Modellierung und Programmierung

Dr. Martin Riplinger

19.12.2012

Funktionszeiger: Vorüberlegungen

- Funktionsaufrufe sind bis jetzt im Code **mit Name explizit** angegeben („hartcodiert“)
- **Folge**: Zur Compilierzeit muss bekannt sein, **welche Funktion** eine bestimmte Aufgabe erfüllen soll.

- **Beispielszenario**: Wir haben einen Sortieralgorithmus geschrieben, der ein Feld von Zahlen sortiert. Dabei wollen wir flexibel bestimmen können, nach welchem **Vergleichskriterium** sortiert werden soll (größer, kleiner, 5. Ziffer in Dezimaldarstellung größer, Quersumme kleiner, . . .).
- Mit den bisherigen Mitteln müsste im Algorithmus nach Vergleichskriterien unterschieden werden, z. B. mittels `if-else` oder `switch`.
- Alternativ müsste für jedes Vergleichskriterium eine separate Funktion geschrieben werden.
- **Einleuchtendere Herangehensweise**:
  - Es gibt einen **allgemeinen** Sortieralgorithmus, der das Feld nach einer bestimmten Methode durchsucht und beim Vergleich zweier Elemente **irgendein** (variables!) Vergleichskriterium heranzieht.
  - Zum Vergleich gibt es eine Reihe von **Vergleichsfunktionen**, die getrennt vom Sortierverfahren deklariert und definiert sind.
  - Beim Aufruf des Sortieralgorithmus wird eine der Vergleichsfunktionen **als Parameter** mit angegeben.

  Genau dieses Verhalten lässt sich mit Funktionszeigern erzeugen!

Funktionszeiger:Deklaration

`Datentyp (*FktZeigerName)(Parameter(typ)liste);`

Deklariert einen Zeiger auf eine Funktion, welche die Signatur

`Datentyp Funktion(Parameter(typ)liste)` besitzt.

Beispiele:

- `void (*InitFkt)(double *, double);`

  Zeiger auf eine Funktion, die einen `double` *- und einen `double`-Parameter nimmt und nichts (`void`) zurückgibt.

- `double (*ZufGen)(void)`

  Zeiger auf eine Funktion, die keine Parameter nimmt und einen `double`-Wert zurückgibt.

- `double *(*NeuesFeld)(unsigned)`

  Zeiger auf eine Funktion, die einen `unsigned`-Parameter nimmt und einen Pointer auf `double` zurückgibt.

Funktionszeiger: Beispiel

Deklaration und Definition von zwei Anzeigevarianten

```
1  #include <stdio.h>
2
3  void anzeigeVariante1(char *text);
4  void anzeigeVariante2(char *text);
5
6  //-------------------------------------------------------------------------
7
8  void anzeigeVariante1(char* text)
9  {
10   printf("\n %s\n", text);
11   return;
12 }
13
14 //-------------------------------------------------------------------------
15
16 void anzeigeVariante2(char* text)
17 {
18   printf("\n **********************************");
19   printf("\n * %-30s *\n", text);
20   printf(" **********************************\n");
21   return;
22 }
```

Funktionszeiger: Beispiel

Hauptprogramm

```
1  int main(void)
2  {
3    void (*AnzFkt)(char *);
4
5    AnzFkt = anzeigeVariante1;
6    AnzFkt("Test Variante 1");
7
8    AnzFkt = anzeigeVariante2;
9    AnzFkt("Test Variante 2");
10
11   return 0;
12 }
```

Ausgabe:

```
Test Variante 1

**********************************
* Test Variante 2                *
**********************************
```

⇝ Vor dem Funktionsnamen in der Zuweisung steht **kein** &-Operator. Wie bei statischen Feldern hätte er keine Auswirkung.

⇝ Beim Aufruf der Funktion via Pointer wird **kein** *-Operator benötigt. Die Schreibweise `(*AnzFkt)(...)` wäre äquivalent, **nicht** jedoch `*AnzFkt(...)`! Wieso?

