Übungspaket 4

Funktionszeiger

Übungsziele:

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

Literatur:

C-Skript[1], Kapitel: 85

Semester:

Wintersemester 2017/18

Betreuer:

Kevin, Peter 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.

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

Teil I: Stoffwiederholung

Aufgabe 1: „Gewöhnliche“ Zeiger

Um wieder in das Thema zu kommen, wiederholen wir hier ein paar wichtige Aspekte von Zeigern.

Was steht in einer Zeigervariablen?	Eine RAM Adresse
Gibt es kleine und große Adressen?	Nein
Sind alle Adressen gleich lang?	Ja
Sind Adressen immer hexadezimale Zahlen?	Nein
Warum sind Adressen meist hexadezimale Zahlen?	Es ist bequem und passt zum RAM

Im Zusammenhang mit Zeigern sind die damit verbundenen Typen von besonderer Wichtigkeit. Nehmen wir an, wir haben folgende Zeigerdefinitionen:

```
1 int    *p1, **p2;
2 char   *p3, **p4;
3 double ***p5;
```

Vervollständige folgende Tabelle:

Ausdruck	Typ	Verbale Beschreibung
`p1`	`int *`	Zeiger auf `int`
`*p1`	`int`	`int`
`**p1`	ungültig	der Zeiger wurde einmal zu oft dereferenziert
`p2`	`int **`	Zeiger auf Zeiger auf `int`
`*p2`	`int *`	Zeiger auf `int`
`**p2`	`int`	`int`
`***p2`	ungültig	der Zeiger wurde einmal zu oft dereferenziert
`p3`	`char *`	Zeiger auf `char`
`*p3`	`char`	`char`
`**p3`	ungültig	der Zeiger wurde einmal zu oft dereferenziert
`p4`	`char **`	Zeiger auf Zeiger auf `char`
`*p4`	`char *`	Zeiger auf `char`
`**p4`	`char`	`char`
`***p4`	ungültig	der Zeiger wurde einmal zu oft dereferenziert
`**p5`	`double *`	Zeiger auf `double`
`p5`	`double ***`	Zeiger auf Zeiger auf Zeiger auf `double`

Aufgabe 2: Funktionszeiger

Vervollständige die folgenden Sätze:

Ein `int`-Zeiger zeigt auf	ein `int`
Ein `double`-Zeiger zeigt auf	ein `double`
Ein Zeiger auf einen `int`-Zeiger zeigt auf	einen `int`-Zeiger
Ein Funktionszeiger zeigt auf	eine Funktion
Der Name eines Arrays repräsentiert	die Anfangsadresse des Arrays
Der Name einer Funktion repräsentiert	die Anfangsadresse der Funktion
Was bindet stärker, die `()` oder der `*`	die Klammern `()`

Beschreibe die beiden folgenden Definitionen:

`int *p1()`	`p1` ist eine Funktion, die einen `int`-Zeiger zurück liefert
`int (*p2)()`	`p2` ist ein Zeiger auf eine Funktion, die `int` zurück liefert

Beschreibe kurz mit eigenen Worten, wodurch der Unterschied in der Definition zustande kommt.

Im ersten Beispiel binden die Klammern `()` stärker an den Namen `p1` als der Stern `*`. Daher handelt es sich bei `p1` um eine Funktion. Der Rückgabewert dieser Funktion ist dann vom Typ `int *`.

Im zweiten Beispiel binden die linken Klammern `()` stärker als die rechten. Daher handelt es sich bei `p2` um einen Zeiger (die linken Klammern) auf eine Funktion (die rechten Klammern), die ein `int *` zurück liefert.

