FH-OÖ Hagenberg/HSD
Betriebssysteme 3, WS 2025
Übung 6

Name:	**Marco Söllinger**	Aufwand in h:	**3**
Mat.Nr:	**s2410306011**	Punkte:	
Übungsgruppe:	**Gruppe 1**	korrigiert:	

Kommando-Interpreter (24 Punkte)

Ein Kommando-Interpreter, auch als Shell oder Befehlsinterpreter bezeichnet, ist eine Softwareanwendung in einem Betriebssystem, die Benutzern die Interaktion mit dem System durch die Eingabe von Befehlen ermöglicht. Der Kommando-Interpreter interpretiert die eingegebenen Befehle und führt die entsprechenden Aktionen aus, indem er Systemressourcen verwaltet, Programme ausführt oder andere Aufgaben erledigt.

Dieses Hilfsmodul erlaubt die Verwaltung von Funktionen, zugehöriger Hilfetexte, den Aufruf einzelner Funktionen samt Parameterübergabe und deren Exekution und Bewertung von Rückgabewerten.

Um den Implementierungsaufwand in Grenzen zu halten, wird ein vorgefertigtes Code-Template zur Verfügung gestellt. Es besteht aus den Modulen *CommandInterpreter* und *CommandTable*.

a) **Modul: *CommandInterpreter***
Die Modul-Schnittstelle des Kommando-Interpreters ist in der Header-Datei *CommandInterpreter.h* bereits vorgegeben.

Damit das Modul unabhängig von der verfügbaren Zeichenausgabe ist, erfolgt diese ausschließlich über eine konfigurierbare Funktion, die jeweils nur ein Zeichen übernimmt:

```
int PutChar(int ch);
```

Die `Print(...)`-Funktion in der Schnittstelle verwendet genau diese Funktion zur Ausgabe. Alle anderen Ausgabefunktionen dürfen nicht verwendet werden.

Initialisierung:
Das Modul bietet eine Initialisierungsfunktion: Parameter sind die Liste der verfügbaren Kommandos `TCmdHndTable` (inkl. Zusatzinformationen) und der Funktionszeiger der Ausgabefunkti-

on (z.B. `int putchar(int c)` aus `<stdio.h>`). Diese Daten werden in modulinternen Variablen verwaltet. Ein Begrüßungstext (z.B. "Welcome Command-Interpreter") und die Ausgabe des Promptzeichens (z.B. ">") signalisieren die Bereitschaft zur zeilenweisen Eingabe von Kommandos.

Kommandoeingabe:
Die Modulbenutzer rufen mit jedem Eintreffen eines Zeichens (`int getchar(void)`) eine Funktion `Process` auf, der als Parameter das Zeichen übergeben wird. Der Rückgabewert dieser Funktion gibt Aufschluss, ob die "Escape" Taste gedrückt wurde und somit die Kommandoeingabe beendet werden soll.

Die einzelnen Zeichen werden in einen modulinternen Puffer kopiert und zusätzlich als Echo über die Ausgabefunktion ausgegeben. Sobald das Zeichen LF (`"\n"`) oder CR (`"\r"`) detektiert worden ist, erfolgt die Analyse, ob im eingegebenen Text ein Kommando aus der definierten Liste enthalten ist. Je Eingabezeile ist ein Kommando erlaubt, das entweder direkt durch LF | CR oder durch ein Leerzeichen begrenzt ist. Wurde das Kommando in der Liste gefunden, erfolgt der Aufruf der zugehörigen Kommandofunktion und die Anzeige dessen Rückgabewertes. Wurde ein ungültiges Kommando eingegeben (nicht in der Liste enthalten), wird dies entsprechend angezeigt. Als Abschluss erfolgt wiederum die Ausgabe des Promptzeichens.

b) **Modul: *CommandTable***
Die Schnittstelle ist bereits vorgegeben unb besteht aus dem Typ der Kommando-Tabelle *TCmdHndTable* und einer Zugriffsfunktion *GetCmdHndTable()*, die einen Zeiger auf die Tabelle liefert.

Das Modul beinhaltet intern die vordefinierte Kommando-Tabelle mit allen Kommandos, wobei die internen Kommando-Funktionen bereits vordeklariert sind, die vollständige Implementierung fehlt und ist fertigzustellen.

