Prof. Dr. G. Büchel Bonn, 1.6.2012

Arbeitsblätter: Kap.15: C – Programmierung für Java – Kenner

0. Vorbemerkung

Die Programmiersprache C ist den 1970iger Jahren entwickelt worden und somit eine deutlich ältere Programmiersprache als Java. Bei der Entwicklung von Java wurde vieles, das in C gut ist, übernommen. Der Grund, warum wir uns hier mit C beschäftigen, liegt darin, dass C in der hardwarenahen Programmierung (noch) geschätzt wird. In dieser kurzen Übersicht werden Sprachkonstrukte von C unter den Fragen diskutiert: Welche Konstrukte stimmen in C im wesentlichen mit gleichnamigen Konstrukten in Java überein? Welche Konstrukte sind in C verschieden zu Konstrukten in Java? Bei meiner Darstellung folge ich dem sehr schönen Buch[1] von Prof. Dr. Vogt, das ich Ihnen, wenn Sie sich in der Zukunft mit der C - Programmierung beschäftigen werden, empfehlen möchte. Wir betrachten hier C als reine prozedurale Sprache, wie sie mehrfach genormt wurde, z.B. als ANSI-C und **nicht** in Verbindung mit dem später entwickelten objektorientierten Sprachzusatz **C++**.

1. Datentypen

Zum Programmieren braucht man Datentypen. Es gibt in **C** elementare und höhere Datentypen. Wichtige **elementare Datentypen** in C sind **int**, **long**, **float**, **double** und **char**. Ihr Gebrauch für ganze Zahlen (int, long), für rationale Zahlen (float, double) und für Zeichen entspricht dem Java.

a) Bei **float** und **double** gibt es keine Unterschiede zu Java.
b) Bei **int** und **long** gibt es Unterschiede bezüglich des Speicherbereichs, den sie im RAM belegen. Diese Unterschiede variieren in C mit Prozessortypen und Betriebssystemen. Hieran wird deutlich: C ist **keine** plattformunabhängige Sprache. **int** wird mit 2 bzw. 4 Byte, **long** wird in der Regel mit 4 Byte implementiert. Auf den WINDOWS-Rechnern des Labors belegt **int** 4 Byte RAM-Platz.
c) Während mit **char** in Java jedes UNICODE- Zeichen gespeichert werden kann (2 Byte), kann char in C nur ASCII-Zeichen speichern (1 Byte).
d) In C gibt es **kein** Datentyp boolean. Der Wahrheitswert **falsch** wird durch die ganze Zahl 0, der Wahrheitswert **wahr** durch eine ganze Zahl ungleich 0, in der Regel ist es 1 repräsentiert. Vergleiche geben 1 bei wahr (z.B. bei 19 > 17) oder 0 bei falsch (z.B. 31 != 31) zurück.
e) In C besteht die Möglichkeit, ganzahlige Datentypen als vorzeichenlos (**unsigned**) zu für die Verwaltung von nicht-negativen Zahlen zu deklarieren. Man gewinnt dabei eine Zweierpotenz in der Größenordnung der damit darstellbaren Zahlen. (Z.B. geht der Wertebereich von **unsigned int** auf den WINDOWS-Rechnern des Labors bis 2^{32}-1, während **int** nur bis 2^{31}-1 ging.)

2. Operatoren

Alle folgenden Operatoren, die Sie aus Java kennen, werden mit der gleichen Bedeutung in C verwendet:

a) **Numerische** Operatoren: +, -, *, / . Bei ganzzahligen Ausdrücken kann, wie in Java auch % für die Berechnung des ganzahligen Rests verwendet werden. Bei Inkrementierung und Dekrementierung kann auch ++ bzw. -- verwendet werden.
b) **Vergleichsoperatoren**: <, >, ==, !=, <=, >= .
c) **Aussagenlogische** Operatoren: ! (nicht), && (und), || (oder (OR)), ^ (XOR). Weiterhin gibt es die bitweise wirkenden Operatoren: & (und), | (oder (OR)).
d) Der **Zuweisungsoperator**: x = x + 7;

[1] [VOGT] Carsten Vogt: „C für Java-Programmierer“, München (Hanser), ISBN 978-3-446-40797-8, 2007.

e) Der **Cast**-Operator für die explizite Konvertierung des Ausdrucks A in einen anderen Datentyp DTYP einer Zielvariable x: x = (DTYP)A;

3. Anweisungen

C ist wie Java eine blockstruktuierte Programmiersprache. Einzelanweisungen werden mit einem Semikolon (**;**) abgeschlossen. Mehrere Anweisungen können in einem Block zusammengefasst werden, der in geschweiften Klammern steht: **{ … }**.