Nehmen wir an, wir hätten die beiden folgenden Defintionen:

```
1 double sin( double );
2 double (*fptr)( double );
```

Weise nun dem Funktionszeiger `fptr` die Funktion `sin` zu und rufe diese Funktion mittels des Funktionszeigers `fptr` und dem Argument `1.0` auf:

```
1 fptr = sin;              // Hierdurch wird der Zeigers fptr auf
2                          // die Funktion sin()
3
4 fptr( 1.0 );             // neue Syntax
5 (*fptr)( 1.0 );          // alte Syntax
```

Teil II: Quiz

Aufgabe 1: Funktionszeiger und deren Typen

In diesem Quiz wollen wir die mit den Funktionszeigern verbundenen Typen ein wenig üben. Nehmen wir an, wir haben folgendes C-Programm:

```
1 int (*f1)();
2 int (*f2)( int );
3 int (*f3)( double );
4 int *(*f4)();
5 int (*(*f5)())()
6 double sin( double );
```

Vervollständige nun die jeweiligen Typen:

Ausdruck	Typ	Verbale Beschreibung
`f1`	`int (*)()`	Zeiger auf eine Funktion, die ein `int` zurückgibt
`f1()`	`int`	`int` da die aufgerufene Funktion ein `int` liefert
`(*f1)()`	`int`	wie `f1()`, jedoch Funktionsaufruf mittels alter Syntax
`f2`	`int (*)( int )`	Zeiger auf eine `int`-Funktion mit einem `int` Parameter
`f2( 3 )`	`int`	`int`, da die aufgerufene Funktion, ein `int` liefert
`(*f2)(3)`	`int`	wie `f2( 3 )`, jedoch Funktionsaufruf mit alter Syntax
`f3`	`int (*)( double )`	Zeiger auf eine `int`-Funktion mit `double` Parameter
`f3( .1 )`	`int`	`int` da die aufgerufene Funktion ein `int` liefert
`f4`	`int (*)()`	Zeiger auf eine Funktion, die einen `int`-Zeiger liefert
`f4()`	`int *`	die aufgerufene Funktion, liefert einen `int`-Zeiger
`(*f4)()`	`int *`	wie `f4()` jedoch Funktionsaufruf mit alter Syntax
`*f4()`	`int`	der zurückgegebene Zeiger wird einmal dereferenziert
`*(*f4)()`	`int`	wie `*f4()`, jedoch Funktionsaufruf mit alter Syntax
`f5`	`int (*(*))()`	Zeiger auf eine Funktion, die einen Zeiger auf eine `int`-Funktion liefert
`f5()`	`int (*)()`	Rückgabe eines Zeigers auf eine `int`-Funktion
`sin`	`double (*)(double)`	`double`-Funktion mit einem `double`-Parameter
`sin(1.0)`	`double`	Die `sin()`-Funktion liefert ein `double` zurück

Teil III: Fehlersuche

Aufgabe 1: Definition und Verwendung von Funktionszeigern

Dr. Fun P. ist vom Konzept der Funktionszeiger total begeistert: *"Coole Geschichte, endlich mal ein anspruchsvolles Konzept mit mega viel Dynamik."* Ein kurzer Blick auf seine ersten Programmierversuche zeigt aber, dass er noch ein paar Schwierigkeiten mit der Definition sowie der Verwendung der Funktionszeiger hat; irgendwie hat er mit den Klammern und den Sternchen ein bisschen Mühe. Finde und korrigiere die Fehler in folgendem Programmstück. Da die Zeilen `1` bis `3` sowie `7` korrekt sind, befindet sich je ein Fehler in den Zeilen `8` bis `16`. Alle Funktionszeiger fangen immer mit `fp` an.

```
 1 double square(  double x ) { return x * x; }
 2 double qubic(   double x ) { return x * x * x; }
 3 double tripple( double x ) { return 3 * x; }
 4
 5 int main( int argc, char **argv )
 6     {
 7         double y;
 8         double fp1( double );          // Definition der Zeiger
 9         double *fp2( double );
10         double (*fp3);
11         fp1  = *square;                // konkrete Zuweisungen
12         fp2  =  qubic();
13         *fp3 =  tripple;
14         y = fp1;                       // konkrete Aufrufe
15         y = *fp2;
16         y = *fp3( 2.0 );
17     }
```