Eine Kommandofunktion ist folgendermaßen definiert:

```
int Command(void);
```

Jedes Kommando liefert entsprechend seines Bearbeitungsergebnisses einen Rückgabewert: 0... fehlerhaft, sonst OK.

Kommandoverarbeitung:
Ein Kommando kann getrennt durch ein (oder mehrere) Leerzeichen von einem oder mehreren jeweils wieder durch Leerzeichen getrennte Parametern gefolgt werden. Hierfür bietet das Modul Funktionen in der TCmdHndTable

```
1 int CmdParamInt()     { ... }
2 int CmdParamFloat()   { ... }
3 int CmdParamString()  { ... }
4 int CmdParamAddInt()  { ... }
```

zur fortlaufenden Bewertung der Parameter als Integer (int), Fließkomma (float) oder Zeichen-

kette (string) an. Die Anzahl der zu lesenden Parameter ergibt sich aus dem Bedürfnis jedes einzelnen Kommandos, d.h. überzählige Parameter werden ignoriert.

Die einzelnen Kommando-Funktionen sind ebenfalls entsprechend der `TCmdHndTable` in diesem Modul implementiert und haben folgende Eigenschaften:

"help", "?" geben jeweils einen Hilfetext entsprechend obigem Vorbild aus.
"int" ein Parameter wird als Integer-Wert interpretiert und angezeigt.
"float" ein Parameter wird als Fließkomma-Wert interpretiert und angezeigt.
"string" ein Parameter wird als Zeichenkette interpretiert und angezeigt.
"addint" zwei Integer-Werte werden addiert und angezeigt.

c) **Tests**
Ausgabe: Ausgabe nach Aufruf des Kommandos **help**:

```
CmdHnd: Welcome!
>help
help
(press <ESC> to exit)
available commands:
help show help text
? show help text
int [value] first parameter as integer
float [value] first parameter as float
string [value] first parameter as string
addint [value1] [value2] adds two integer values
OK
>
```

Ausgabe nach Aufruf des Kommandos **int** mit Parameter 123:

```
>int 123
int 123
Parameter: 123
OK
>
```

Ausgabe nach Aufruf des Kommandos **float** mit Parameter 3.14:

```
>float 3.14
float 3.14
Parameter: 3.14
OK
>
```

Ausgaben nach Aufruf des Kommandos **string** mit Parameter "Donald":

```
>string Donald
string Donald
Parameter: Donald
OK
>
```

Ausgaben nach Aufruf des Kommandos **float** mit Parameter "Donald":

```
>float Donald
float Donald
ERROR
>
```

Ausgaben nach Aufruf des Kommandos **addint** mit den Parametern 3 und 4:

```
>addint 3 4
addint 3 4
Result: 3 + 4 = 7
OK
>
```

Versehen Sie Ihren Quelltext mit entsprechenden Kommentaren.

Allgemeine Hinweise: Legen Sie bei der Erstellung Ihrer Übung großen Wert auf eine **saubere Strukturierung** und auf eine **sorgfältige Ausarbeitung!** Dokumentieren Sie alle Schnittstellen und versehen Sie Ihre Algorithmen an entscheidenden Stellen ausführlich mit Kommentaren! Testen Sie ihre Implementierungen ausführlich! Geben Sie den **Testoutput** mit ab!

Beispiel 1

Es wird keine Lösungsidee gefordert, deshalb wurde der Code entsprechend kommentiert.

Das Programm wurde mit einer Makefile kompiliert.
Da in der Angabe nicht genau spezifiziert wurde, wie das Programm sich verhalten soll, wenn mehr Parameter als benötigt übergeben werden, wurde entschieden, dass die zusätzlichen Parameter ignoriert werden.
Weiters Falls ein Buffer Overflow auftritt, wird eine entsprechende Fehlermeldung ausgegeben und der Buffer wird geleert.
Es wird dann noch versucht die restlichen Symbole einzulesen und diese als Befehl zu interpretieren.

Ein paar Testfaelle wurden in der main.c Datei implementiert, der Rest wurdene manuell im Terminal getestet.