C verfügt genau über die gleichen Steuerungsanweisungen wie Java, die in der gleichen Syntax formuliert sind und für den Programmablauf genau das Gleiche bewirken:

a) Die **for**-Schleife: Z.B.: **for** (i=0; i<n; i++) { /* Anweisungsblock */ … }
b) Die **while**-Schleife: Z.B.: **while** (a>b) { /* Anweisungsblock */ … }
c) Die **do-while**-Schleife: Z.B.: **do** { /* Anweisungsblock */ …} **while** (i==0);
d) Die **if**-Anweisung: Z.B.: **if** (b!=0) { /* Anweisungsblock */ …}
else { /* Anweisungsblock */ …}
e) Die **switch**-Anweisung: Z.B.: **switch**(i)
{case 1: /* Anweisungsfolge */ … **break;**
case 2: /* Anweisungsfolge */ … **break;**
… … …
default: /* Anweisungsfolge */ …}
f) Die Abbruch-Anweisung **break;** für Schleifen und Anweisungsfolgen in switch()-Fällen.
g) Die Anweisung für den Abbruch eines Schleifendurchlaufs: **continue;**

4. Ausgaben auf die Konsole / Tastatureingaben / Präcompileranweisung

Anders als in Java wird in **C** die Ausgabe auf die Konsole (Bildschirm) bzw. das Einlesen von der Tastatur mit dem Aufruf von Systemfunktionen ausgeführt. Systemfunktionen sind in den **C-Funktionsbibliotheken** (engl. libraries) enthalten und gehören zur Umgebung des C-Kompilers. Die C-Funktionsbibliotheken enthalten den auf das konkrete Betriebssystem bezogenen ausführbaren Objektcode der jeweiligen Systemfunktion. Damit der Kompiler den Objektcode der im Quelltext aufgerufenen Systemfunktion zuordnen kann, ist eine entsprechende **Präcompileranweisung** erforderlich, die die sogenannte **Header-Datei** der zugehörigen Funktionsbibliothek benennt und in den Quelltext beim Kompilieren einfügt. Alle für die Ein- und Ausgaben relevanten Systemfunktionen, werden über die Header-Datei **<stdio.h>** angesprochen (**h** steht für Headerdatei, **stdio** für die Bibliothek der **St**andard-**I**nput/-**O**utput Funktionen). Die hier notwendige Präcompileranweisung lautet:
#include <stdio.h>

Die allgemeine Syntax der Funktion **printf()** für formatierte Ausgaben auf die Konsole lautet:
printf(FORMATSTRING, Variablenliste);

Der **FORMATSTRING** besteht in der Regel aus konstantem Text, Platzhaltersymbolen für zu schreibende Variablenwerte gemäß ihres Datentyps und Formatierungszeichen (z.B. **\n** für Zeilenumbruch). Die **Variablenliste** enthält die Folge der Variablen, deren Werte auszugeben sind. Es können z.B. die folgenden Platzhaltersymbole verwendet werden:

Datentyp	int (dezimal)	float	double	unsigned int	unsigned long	char	Zeichenkette
Platzhaltersymbol	`%d`	`%f`	`%lf`	`%u`	`%lu`	`%c`	`%s`

BSP.1: Ausgabe von int x=17; und double p=3.14; : **printf("x = %d , p = %lf \n",x,p);**

Um den Wert einer Variablen **x** von der Tastatur einzulesen, kann man die Funktion **scanf()** verwenden, die Daten formatiert einliest. Ihre allgemeine Syntax lautet:
scanf(FORMATSTRING, Adressliste);