Zeile	Fehler	Erläuterung	Korrektur
8	`(*)` fehlen	`fp1()` *deklariert* die Funktion `fp1`, die ein `double` zurück gibt; der `*` für einen Zeiger und die `()` fehlen.	`(*fp1)(double)`
9	`()` fehlen	`*fp2()` *deklariert* eine Funktion namens `fp2`, die einen Zeiger auf ein `double` zurück gibt, da die `()` stärker als der `*` binden.	`(*fp2)(double)`
10	`()` fehlen	`(*fp3)` definiert einen `double`-Zeiger, keinen Zeiger auf eine Funktion, da die hinteren `()` fehlen.	`(*fp3)(double)`

Zeile	Fehler	Erläuterung	Korrektur
11	* ist zu viel	Durch den * wird an „das Ende“ des Funktionszeigers gegangen; gesucht ist hier aber die Adresse der Funktion.	square
12	() sind zu viel	Durch die Klammern() wird die Funktion qubic *aufgerufen*; gesucht ist hier aber die Adresse der Funktion.	qubic
13	* ist zu viel	Der Zeigervariablen fp3 muss der Wert tripple zugewiesen wird; durch *fp3 wird dieser Wert an irgend eine Stelle des Speichers geschrieben.	fp3
14	(2.0) fehlt	fp1 ist nur die Anfangsadresse der Funktion, stellt aber keinen Funktionsaufruf dar.	fp1(2.0)
15	(2.0) und () fehlen	*fp2 geht an „das Ende“ des Zeigers, stellt aber keinen Funktionsaufruf dar.	fp2(2.0)
16	vordere () fehlen	Hier wird zwar die Funktion aufgerufen, aber das Ergebnis wird dereferenziert. Bei der alten Syntax fehlen noch die vorderen ().	(*fp3)(2.0)

Programm mit Korrekturen:

```
 1 double square(  double x ) { return x * x; }
 2 double qubic(   double x ) { return x * x * x; }
 3 double tripple( double x ) { return 3 * x; }
 4
 5 int main( int argc, char **argv )
 6     {
 7         double y;
 8         double (*fp1)( double );        // Definition der Zeiger
 9         double (*fp2)( double );
10         double (*fp3)( double );
11         fp1 = square;                    // konkrete Zuweisungen
12         fp2 = qubic;
13         fp3 = tripple;
14         y = fp1( 2.0 );                       // konkrete Aufrufe
15         y = fp2( 2.0 );
16         y = (*fp3)( 2.0 );         // alte syntax; neu: fp3(2.0)
17     }
```

Teil IV: Anwendungen

Aufgabe 1: Ein erstes Programm mit Funktionszeigern

1. Aufgabenstellung

Schreibe ein einfaches Programm, das eine einfache quadratische Funktion $f(x) = x^2$ implementiert. Diese Funktion soll nicht über ihren Namen sondern mittels eines zusätzlichen Funktionszeigers aufgerufen werden. Überprüfe mittels neun ausgewählter x-Werte aus dem Intervall $x \in [-4..4]$, ob der Funktionsaufruf mittels Funktionszeiger zum richtigen Resultat führt.

2. Vorüberlegungen

Die folgenden Vorüberlegungen sind für Fortgeschrittene recht naheliegend:

1. Wir kodieren eine quadratische Funktion `double sphere( double x )`, die als Ergebnis den Wert `x * x` liefert.
2. Im Hauptprogramm definieren wir einen Funktionszeiger `fp` und setzen ihn auf die Funktion `sphere()`.
3. Im Hauptprogramm lassen wir das Argument von `-4` bis `4` in Einerschritten laufen und geben die Funktionswerte aus, die wir über den Funktionszeiger sowie durch den direkten Funktionswert erhalten.