Funktionszeiger: Bemerkungen

- Bei Zuweisungen mit Funktionspointern ist **unbedingt** darauf zu achten, dass die **Signaturen von Pointer und Funktion übereinstimmen**. Alles andere führt zu unkontrollierbarem Programmverhalten! Vor Fehlern dieser Art **warnt** der Compiler bestenfalls.
- Deklarationen von Zeigern auf Funktionen mit langer Parameterliste werden leicht unleserlich (vor allem bei Funktionen mit Funktionszeigern als Parameter):

```
void funktion(int a, double b,
              double *(*f)(double, int, int, double *, double *));

int main(void)
{
  double *(*fp)(double, int, int, double *, double *);
  fp = testfkt; // testfkt sei passend deklariert
  funktion(42, 3.14, fp);
  return 0;
}
```

Ein `typedef` schafft Abhilfe:

```
typedef double *(*MeineFkt)(double, int, int, double *, double *);
void funktion(int a, double b, MeineFkt f);
```

Deklaration der Funktionspointer-Variable in `main`: `MeineFkt fp;`

Rangfolge von Operatoren (Überblick)

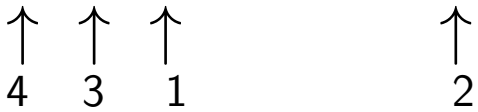
Priorität	Operator	Bedeutung	Assoziativität
Priorität 1	()	Funktionenaufruf	linksassoziativ
	[]	Array/Index-Operator	
	. , ->	Member-Zugriff	
Priorität 2	!	Logische Negation	rechtsassoziativ
	$++$, $--$	Inkrement, Dekrement	
	$-$, $+$	Unäres Plus, unäres Minus	
	&	Adress-Operator	
	*	Dereferenzierung	
	(type)	Cast	
Priorität 3	* , /	Multiplikation, Division	linksassoziativ
	%	Modulo	
Priorität 4	$+$, $-$	Plus, Minus	linksassoziativ
Priorität 6	$<, <=, >, >=$	kleiner, ...	linksassoziativ
Priorität 7	$==$, $!=$	gleich, ungleich	linksassoziativ
Priorität 11	&&	logisches UND	linksassoziativ
Priorität 12	||	logisches ODER	linksassoziativ

Merkregeln und lesen von Deklarationen

Merke:

- Rechtsassoziativ sind lediglich: Zuweisungsoperatoren, Bedingungsoperator (? :) und unäre Operatoren.
- Sinnvolle Klammerungen können die Lesbarkeit von Code deutlich erhöhen!

```
int * (*Funkzeiger)();
```



Resultat:
Funkzeiger ist ein Zeiger (1) auf eine Funktion (2) mit leerer Parameterliste, die einen Zeiger (3) auf Integer (4) zurückgibt.

Publikumsfrage

Was deklarieren die folgenden Statements?

```
double *f(double *, int);
```

⇝ Funktion, die als Parameter einen `double *` und einen `int` nimmt und einen `double *` zurückgibt.

```
double (*f)(double *, int);
```

⇝ Zeiger auf eine Funktion, die als Parameter einen `double *` und einen `int` nimmt und einen `double` zurückgibt.

```
double *(*f)(double *, int);
```

⇝ Zeiger auf eine Funktion, die als Parameter einen `double *` und einen `int` nimmt und einen `double *` zurückgibt.

```
double *g[20];
```

⇝ g ist Feld mit 20 Einträgen vom Typ `double *`.

Funktionszeiger: Beispiel Sortierverfahren

In `stdlib.h` ist die folgende Sortierfunktion deklariert („**quicksort**“):

```
void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));
```

Parameter:

base	Zu sortierendes Feld eines (noch) nicht festgelegten Datentyps
nmemb	Anzahl der Feldelemente
size	Größe eines Feldelements in Byte
compar	Zeiger auf eine (Vergleichs-)Funktion, die zwei `void *`-Zeiger auf zu vergleichende Elemente als Parameter nimmt und einen `int` zurückgibt.