Wir betrachten zunächst nur eine einfache Form der Verwendung von scanf(). Hierbei ist PHS ein Platzhaltersymbol gemäß obiger Tabelle: Der Befehl **scanf("PHS",&x);** liest eine Zeichenfolge von der Tastatur ein, die durch ein RETURN abgeschlossen wird, konvertiert die Zeichenfolge gemäß des Platzhaltersymbols PHS in einen Wert in interner Bitdarstellung und schreibt dann den Wert an die RAM-Adresse von **x**. Die RAM-Adresse von x wird durch das Symbol **&x** dargestellt. Das Zeichen **&** nennt man den **Adressoperator**.

BSP.2: Eingabe einer ganzen Dezimalzahl von der Tastatur: **int a; scanf("%d",&a);**

5. Die main()-Funktion / C-Programme kompilieren

Ein ausführbares **C-Programm** benötigt eine **main()-Funktion**. Die main()-Funktion entspricht der **main()-Methode** in einem ausführbaren Java-Programm. Genau wie im modularen Entwurf von Java-Programmen werden Funktionen in C durch ihren **Prototyp** spezifiziert:

Prototyp der main()-Funktion: int main (void)

Im Unterschied zu Java wird in C erwartet, dass die main()-Funktion eine ganze Zahl an das Betriebssystem zurückgibt. Man kann auch die main()-Funktion mit dem Rückgabedatentyp **void** (leere Rückgabe), aber da gibt der Standard-Compiler **gcc** (Gnu C Compiler) eine Warnung aus. Man kann die main()-Funktion auch ohne explizite Angabe von void als Argumentdatentyp programmieren, wie es in den nachfolgenden Beispielen ausgeführt ist.

BSP.3: Ein ganz einfaches Programm, das nur eine Ausgabe auf die Konsole schreibt:

```
/*************************************************************/
/* Verfasser: Prof. Dr. Gregor Büchel                        */
/* Source   : Q1.c                                           */
/* Zweck    : Einfaches main() Programm                      */
/* Stand    : 01.06.2012                                     */
/*************************************************************/
#include <stdio.h>
int main ()
{
     printf("Noch lacht im Feld der frohe Hase,\n");
     printf("das Programm ist schon in der 1. Phase.\n");
     return 0;
}
```

Wir nutzen hier in der Lehrveranstaltung den **gcc** (Gnu C Compiler), den es sowohl für WINDOWS- als auch für LINUX-Betriebssysteme gibt und der unter einer PUBLIC-Lizenz steht[2]. Der **gcc** führt alle Schritte, die zur Erzeugung eines ausführbaren Programms erforderlich sind, aus. Das sind die folgenden Schritte:

I. Präkompilieren (z.B. Einfügen der Header-Dateien für System-Funktionen).
II. Kompilieren (Erzeugung eines maschinenabhängigen Objektcodes).
III. Linken (Zusammenfügen des Objektcodes des aktuellen C-Programms mit den Objektcodes der benutzten System-Funktionen). Das Ergebnis des Linkens ist eine ausführbare Datei (engl. executable, deshalb hat diese Datei die Endung **.exe**). Die **.exe**-Datei wird unter Angabe des Dateinamens (**.exe** kann dabei auch weggelassen werden) direkt von der Betriebssystemkommandozeile aus aufgerufen. Wenn man den **gcc** mit dem Parameter **–o** aufruft, kann man damit den Namen der **.exe**-Datei festlegen. Vergisst man dieses, legt der Linker das ausführbare Programm unter dem Namen **a.exe** (in Linux unter dem Namen **a.out**) an.