3. Kodierung

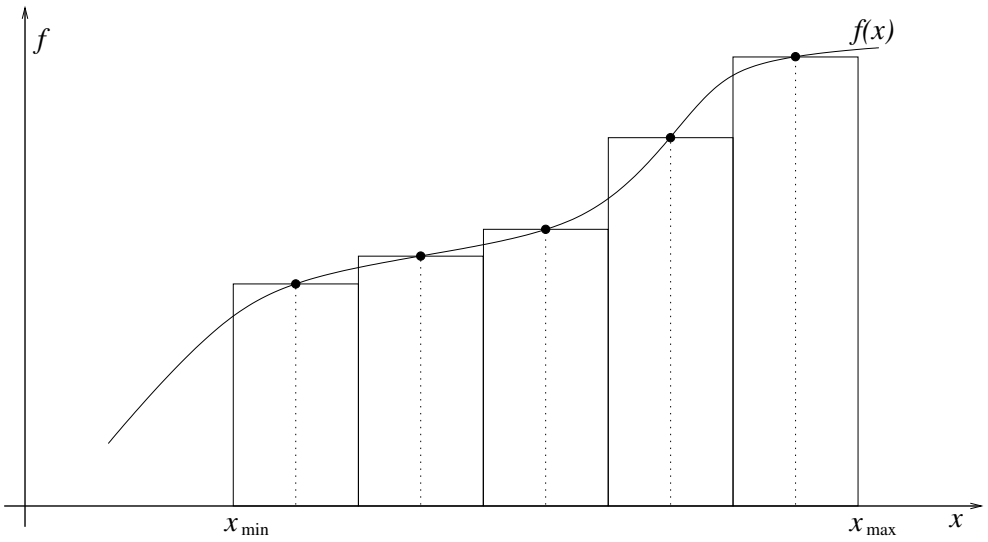
```
 1 #include <stdio.h>
 2
 3 double sphere( double x )
 4        {
 5           return x * x;
 6        }
 7
 8 int main( int argc, char **argv )
 9     {
10        double x, (*fp)( double );
11        fp = sphere;
12        for( x = -4; x <= 4.5; x += 1.0 )
13          printf( "x=%7.3f fp(x)=%7.3f sphere(x)=%7.3f\n",
14                   x, fp( x ), sphere( x ) );
15        return 0;
16     }
```

Aufgabe 2: Numerische Integration von Funktionen

In dieser Aufgabe werden wir eine C-Funktion schrittweise entwickeln, die eine gegebene mathematische Funktion $f(x)$ innerhalb eines gegebenen Intervalls numerisch integriert. Die wesentliche Idee dabei ist, dass wir die Integration für unterschiedliche mathematische Funktionen verwenden wollen, ohne ihre Implementierung mehrmals hinzuschreiben. Eben, es soll eine C-Funktion sein. Da aber die konkreten Werte der mathematischen Funktion immer innerhalb dieser Integrier Funktion bestimmen müssen, müssen wir die zu integrierende mathematische Funktion $f(x)$ irgendwie als Parameter in geeigneter Form übergeben. „Überraschenderweise“ wird dies in Form eines Funktionszeiger geschehen. Zunächst aber werden wir die Grundlagen der *numerischen* Integration kurz wiederholen.

1. Wiederholung: numerische Integration

Die numerische Integration wird verwendet, wenn die zu integrierende Funktion nicht in analytischer Form bekannt ist. In diesem Falle wird die Fläche als Summe kleiner Rechtecke angenähert. Dabei wird die Höhe des Rechtecks durch den Funktionswert in seiner Mitte definiert. Dieser Sachverhalt ist in folgendem Bild veranschaulicht, in dem das Integral von $x_{\min}$ bis $x_{\max}$ durch fünf Rechtecke angenähert wurde.



