir mit Ausnahme von `cb_init()` und `cb_malloc()` alle Methoden als `static` spezifizieren.

Typdefinition `CBUF` (`cbuf.h`):

```
 1  typedef struct _cbuf {
 2            int len;                 // length incl. '\0'
 3            int size;                // total size of buf
 4            char *buf;               // the acutal buffer
 5
 6            struct _cbuf *(* cb_reset)( struct _cbuf *p );
 7            void   (* cb_close )(struct _cbuf *p);
 8            void   (* cb_free  )(struct _cbuf *p);
 9            char  *(* cb_addc  )(struct _cbuf *p, char c);
10            char  *(* cb_addstr)(struct _cbuf *p, char *s);
11            char  *(* cb_buf   )(struct _cbuf *p);
12          } CBUF, *CB_PTR;
13
14  CB_PTR cb_init  ( CB_PTR p ); // constructor
15  CB_PTR cb_malloc();           // new CBUF and constructor
```

Die `cb_init()`-Methode (`cbuf.c`):

```
 1  CB_PTR cb_init( CB_PTR p )
 2         {
 3            p->buf = new_mem( p->size=INCR, "cb_init()" );
 4            p->cb_close  = cb_close ;
 5            p->cb_free   = cb_free  ;
 6            p->cb_reset  = cb_reset ;
 7            p->cb_addc   = cb_addc  ;
 8            p->cb_addstr = cb_addstr;
 9            p->cb_buf    = cb_buf   ;
10            return cb_reset( p );
11         }
```

Das Hauptprogramm `main()` (`main.c`):

```
    int main( int argc, char **argv )
        {
           CB_PTR p = cb_malloc();
           p->cb_addc( p, 'h' );
           ..................
        }
```