[2] Man kann den **gcc** für WINDOWS incl. der Entwicklungsumgebung Dev-C++ (in ihrer Leistungsfähigkeit ähnlich dem Java-Editor) beziehen von folgender URL:
http://www.bloodshed.net/dev/devcpp.html

Im nachfolgenden RUNTIME-Protokoll ist der Kompileraufruf, mit dem alle drei Schritte I., II. und III. ausgeführt werden und die Ausführung des Programms **Q1.exe** dokumentiert:

```
C:\gbC>gcc -o Q1 Q1.c

C:\gbC>Q1.exe
Noch lacht im Feld der frohe Hase,
das Programm ist schon in der 1. Phase.

C:\gbC>
```

BSP.4: Ein einfaches C-Programm, mit dem alle Potenzen $\mathbf{y = a^k}$ einer natürlichen Zahl **a** für alle $\mathbf{0 \le k \le n}$ berechnet, wobei **n** von der Tastatur eingelesen wird:

```
/*************************************************************/
/* Verfasser: Prof. Dr. Gregor Büchel                        */
/* Source   : PotA.c                                         */
/* Zweck    : Berechnung von Potenzen zu Basis a             */
/* Stand    : 01.06.2012                                     */
/*************************************************************/
#include <stdio.h>
int main ()
{int i,n,a=3, y=1;
 printf("Berechnung der Potenzen von %d:\n",a);
 printf("Eingabe des höchsten Exponenten:\n");
 do
 { printf("Bitte geben Sie eine natuerliche Zahl ein:\n");
   scanf("%d",&n);
   if (n<1)
   { printf("%d ist keine natuerliche Zahl. Neueingabe!\n",n);
   }
 } while(n<1);
 for (i=0; i<=n; i++)
 { printf("%d hoch %d = %d \n",a,i,y);
   y=y*a;
 }
}
```

6. Funktionen

Funktionen in **C** entsprechen den statischen Methoden in Java. Sie dienen der Modularisierung von Programmen. Sie werden durch ihren **Prototyp** spezifiziert. Der Prototyp ist vor dem Quelltext der main()-Methode anzugeben, wenn aus der main()-Methode Funktionen aufgerufen werden.

Allgemeine Syntax eines Funktionsprototypen: dtR fname(dt1 v1, dt2 v2, …, dtN vN);

Hierbei ist: **dtR** : Datentyp des Rückgabewerts der Funktion.
fname : Funktionsname
dti : Datentyp der Übergabevariable **vi** (für alle $1 \le i \le N$)

Genauso wie in Java gibt es in **C** Funktionen, die „nichts“ zurückgeben, diese haben dann wie in Java den Datentyp **void**. Andererseits gibt es wie in Java auch Funktionen, die keine Übergabewerte benötigen, hier schreibt man als Argument void (Prototyp: dtR fname(**void**);) oder man lässt wie in Java die Klammer leer.

Der Aufruf einer Funktion geschieht ähnlich wie in Java. Er ist sogar einfacher, man benötigt nicht mehr den Klassennamen der Klasse, der die statische Methode angehört. Für eine Funktion, die nicht void als **dtR** hat und deren Rückgabewert man in einer Variable **x** empfangen will, sieht die allgemeine Syntax des Funktionsaufrufs folgendermaßen aus:

dtR x;
x=fname(w1,w2, …, wN); Hierbei sind für alle i ($1 \le i \le N$) **wi** Werte vom Datentyp **dti**.

BSP.5: Wir nehmen das C-Programm aus **BSP.4**, mit dem alle Potenzen $\mathbf{y = a^k}$ einer natürlichen Zahl **a** für alle $\mathbf{0 \le k \le n}$ berechnet werden, und modularisieren es dahingehend, dass sowohl die Basis **a** als auch **n** mittels einer Funktion **einNat()** von der Tastatur eingelesen werden: **Prototyp: int einNat();**

```
/*************************************************************/
/* Verfasser: Prof. Dr. Gregor Büchel                        */
/* Source   : PotEN.c                                        */
/* Zweck    : Berechnung von Potenzen zu Basis a             */
/* Stand    : 01.06.2012                                     */
/*************************************************************/
#include <stdio.h>
/* Prototyp */
int eiNat();
int main ()
{int i,n,a, y=1;
 printf("Berechnung von natuerlichen Potenzen :\n");
 printf("Eingabe der Basis :\n");
 a=einNat();
 printf("Eingabe des höchsten Exponenten:\n");
 n=einNat();
  for (i=0; i<=n; i++)
 { printf("%d hoch %d = %d \n",a,i,y);
   y=y*a;
 }
}

int einNat()
{int k;
 do
 { printf("Bitte geben Sie eine natuerliche Zahl ein:\n");
   scanf("%d",&k);
   if (k<1)
   { printf("%d ist keine natuerliche Zahl. Neueingabe!\n",k);
   }
 } while(k<1);
 return k;
}
```