2. Aufgabe: Bestimmung der einzelnen Rechtecke

Die numerische Integration besteht aus den drei Teilen: Berechnung der Stützstellen x_i, Berechnung der einzelnen Rechtecke $F_i = \Delta x f(x_i)$, und Berechnung der Fläche $F = \sum_i F_i$. Zunächst gehen wir davon aus, dass die zu integrierende Funktion $f(x)$ bekannt und fest einprogrammiert ist. Somit benötigen wir eine Funktion `double integrate( double xmin, double xmax, int slices )`. Im ersten Schritt soll diese Funktion *lediglich* die einzelnen Stützstellen x_i berechnen und ausgeben. Getestet werden soll diese Funktion mit geeigneten Parameterkombinationen.

```
 1 #include <stdio.h>
 2
 3 double integrate( double xmin, double xmax, int slices )
 4        {
 5           double x, delta_x = (xmax - xmin)/slices;
 6           int i;
 7           for( i = 0; i < slices; i++ )
 8           {
 9              x = xmin + 0.5 * delta_x + i * delta_x;
10              printf( "x= %e\n", x );
11           }
12           return 0.0;  // in order to please the compiler
13        }
14
15 int main( int argc, char ** argv )
16     {
17        integrate( 0.0, 1.0,  1 );              // Test 1
18        integrate( 0.0, 1.0, 10 );              // Test 2
19     }
```

3. Aufgabe: Die erste Flächenberechnung

Nach dem wir nun das Grundgerüst unserer Funktion `integrate()` haben, können wir uns der Flächenberechnung widmen. Hierzu müssen wir die Ausgabe der Stützstellen (der x_i) durch die eigentliche Flächenberechnung ersetzen. Um das Testen weiterhin möglichst einfach zu gestalten, wählen wir als zu integrierende Funktion die Identität $\text{ident}(x) = x$ und rufen sie in direkt auf: `ident( x ) * delta_x;`

Funktionen `ident()` und `integrate()`:

```
 1 #include <stdio.h>
 2
 3 double ident( double x ){ return x; }
 4
 5 double integrate( double xmin, double xmax, int slices )
 6        {
 7           double x, sum, delta_x = (xmax - xmin)/slices;
 8           int i;
 9           for( sum = 0.0, i = 0; i < slices; i++ )
10           {
11              x = xmin + 0.5 * delta_x + i * delta_x;
12              sum += ident( x ) * delta_x;
13           }
14           return sum;
15        }
```

Hauptprogramm (`main()`) zu Testzwecken:

```
1  int main( int argc, char ** argv )
2      {
3          printf( "integrate( 0.0, 1.0,  1 )= %e\n",
4                   integrate( 0.0, 1.0,  1 ));
5          printf( "integrate( 0.0, 1.0, 10 )= %e\n",
6                   integrate( 0.0, 1.0, 10 ));
7      }
```

4. Aufgabe: Flächenberechnung für beliebige, eindimensionale Funktionen

Im letzten Schritt müssen wir unsere Funktion `integrate()` so erweitern, dass sie beliebige, eindimensionale Funktionen numerisch integrieren kann. Dazu müssen wir die zu integrierende Funktion als Parameter übergeben, der offensichtlich ein Funktionszeiger sein muss. Nach Anpassung der Parameterliste (Signatur) soll sie mit den Funktionen $f(x) = x^2$, $f(x) = x^3$ und der vorhandenen Funktion $f(x) = \sin(x)$ getestet werden. Nebenbei haben wir die Funktion `integrate()` ein wenig gekürzt.

```
1  #include <stdio.h>
2  #include <math.h>
3
4  double sphere( double x ){ return x * x; }
5  double qubic(  double x ){ return x * x * x; }
6
7  double integrate( double (*fp)( double ),
8                    double xmin, double xmax, int slices )
9          {
10             double sum, delta_x = (xmax - xmin)/slices;
11             int i;
12             for( sum = 0.0, i = 0; i < slices; i++ )
13                sum += fp(xmin + 0.5*delta_x + i * delta_x);
14             return sum * delta_x;
15         }
16
17 int main( int argc, char ** argv )
18     {
19         printf( "x^2: 0.0 .. 2.0,  60 )= %e\n",
20                  integrate( sphere, 0.0, 2.0, 60 ));
21         printf( "x^2: 0.0 .. 2.0, 200 )= %e\n",
22                  integrate( sphere, 0.0, 2.0, 200 ));
23         printf( "x^3: 0.0 .. 1.0, 100 )= %e\n",
24                  integrate( qubic, 0.0, 1.0, 100 ));
25         printf( "sin: 0.0 ..  pi, 200 )= %e\n",
26                  integrate( sin, 0.0, M_PI, 200 ));
27     }
```