1.1 Code

main.c

```c
#include "CommandInterpreter.h"
#include "CommandTable.h"
#include <stdio.h>
#include <stdlib.h>

int main() {

  char ch = '\0';

  // Test Command Interpreter function without init
  Process('a');  // should print error message
  Print("Test"); // should print error message

  Init(NULL, putchar); // should print error message
  Process('a');        // should print error message
  Print("Test");       // should print error message

  Init(GetCmdHndTable(), NULL); // should print error message
  Process('a');                 // should print error message
  Print("Test");                // should print error message

  // initialize the command interpreter
  Init(GetCmdHndTable(), putchar);

  // read characters from input
  do {
    ch = getchar(); // wait for key
  } while (Process(ch) != cESC); // ESC typed?

  return EXIT_SUCCESS;
}
```

CommandInterpreter.h

```c
/*CommandInterpreter.h*/
#ifndef COMMANDINTERPRETER_H
#define COMMANDINTERPRETER_H
#include "CommandTable.h"

// Constants
#define MAX_INPUTLEN 128
#define MAX_PARAMS 16
#define MAX_PARAMLEN 32

// code for program quit
static int const cESC = 0x1B;

// return type of GetParams...holding a 2D Array
typedef struct {
  char params[MAX_PARAMS + 1][MAX_PARAMLEN + 1];
} TParameters;

// initialize Function. 1st Parameter: CommandTable, 2nd: a put char function
```

```
20 void Init(TCmdHndTable const cmds[], int (*PutCh)(int));
21
22 // provides access to all entered parameters of current command
23 // 1st parameter is command itself
24 TParameters GetParams();
25
26 // function that handles every single incoming char
27 int Process(char const ch);
28
29 // function which writes a string to the given put-char-function
30 void Print(char const *str);
31
32 #endif
```