7. Felder (Arrays)
Felder sind in **C** genau wie in Java Referenzdatentypen. Im Unterschied zu Java wird nicht zwischen Deklaration einer Feldvariable und Speicherplatzbeschaffung beim Anlegen eines Feldes unterschieden, beides wird bei einer Felddeklarationsanweisung zusammen ausgeführt. Ist **N** eine gegebene **natürliche** Zahl, die die Anzahl der Feldkomponenten bestimmt, dann lautet die **allgemeine Syntax des Anlegens** eines Feldes, dessen Komponenten vom Datentyp **dtyp** sind: **dtyp feldname[N];**

BSP.6: Wir führen hier ein Programm zur Vektorrechnung vor, in dem die Vektorlänge und Vektoren x[] und y[] von der Tastatur eingelesen werden, der Vektorbetrag |**x**| unter Verwendung der Wurzelfunktion **sqrt()** aus der Bibliothek der mathematischen Funktionen (deshalb wird die Präcompileranweisung **#include <math.h>; benötigt.**) berechnet wird. Weiterhin wird der Summenvektor der Vektoren x[] und y[] durch die Funktion **vektadd()** berechnet, die nach dem **Call by Reference** (CbR) Prinzip arbeitet. Ihr Prototyp ist:
int vektadd(double x[], double y[], int n);

Ihr werden die Felder x[] und y[], die Referenzvariablen sind, d.h. für Adressen stehen, übergeben. Die Funktion vektadd() schreibt den Summenvektor in das Feld x[], das der aufrufenden Funktion noch zur Verfügung steht, da sie das Feld x[] angelegt hat und damit über die Adresse, unter der die Speicherzellen von x[] liegen, verwaltet.

```
/*************************************************************/
/* Verfasser: Prof. Dr. Gregor Büchel                        */
/* Source   : Vektor1.c                                      */
/* Zweck    : Anlegen und Einlesen eines Vektors, Berechnung */
/*            seines Betrages und entspr. Konsolenausgaben.  */
/*            Einlesen eines zweiten Vektors, Berechnung der */
/*            Vektorsumme mittels der Methode vektadd(), die */
/*            nach dem CbR-Prinzip arbeitet und den Summenvek-*/
/*            tor im Feld x[] zurueckgibt.                   */
/* Stand    : 01.06.2012                                     */
/*************************************************************/

#include <stdio.h>
#include <math.h>

/* Prototypen */
int einNat();
void vektrech(int n);
void vektaus(double z[], int l);
double vektbet(double x[], int m);
int vektadd(double x[], double y[], int n);

int main ()
{int n;
 printf("Vektorrechnung:\n");
 printf("Eingabe der Vektorlaenge:\n");
 n=einNat();
 printf("Vektorlaenge=%d\n",n);
 vektrech(n);
 return 0;
}

int einNat()
{int k;
 do
 { printf("Bitte geben Sie eine natuerliche Zahl ein:\n");
   scanf("%d",&k);
   if (k<1)
   { printf("%d ist keine natuerliche Zahl. Neueingabe!\n",k);
   }
 } while(k<1);
 return k;
}

void vektrech(int n)
{/* Deklaration und Speicherplatzbeschaffung eines Feldes */
 double x[n], y[n], a;
 int i;
 for(i=0; i<n; i++)
 { printf("Eingabe x[%d]: \n",i);
   scanf("%lf",&x[i]);
 }
```

```
 vektaus(x,n);
 a=vektbet(x,n);
 printf("|x|=%lf \n",a);
 printf("Geben Sie einen zweiten Vektor ein:\n");
 for(i=0; i<n; i++)
 { printf("Eingabe y[%d]: \n",i);
   scanf("%lf",&y[i]);
 }
 printf("Die Vektorsumme von x[] und y[]: \n");
 vektadd(x,y,n);
 vektaus(x,n);
}

void vektaus(double x[], int n)
{ int i;
  for(i=0; i<n; i++)
  { printf("x[%d]=%lf \n",i,x[i]);
  }
}

double vektbet(double x[], int n)
{ double w=0.;
  int i;
  for(i=0; i<n; i++)
  { w=w+x[i]*x[i];
  }
  w=sqrt(w);
  return w;
}

int vektadd(double x[], double y[], int n)
{ int i;
  for(i=0; i<n; i++)
  { x[i]=x[i]+y[i];
  }
  return n;
}
```