5. Aufgabe: Entwicklung eines Moduls

Für die Wiederverwendung einer derartigen Funktion `integrate()` ist es sinnvoll, diese in Form eines Moduls, bestehend aus einer entsprechenden `.h` und `.c`-Datei, zu organisieren. Dies ist genau jetzt die Aufgabe ;-) Teste mit 99 Rechtecken.

Schnittstelle (`.h`-Datei) des Moduls `integration`:

```
1  /*
2   * integrate.h:
3   * interface of the module integrate.c for the numerical
4   * integration of one-dimensional functions that are
5   * passed as a function pointer.
6   */
7
8  double integrate( double (*fp)( double ),
9                    double xmin, double xmax, int slices );
```

Implementierung (`.c`-Datei) des Moduls `integration`:

```
1  /*
2   * integrate.c: implementation of the func. integrate()
3   */
4
5  #include "integrate.h"
6
7  double integrate( double (*fp)( double ),
8                    double xmin, double xmax, int slices )
9         {
10           double sum, delta_x = (xmax - xmin)/slices;
11           int i;
12           for( sum = 0.0, i = 0; i < slices; i++ )
13              sum += fp(xmin + 0.5*delta_x + i * delta_x);
14           return sum * delta_x;
15        }
```

Hauptprogramm (`main()`) zu Testzwecken:

```
1  #include <stdio.h>
2  #include <math.h>
3  #include "integrate.h"
4
5  int main( int argc, char ** argv )
6      {
7         printf( "sin: %e\n", integrate(sin, 0, M_PI, 99));
8      }
```

Aufgabe 3: Funktionszeiger und `typedef`

Wie wir weiter oben im Quiz gesehen haben, kann die Definition von Funktionszeigern recht schnell komplex werden. Hier kann die Verwendung von `typedef` ein wenig Abhilfe schaffen. Dies versuchen wir in dieser Aufgabe ein wenig zu beleuchten. Beschreibe zunächst mit eigenen Worten und anhand zweier „normaler“ Beispiele wie `typedef` funktioniert.

Beschreibung: Mittels `typedef` kann man neue, komplexere Typen definieren und anschließend wie jeden anderen Datentyp verwenden.

Syntax: `typedef bekannter_Typ neuer_Typ [, neuer_Typ [...]]`

Beispiele:

```
typedef int  *INT_PTR; int  i; INT_PTR ip = & i; // Zeiger auf ein int
typedef char *CHR_PTR; char c; CHR_PTR cp = & c; // Zeiger auf ein char
```

Nun können wir uns mit den Funktionszeigern beschäftigen. Definiere einen neuen Typ `FNC_PTR`, der ein Zeiger auf eine `double`-Funktion mit einem `double`-Argument ist:

```
typedef double (*FNC_PTR)( double );
```

Definiere eine Variable (also einen Zeiger) dieses neuen Typs und lass ihn auf die bekannte Funktion `sin()` zeigen:

```
FNC_PTR fp; fp = sin;
```

Rufe die `sin()`-Funktion mittels dieses Zeigers auf und übergebe ihr das Argument `0.5`:

neue Syntax: `fp( 0.5 )`
alte Syntax : `(*fp)( 0.5 )`

Definiere nun die folgenden Datentypen:

1. Zeiger auf eine Funktion, die einen `double`-Zeiger liefert:

   ```
   typedef double *(*DZ_FNCP)();
   ```

2. Zeiger auf eine Funktion, die einen Zeiger auf eine `int`-Funktion liefert:

   ```
   typedef int      (*INT_FNCP)();
   typedef INT_FNCP (*INT_FNCPP)();
   ```