CommandInterpreter.c

```
1  /*CommandInterpreter.c*/
2  #include "CommandInterpreter.h"
3  #include <stdio.h>
4  #include <string.h>
5
6  // module internal members
7  static TCmdHndTable const *gcmds = 0; // table of available commands
8  static int (*gpPutChar)(int); // function pointer for a put-char-function
9  // struct that holds all entered params to last cmd
10 static TParameters gParams = {0};       // set all to /0
11 static char gInput[MAX_INPUTLEN] = {0}; // modul internal puffer
12 static int gInputLen = 0;               // length of input string
13
14 // error messages
15 static char *cErrPrint = "Error in Print(...) -> null pointer in parameter\n";
16 static char *cErrInit = "Must initialize module before use!\n";
17 static char *cErrNoCmds =
18     "Error in Init(...) -> null pointer given for command table\n";
19 static char *cErrNoPutCh =
20     "Error in Init(...) -> null pointer given for put-char-function\n";
21 static char *cErrBufferOverflow =
22     "Input buffer overflow! Max input length exceeded.\n";
23 static char *cErrTooManyParams =
24     "Too many parameters! Max number of parameters exceeded.\n";
25 static char *cErrParamTooLong =
26     "Parameter too long! Max length of parameter exceeded.\n";
27 static char *cErrUnknownCmd = "Unknown command!\n";
28
29 static char *HelloText = "Welcome Command-Interpreter\n>";
30
31 ////////////////////////////////
32 // interface module functions
33 ////////////////////////////////
34
35 void Init(TCmdHndTable const cmds[], int (*PutCh)(int)) {
36   // for error checking here use fprintf to stderr because Print can't be used
37   // yet
38   if (cmds == 0) {
39     fprintf(stderr, "%s", cErrNoCmds);
40     return; // error: null pointer given
41   }
42   if (PutCh == 0) {
43     fprintf(stderr, "%s", cErrNoPutCh);
44     return; // error: null pointer given
45   }
46
47   gcmds = cmds;      // store command table
48   gpPutChar = PutCh; // store put-char-function pointer
49
50   Print(HelloText);
51 }
52
53 // pre declaration
54 static void evaluateCmd();
55 static void checkInput();
56
57 int Process(char const ch) {
58   if (gcmds == 0 || gpPutChar == 0) {
59     fprintf(stderr, "%s",
60             cErrInit); // Only here because can't really Print without init
```

```
61      return ch;          // error: module not initialized
62    }
63    if (ch == cESC) {
64      return cESC;
65    }
66
67    if (gInputLen >= MAX_INPUTLEN) {
68      Print(cErrBufferOverflow);
69      gInputLen = 0; // reset input length
70      return ch;
71    }
72
73    if (ch == '\r' || ch == '\n') {
74      gInput[gInputLen] = '\0';
75      checkInput();
76
77      gInputLen = 0;
78      Print("\n>");
79      return ch;
80    }
81
82    gInput[gInputLen++] = ch;
83
84    return ch;
85  }
86
87  TParameters GetParams() { return gParams; }
88
89  void Print(char const *str) {
90    if (gcmds == 0 || gpPutChar == 0) {
91      fprintf(stderr, "%s",
92              cErrInit); // Only here because can't really Print without init
93    }
94
95    // check parameter
96    if (str == 0) {
97      Print(cErrPrint);
98      return;
99    }
100
101   int i = 0;
102   if (gpPutChar != 0) {       // put char function pointer given?
103     while (str[i] != '\0') { // iterate given string
104       gpPutChar(str[i++]);
105     }
106   }
107 }
108
109 ////////////////////////////
110 // internal module functions
111 ////////////////////////////
112
113 // evaluates a completed command
114 static void evaluateCmd() {
115   int i = 0;
116
117   while (gcmds[i].func != 0) {
118     if (strcmp(gParams.params[0], gcmds[i].command) == 0) {
119       int ret = gcmds[i].func();
120       if (ret == 1) {
121         Print("OK");
122       } else {
123         Print("ERROR");
124       }
125       return;
126     }
127     i++;
128   }
129
130   Print(cErrUnknownCmd);
131 }
132
133 // checks the input for correctness
134 static void checkInput() {
135   char *ptr = gInput;
136   // Must do before to ensure paramIndex = 0 correct
```

```
  while ((*ptr == ' ' || *ptr == '\t') && *ptr != '\0') {
    ptr++; // skip leading spaces
  }

  if (*ptr == '\0') {
    Print(">");
    return;
  }

  memset(&gParams, 0, sizeof(TParameters));

  int paramIndex = 0;
  int charIndex = 0;

  while (*ptr != '\0' && paramIndex < MAX_PARAMS + 1) {

    if (*ptr != ' ' && *ptr != '\t') {
      // stop at -1 because we need space for null-terminator
      if (charIndex >= MAX_PARAMLEN - 1) {
        Print(cErrParamTooLong);
        return; // stop processing on error
      }

      gParams.params[paramIndex][charIndex] = *ptr;
      charIndex++;
      ptr++;

    } else {
      gParams.params[paramIndex][charIndex] = '\0'; // null-terminate parameter
      paramIndex++;

      if (paramIndex >= MAX_PARAMS) {
        Print(cErrTooManyParams);
        return; // stop processing on error
      }

      charIndex = 0;
      while ((*ptr == ' ' || *ptr == '\t') && *ptr != '\0') {
        ptr++; // skip leading spaces
      }
    }
  }

  for (int i = 0; i <= paramIndex; i++) {
    Print(gParams.params[i]);
    Print(" ");
  }

  Print("\n");
  evaluateCmd();
}
```

CommandTable.h

```
#ifndef COMMAND_TABLE
#define COMMAND_TABLE

// type of command table
typedef struct {
  int (*func)(void);
  char *command;
  char *description;
} TCmdHndTable;

// returns address of command table
TCmdHndTable const *GetCmdHndTable();

#endif // COMMAND_TABLE
```