Interpretation: `compar` repräsentiert eine mathematische Ordnungsrelation „$\preceq$“ auf einer Menge M, d. h. für zwei beliebige Werte $a, b \in M$ gilt entweder $a \preceq b$, $b \preceq a$ oder beides. Die zu vergleichenden Elemente von `base` stammen aus M.

Der Rückgabewert von `compar` ist `-1` ($a \preceq b$), `0` (Gleichheit) oder `+1` ($b \preceq a$), wobei a dem ersten und b dem zweiten Parameter von `compar` entspricht.

Funktionszeiger: Beispiel Sortierverfahren

Anwendung:

```c
#include <stdio.h>
#include <stdlib.h>

int unsign_qsumme_kleiner(const void *pa, const void *pb);

int main(void)
{
  unsigned z[5] = {23, 511, 10100, 8, 333};
  qsort(z, 5, sizeof(unsigned), unsign_qsumme_kleiner);

  printf("Sortiertes Feld:\n");
  printf("%d %d %d %d %d\n", z[0], z[1], z[2], z[3], z[4]);
  return 0;
}

int unsign_qsumme_kleiner(const void *pa, const void *pb)
{
  unsigned a = *((unsigned *) pa), b = *((unsigned *) pb);
  unsigned qs_a = a % 10, qs_b = b % 10;

  while (a /= 10)
    qs_a += a % 10;

  while (b /= 10)
    qs_b += b % 10;

  return (qs_a < qs_b)? -1 : (qs_b < qs_a)? +1 : 0;
}
```

Funktionszeiger: Beispiel Sortierverfahren – Codeanalyse

Deklaration und Definition der Sortierfunktion

- `unsign_qsumme_kleiner` soll die Quersumme von `unsigned`-Werten vergleichen.
- Die Signatur **muss** `int fkt(const void *, const void *)` sein.
- Im Funktionsrumpf werden die Parameter `pa` und `pb` zunächst als `unsigned *` **gecastet**, anschließend **dereferenziert** und die **Werte** in `unsigned`-Variablen `a` und `b` **geschrieben**. Exemplarisch für `pa`:

  ```c
  unsigned a = *( (unsigned *) pa );
  ```

- In den Schleifen werden die Quersummen von `a` und `b` berechnet und in `qs_a` bzw. `qs_b` gespeichert.

  Beachte: `while (a /= 10)` führt zuerst Ganzzahl-Division durch und prüft anschließend, ob das Ergebnis ungleich `0` ist (in diesem Fall besteht `a` noch nicht aus einer einzigen Dezimalziffer).

- Die `return`-Zeile verwendet die verkürzte Fallunterscheidung

  ```c
  (Bedingung)? Wert_Falls_Ja : Wert_Falls_Nein;
  ```

Beispiel: `absx = (x > 0)? x : -x;` speichert genau wie

`if (x > 0) absx = x; else absx = -x;` den Betrag von `x` in `absx`.

Funktionszeiger: Beispiel Sortierverfahren – Codeanalyse

Hauptprogramm

- Das Feld `z` mit 5 `unsigned`-Einträgen soll aufsteigend bzgl. der Quersumme sortiert werden.
- Dazu wird `qsort` mit den Parametern `z` (`base`), `5` (`nmemb`), `sizeof(unsigned)` (`size`) und `unsign_qsumme_kleiner` (`compar`) aufgerufen:

```
qsort(z, 5, sizeof(unsigned), unsign_qsumme_kleiner);
```

- **Ausgabe**:

```
Sortiertes Feld:
10100 23 511 8 333
```

Quod erat expectandum!