3. Zeiger auf eine `char`-Funktion, die einen `int`-Zeiger als Argument hat:

   ```
   typedef int  INT_PTR;
   typedef char CHAR_FNCP( INT_PTR )();
   ```

4. Zeiger auf eine Funktion, die einen Zeiger auf eine `int`-Funktion liefert:

   ```
   typedef int      (*INT_FNCP)();
   typedef INT_FNCP (*INT_FNC_FNC)();
   ```

Aufgabe 4: Abstrakter Datentyp

So weit, so gut. In der letzten Aufgabe wollen wir die Konzepte `struct` und Funktionszeiger zusammenführen. Wir werden also `structs` definieren, die neben Variablen auch Funktionszeiger beinhalten. Auch hier werden wir uns der Problematik wieder in Form eines interaktiven Tutorials nähern.

Nehmen wir nun an, wir bräuchten eine Schwellwertfunktion f_step(x, a, b), die für Argumente $x \geq a$ den Wert b ansonsten den Wert 0 hat. Erstelle zunächst eine Implementierung dieser Schwellwertfunktion:

```
1 double f_step( double x, double a, double b )
2        {
3           return (x >= a)? b: 0.0;
4        }
```

Erstellen wir gleich noch eine weitere Funktion f_cos$(x, a, b) = a\cos(x + b)$:

```
1 double f_cos( double x, double a, double b )
2        {
3           return a * cos(x + b);
4        }
```

Ein Blick auf die Funktionsköpfe zeigt, dass beide die gleichen Signaturen (Parameterlisten) besitzen, was wir später noch brauchen.

Nehmen wir nun an, dass unser Programm zwei unterschiedliche Schwellwertfunktionen sowie eine cos-Funktion benötigt. Eine sehr einfache Implementierung würde beispielsweise je drei Variablen für die Konstanten a und b definieren, ihnen Werte zuweisen und sie bei den Funktionsaufrufen übergeben. Das sähe beispielsweise wie folgt aus:

```
1 int main( int argc, char **argv )
2     {
3        double x, a1, a2, a3, b1, b2, b3;
4        a1=1.0; b1=2.0; a2=2.0; b2=1.0; a3=0.5; b3=0.0;
5        for( x = 0; x < 3.75; x += 0.25 )
6           printf( "x= %e step_1= %e step_2= %e cos= %+e\n",
7                   x, f_step( x, a1, b1 ),
8                   f_step( x, a2, b2 ), f_cos( x, a3, b3 ) );
9     }
```

So würde es funktionieren. Aber so richtig prickelnd ist es nicht. Bei genügend vielen Funktionen wird die Zahl der Parameter a_i und b_i recht unübersichtlich. Ferner ist nur schwer ersichtlich, welche Dinge zusammen gehören, was die Wartung und Lesbarkeit eines derartigen Programms erschwert Also schlagen wir diesen Weg nicht ein, sondern bringen zusammen, was zusammen gehört.

„Zusammenbringen? Die Parameter a_i und b_i könnte ich ja einfach in einem Array ablegen, da sie alle vom selben Datentyp sind :-) Aber wie soll ich diese mit den Funktionen zusammen bringen? Ich kann die Parameter doch nicht in die Funktionen direkt hinein schreiben! Dann bräuchte ich für jeden neunen Fall eine neue Funktion, die intern nur ein paar andere Parameter hat. Das wäre ja sinnlos.“ Simmt! Aber wie können wir Dinge unterschiedlicher Datentypen zusammen bringen und wie kann man Funktionen außer über ihren Namen noch ansprechen? *„Ach ja, da gibt's ja noch diese* **`struct`***s und die Funktionszeiger. Misst, ich wusste doch, dass ich das hätte ordentlich machen sollte. verdammt.“*