CommandTable.c

```
#include "CommandTable.h"
#include "CommandInterpreter.h"
#include <memory.h>
#include <stdio.h>
#include <stdlib.h>
```

```
6
7  // return code for command functions
8  int const OK = 1;
9  int const NOK = 0;
10
11 // internal command functions
12 static int CmdHelp();
13 static int CmdParamInt();
14 static int CmdParamFloat();
15 static int CmdParamString();
16 static int CmdAddInt();
17
18 // CommandTable
19 static TCmdHndTable const cmds[] = {
20     {CmdHelp, "help", "show help text"},
21     {CmdHelp, "?", "show help text"},
22     {CmdParamString, "string", "[value] first parameter as string"},
23     {CmdParamInt, "int", "[value] first parameter as integer"},
24     {CmdParamFloat, "float", "[value] first parameter as float"},
25     {CmdAddInt, "addint", "[value1] + [value1] adds two integer values"},
26     {0, 0, 0} // end criteria
27 };
28
29 // returns address of command table
30 TCmdHndTable const *GetCmdHndTable() { return cmds; }
31
32 ////////////////////////////////
33 // internal command functions
34 ////////////////////////////////
35
36 // prints help text of all commandos
37 // I trust that cmds was filled correctly so no internal null-pointer checks
38 static int CmdHelp() {
39   int i = 0;
40   Print("(press <ESC> to exit)\n");
41   Print("available commands:\n");
42
43   while (cmds[i].func != 0) {
44     Print(cmds[i].command);
45     Print(": ");
46     Print(cmds[i].description);
47     Print("\n");
48     i++;
49   }
50
51   return OK;
52 }
53
54 // interprets 1st parameter as integer and prints it
55 static int CmdParamInt() {
56   TParameters p = GetParams();
57   char *param = p.params[1];
58
59   // kein Parameter
60   if (param[0] == '\0') {
61     return NOK;
62   }
63
64   char *endptr;
65   strtol(param, &endptr, 10);
66
67   // there was something that is not a number
68   if (param == endptr) {
69     return NOK;
70   }
71
72   // there are trailing characters
73   if (*endptr != '\0') {
74     return NOK;
75   }
76
77   Print("Parameter: ");
78   Print(param);
79   Print("\n");
80
81   return OK;
```

```
82  }
83
84  // interprets 1st parameter as float and prints it
85  static int CmdParamFloat() {
86    TParameters p = GetParams();
87    char *param = p.params[1];
88
89    // kein Parameter
90    if (param[0] == '\0') {
91      return NOK;
92    }
93
94    char *endptr;
95    strtof(param, &endptr);
96
97    // there was something that is not a number
98    if (param == endptr) {
99      return NOK;
100   }
101
102   // there are trailing characters
103   if (*endptr != '\0') {
104     return NOK;
105   }
106
107   Print("Parameter: ");
108   Print(param);
109   Print("\n");
110
111   return OK;
112 }
113
114 // interprets 1st parameter as string and prints it
115 static int CmdParamString() {
116   TParameters p = GetParams();
117   char *param = p.params[1];
118
119   // kein Parameter
120   if (param[0] == '\0') {
121     return NOK;
122   }
123
124   Print("Parameter: ");
125   Print(param);
126   Print("\n");
127
128   return OK;
129 }
130
131 // adds two integer parameter and prints the result
132 static int CmdAddInt() {
133   TParameters p = GetParams();
134   char *param1 = p.params[1];
135   char *param2 = p.params[2];
136
137   // brauchen zwei Parameter
138   if (param1[0] == '\0' || param2[0] == '\0') {
139     return NOK;
140   }
141
142   char *endptr1;
143   char *endptr2;
144
145   int value1 = (int)strtol(param1, &endptr1, 10);
146   if (param1 == endptr1 || *endptr1 != '\0') {
147     return NOK;
148   }
149
150   int value2 = (int)strtol(param2, &endptr2, 10);
151   if (param2 == endptr2 || *endptr2 != '\0') {
152     return NOK;
153   }
154
155   int sum = value1 + value2;
156
157   char buf[32];
```

```
158   sprintf(buf, "%d", sum);
159
160   Print("Result: ");
161   Print(buf);
162   Print("\n");
163
164   return OK;
165 }
```

1.2 Test

Es wurden folgende Testfälle implementiert und ausgeführt:

Terminal Output

```
1  flashfish@fedora ~/D/R/F/B/Uebung06 (main)> make run
2  ./CommandInterpreter
3  Must initialize module before use!
4  Must initialize module before use!
5  Error in Init(...) -> null pointer given for command table
6  Must initialize module before use!
7  Must initialize module before use!
8  Error in Init(...) -> null pointer given for put-char-function
9  Must initialize module before use!
10 Must initialize module before use!
11 Welcome Command-Interpreter
12 >help
13 help
14 (press <ESC> to exit)
15 available commands:
16 help: show help text
17 ?: show help text
18 string: [value] first parameter as string
19 int: [value] first parameter as integer
20 float: [value] first parameter as float
21 addint: [value1] + [value1] adds two integer values
22 OK
23 >?
24 ?
25 (press <ESC> to exit)
26 available commands:
27 help: show help text
28 ?: show help text
29 string: [value] first parameter as string
30 int: [value] first parameter as integer
31 float: [value] first parameter as float
32 addint: [value1] + [value1] adds two integer values
33 OK
34 >int 123
35 int 123
36 Parameter: 123
37 OK
38 >float 3.14
39 float 3.14
40 Parameter: 3.14
41 OK
42 >addint 3 4
43 addint 3 4
44 Result: 7
45 OK
46 >int 123 123
47 int 123 123
48 Parameter: 123
49 OK
50 >float 123 123
51 float 123 123
52 Parameter: 123
53 OK
54 >addint 5
55 addint 5
56 ERROR
57 >addint
58 addint
59 ERROR
60 >int
```

```
int
ERROR
>float
float
ERROR
>help 475
help 475
(press <ESC> to exit)
available commands:
help: show help text
?: show help text
string: [value] first parameter as string
int: [value] first parameter as integer
float: [value] first parameter as float
addint: [value1] + [value1] adds two integer values
OK
>adf
adf
Unknown command!

>
```

Terminal Output

```
flashfish@fedora ~/D/R/F/B/Uebung06 (main)> make run
./CommandInterpreter
Must initialize module before use!
Must initialize module before use!
Error in Init(...) -> null pointer given for command table
Must initialize module before use!
Must initialize module before use!
Error in Init(...) -> null pointer given for put-char-function
Must initialize module before use!
Must initialize module before use!
Welcome Command-Interpreter
>
    BufferOverflowTest1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890ABCDEFGHIJKL

Input buffer overflow! Max input length exceeded.
Input buffer overflow! Max input length exceeded.
Input buffer overflow! Max input length exceeded.
xyz1234567890
Unknown command!

>
```

Terminal Output

```
flashfish@fedora ~/D/R/F/B/Uebung06 (main) [SIGINT]> make run
./CommandInterpreter
Must initialize module before use!
Must initialize module before use!
Error in Init(...) -> null pointer given for command table
Must initialize module before use!
Must initialize module before use!
Error in Init(...) -> null pointer given for put-char-function
Must initialize module before use!
Must initialize module before use!
Welcome Command-Interpreter
>int 1Param 2Param 3Param 4Param 5Param 6Param 7Param 8Param 9Param 10Param 11Param 12Param 13
    Param 14Param 15Param 16Param 17Param 18Param 19Param 20Param
Input buffer overflow! Max input length exceeded.
m 18Param 19Param 20Param
Unknown command!

>
```

Terminal Output

```
flashfish@fedora ~/D/R/F/B/Uebung06 (main)> make run
clang  -Wall -Wextra -c CommandTable.c -o build/CommandTable.o
clang -g -o CommandInterpreter build/CommandInterpreter.o build/CommandTable.o build/main.o -
    lm
./CommandInterpreter
Must initialize module before use!
```

```
Must initialize module before use!
Error in Init(...) -> null pointer given for command table
Must initialize module before use!
Must initialize module before use!
Error in Init(...) -> null pointer given for put-char-function
Must initialize module before use!
Must initialize module before use!
Welcome Command-Interpreter
>    int 325
int 325
Parameter: 325
OK
>int         345
int 345
Parameter: 345
OK
>int Ha345
int Ha345
ERROR
>int 352Hal
int 352Hal
ERROR
>float 3h3
float 3h3
ERROR
>float 3.0
float 3.0
Parameter: 3.0
OK
>string Hallo353
string Hallo353
Parameter: Hallo353
OK
>string 343
string 343
Parameter: 343
OK
>addint 34h 3
addint 34h 3
ERROR
>add int 3 3h
add int 3 3h
Unknown command!

>addint 3 3h
addint 3 3h
ERROR
>
```