Anm.: Genau wie bei Java wird in **C** bei Funktionsaufrufen zwischen **Call by Value (CbV)**, wo alle Übergabevariablen von einem einfachen Datentyp sind und somit nur skalare Werte übergeben werden können, die durch die aufgerufene Funktion nicht abgeändert werden können, und **Call by Reference (CbR)** unterschieden, wo mindestens eine Übergabevariable von einem Referenzdatentyp ist und somit beim Aufruf eine Adresse übergeben wird. Dadurch bleiben Änderungen auf den Speicherzellen, wie z.B. auf den Feldkomponenten x[i] auch immer noch wirksam, wenn die aufgerufene Funktion (hier z.B. vektadd()) beendet ist.

Funktion	**Aufruf-art**	**Referenzvariablen**
`int einNat()`	CbV	-
`void vektrech(int n)`	CbV	-
`void vektaus(double z[], int l)`	CbR	z[]
`double vektbet(double x[], int m)`	CbR	x[]
`int vektadd(double x[], double y[], int n)`	CbR	x[] , y[]

8. Kompilieren von Funktionen

Funktionen sind in **C** die kleinsten komplierbaren Einheiten. In Java waren die Klassen die kleinsten kompilierbaren Einheiten. Wie werden dieses an folgendem Beispiel zeigen. Wir nehmen den Quelltext **PotEN.C** aus **BSP.5** und zerlegen ihn in zwei Quelltexte: a) In den Quelltext **PotM.C**, der nur die **main()**-Funktion aus PotEN.C mit den Aufrufen der Funktion einNat() zur Eingabe der Basis **a** und des höchsten Exponents **n** und mit der Berechnung der Potenzen enthält. b) in den Quelltext **EinNat.C**, der nur die Funktion **einNat()** enthält. Nachfolgend ist das Runtime-Protokoll gegeben, das folgende Schritte enthält:

(1) Getrenntes Präkomplieren und Kompilieren der Quelltexte EinNat.C und PotM.C. Hieraus entstehen die Objektdateien **EinNat.o** und **PotM.o**.
(2) Linken der Objektdateien EinNat.o und PotM.o (durch den Aufruf: **`gcc -o PotM PotM.o EinNat.o`**). Dadurch entsteht die ausführbare Datei **PotM.exe**.
(3) Ausführung des Programms durch Laden der **exe**-Datei. (**`C:\gbC>PotM.exe`**).

```
C:\gbC>gcc -c EinNat.c

C:\gbC>gcc -c PotM.c

C:\gbC>gcc -o PotM PotM.o EinNat.o

C:\gbC>PotM.exe
Berechnung von natuerlichen Potenzen :
Eingabe der Basis :
Bitte geben Sie eine natuerliche Zahl ein:
7
Eingabe des h÷chsten Exponenten:
Bitte geben Sie eine natuerliche Zahl ein:
10
7 hoch 0 = 1
7 hoch 1 = 7
7 hoch 2 = 49
7 hoch 3 = 343
7 hoch 4 = 2401
7 hoch 5 = 16807
7 hoch 6 = 117649
7 hoch 7 = 823543
7 hoch 8 = 5764801
7 hoch 9 = 40353607
7 hoch 10 = 282475249

C:\gbC>
```