Genau, wir definieren einen `struct`, in dem sich die Parameter a_i und b_i sowie ein Zeiger auf die konkrete Funktion befindet. *„Jo, so mit den ganzen Signaturen und dem ganzen anderen Gedöns?“* Genau, probier einfach mal. *„Puh ...“*:

```
1 typedef struct {
2           double a, b;
3           double (*fnc)(double, double, double);
4         } FNC;
```

Gar nicht mal schlecht. Wie würde jetzt obiges Hauptprogramm aussehen, wenn wir nur die erste Schwellwertfunktion bräuchten?

```
1 int main( int argc, char **argv )
2     {
3        double x;
4        FNC    fnc = { f_step, 1.0, 2.0 };
5        for( x = 0; x < 3.75; x += 0.25 )
6           printf( "x= %e step_1= %e\n", x,
7                   fnc.fnc( x, fnc.a, fnc.b ) );
8     }
```

Und was sagst du zu deinem eigenen Resultat? *„Na ja, eigentlich schon so ein bisschen überzeugend. Schön ist ja Zeile 4: Definition eines* **`struct`***s mit gleichzeitiger Initialisierung seiner Kompontenten. Aber wenn ich länger auf die acht Programmzeilen schaue, dann gefällt mir ehrlich gesagt Zeile 7 nicht so richtig: das sieht einfach umständlich und wartungs*unfreundlich *aus.“* Stimmt, perfekt ist es noch nicht. Aber was stört dich denn an Zeile 7? *„Na ja, Wenn ich in meinem* **`struct`** *noch mehr Komponenten habe, wird's länger und komplizierter, Zumal es die aufrufende Stelle eigentlich gar nichts angeht, wie die einzelnen Komponenten in Wirklichkeit heißen und wie man auf sie zugreift.“* Ja, gar nicht schlecht bemerkt. Völlig richtig. Du willst also diese Detailinformationen des `struct`s verbergen und nur „innerhalb“ des Funktionszeigers verwenden? *„Ja!“* Schön, du hast viel gelernt! Was müssten wir also machen, damit wir nicht alle Komponenten einzeln übergeben müssen? *„Einfach das ganze* **`struct`** *oder vielleicht noch besser einen Zeiger auf dieses* **`struct`** *übergeben.“* Genau. Dann pass doch mal die Typdefinition, die Schwellwertfunktion sowie das Hauptprogramm entsprechend an:

```
 1 typedef struct _fnc {
 2           double (*fnc)(struct _fnc *, double);
 3                                 // function( struct pointer, x )
 4           double a, b;                          // the parameters
 5         } FNC, *FNC_PTR;
 6
 7 double f_step( FNC_PTR this, double x )
 8        {
 9           return (x >= this->a)? this->b: 0.0;
10        }
11
12 int main( int argc, char **argv )
13     {
14        double x;
15        FNC    fnc = { f_step, 1.0, 2.0 };
16        for( x = 0; x < 3.75; x += 0.25 )
17           printf( "x= %e step_1= %e\n", x, fnc.fnc(& fnc, x));
18     }
```

Und, was meinst du selbst zu deinem Werk? *„Ein bisschen komplex und ungewohnt, aber dennoch irgendwie geil, vor allem Zeile 17 :-)“* Genau, eigentlich recht schön.

Die Verwendung weiterer Funktionen innerhalb von `main()` sollte jetzt kein Problem mehr darstellen. Ebenso sollte klar sein, wie man die Struktur `FNC` um weitere Zugriffsfunktionen erweitern kann. Hierzu zählen beispielsweise Funktionen zum Einlesen der Parameter oder zur Ausgabe eines spezifischen Kommentars.

Von hier ist es jetzt nur noch ein ganz kleiner Schritt zum objektorientierten Programmieren. Letztlich haben wir mittels der Struktur `FNC` eine Klasse definiert, wobei die Komponente `fnc` eine Methode und die Komponenten `a` und `b` Attribute darstellen, und mittels `fnc` unser erstes Objekt instanziiert. Aber das ist Gegenstand der nächsten Vorlesungen und Übungen.