- **Fazit**: Wie `qsort` genau funktioniert, ist hier völlig unerheblich. Entscheidend ist, **dass** die Funktion ein Feld anhand einer gegebenen Vergleichsroutine sortiert.

Kommandozeilenargumente

Bisher:

```
$ gcc -o ProgName C_Code.c [-Wall]
$ ./ProgName
  ⋮
```

Neu: Komandozeilenargumente

- Die `main`-Funktion lässt sich auch mit zwei Parametern aufrufen.
- Vollständige Deklaration von `main`:

```
int main(int argc, char *argv[])
```

`argc` **arg**ument **c**ounter: Anzahl der beim Programmaufruf übergebenen Argumente, einschließlich des Programmnamens. Erst wenn `argc` > 1 ist, werden tatsächlich Parameter übergeben.

`argv` **arg**ument **v**ector: Feld von Strings

Bemerkung: Die Namen `argc` und `argv` (auch üblich: `args`) sind Konvention und nicht zwingend festgelegt. Möglich (aber wenig sinnvoll) wäre auch

```
int main(bla, blubb)
```

Kommandozeilenargumente

Beispiel: `argumente.c`

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  int i;

  printf("Anzahl der Argumente: %d\n\n", argc);

  for (i=0; i<=argc; i++)
    printf("Argument %d: %s\n", i, argv[i]);

  return 0;
}
```

Ausgabe: Der Aufruf `./argumente arg1 arg2 --help +~ 125.777 -99` liefert

```
Anzahl der Argumente: 7

Argument 0: ./argumente          Argument 4: +~
Argument 1: arg1                 Argument 5: 125.777
Argument 2: arg2                 Argument 6: -99
Argument 3: --help               Argument 7: (null)
```

Kommandozeilenargumente

Beachte:

- Parameter werden durch Leerzeichen getrennt. Ausdrücke wie `-o Ausgabe` werden als zwei separate Argumente aufgefasst.
- Jedes Argument wird als Zeichenkette in `argv` gespeichert. Die Reihenfolge der Strings in `argv` entspricht dabei der Reihenfolge auf der Kommandozeile.
- Werden Zahlen als Argumente übergeben, müssen diese mit Hilfe der entsprechenden Umwandlungsfunktionen (z. B. `atoi` oder `atof`) in ein Zahlenformat konvertiert werden.
- Mit String-Vergleichsfunktionen (z. B. `strncmp`) lässt sich beispielsweise prüfen, ob ein Programm mit einem bestimmten Optionsparameter aufgerufen wurde. Da jedoch die Reihenfolge der Optionsargumente festgelegt ist, benötigen Programme mit vielen Optionen einen flexibleren Ansatz zur Auswertung der Kommandozeile ($\rightarrow$ *Parser*).

Kommandozeilenargumente: Beispiel

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){
  double x, y, z;
  if (argc < 4){            // Es fehlen Argumente
    printf("\nKorrekter Aufruf: ");
    printf("%s zahl1 op zahl2\n", argv[0]);
    return 1;
  }
  x = atof(argv[1]);
  y = atof(argv[3]);
  switch (argv[2][0]) {
    case '+':
        z = x + y; break;
    case '-':
        z = x - y; break;
    case 'x':
    case '*':
        z = x * y; break;
    case '/':
        z = x / y; break;
    default:
      printf("\nFalscher Operator! ABBRUCH!\n");
      return -1;
  }

  printf("\n%s %s %s = %lf", argv[1], argv[2], argv[3], z);
  return 0;
}
```

Kommandozeilenargumente

```
$ gcc -o berechne taschenrechner.c -Wall

$ ./berechne 2 + 5

2 + 5 = 7.000000



$ ./berechne 2 x 5

2 x 5 = 10.000000



$ ./berechne 2 / 5

2 / 5 = 0.400000



$ ./berechne 2 /

Korrekter Aufruf: ./berechne zahl1 op zahl2
```