9. Zeiger (Pointer)

Zeiger sind in **C** die Möglichkeit, Referenzen zu verwalten. In Java hat man dieses mit Variablen getan, deren Datentyp ein Referenzdatentyp war, z. B. Instanzen einer Klasse oder Felder. In Java gabe es keine Möglichkeit, diese Referenzen (RAM-Adressen der Speicherbereiche dieser Variablen) **direkt** zu beeinflussen. Dieses ist in **C** anders. Hier können der Inhalt einer Speicherzelle und die Adresse einer Speicherzelle getrennt angesprochen werden. Dieses macht u.a. die Beliebtheit von **C** in der Gemeinschaft der hardwarenahen Programmierer aus. Wir versuchen hier, ohne Anspruch auf Vollständigkeit, ein kleines 1 * 1 der Pointer-Programmierung in C vorzustellen.

9.1 Einfache Eigenschaften von Zeigern (Pointern)

Def.1: Zeiger (Pointer) kleinsten komplierbaren Einheiten. In Java waren die Klassen die kleinsten kompilierbaren Einheiten. Zu jedem einfachen Datentyp in C gibt es ein Pointer-Datentyp.

a) **Deklaration von Zeigern**: Die Deklaration einer Pointer-Variable hat den gleichen syntaktischen Aufbau wie die einer gewöhnlichen Variable, nur steht vor der Pointer-Variablen einStern (*):

BSP.7:
```
int *a; unsigned long *b; char *c; double *x;
```

b) **Zuweisung von Adresswerten**: Pointer-Variablen bekommen ihre Adresswerte entweder als Adresswerte von Speicherbereichen, die durch bereits existierende Variablen gewöhnlicher Datentypen (z.B.: einfache Datentypen, Felder) verwaltet werden oder von Speicherbereichen, die eigens unter der Verwaltung von Zeigern angelegt werden (darauf werden wir später noch eingehen). Die Zuweisung von Adressen von Speicherbereichen gewöhnlicher Datentypen benötigt den **Adressoperator &**, der der Variable, von der man die Adresse wissen möchte vorangestellt wird.

BSP.8:
```
int *a, b=7; a=&b;
```

c) **Zugriff auf den Inhalt** einer Speicherzelle, deren Adresse durch einen Pointer verwaltet wird: Der Zugriff auf den Inhalt wird mit dem Inhaltsoperator * vollzogen, der dem Pointer vorangestellt wird. Hierbei ist Voraussetzung, dass der Pointer tatsächlich über einen Speicherbereich verfügt (z.B. durch eine Zuweisung, wie unter b)). Hat der Pointer keinen Speicherbereich, dann stürzt das Programm ab.

BSP.9:
```
int *a, b=11, c; a=&b; c = *a; /* c hat dann den Wert 11 */
```

d) **Ausgabe** von Adresswerten: Zeigerwerte können mit printf() unter Verwendung des Formats **%p** ausgegeben werden:

BSP.10:
```
/* a ist durch die Anweisungsfolge von BSP.9 gefüllt */
       printf("Zeiger: Inhalt: %d Adresse: %p \n",*a,a);
```

BSP.11: Nachfolgend ist ein kleines Pointer-Anwendungsprogramm gezeigt, das die genannten Aktivitäten ausführt:

```
/*************************************************************/
/* Verfasser: Prof. Dr. Gregor Büchel                        */
/* Source   : Point1.c                                       */
/* Zweck    : Einfache Pointeranwendung                      */
/* Stand    : 01.06.2012                                     */
/*************************************************************/
#include <stdio.h>
/* Prototyp */
int eiNat();

int main ()
{int *a, b, c;
 printf("Pointer: \n");
 b=einNat();
 a=&b;
 c=*a;
 printf("Inhalt: %d  Adresse: a = %p  Inhalt: %d \n",c,a,*a);
}
```

9.2 Zeiger und Felder

Wenn man mit einem **Zeiger** den Adressbereich eines **Feldes x[]** verwalten möchte, ordnet man dem Pointer die Startadresse des Feldes zu: double *z; **z=&x[0];**

Man kann in C auch diese Zuordnung in verkürzter Form ausführen, da der Feldname die Startadresse des Feldes äquivalent repräsentiert: **z=x;**

Wenn das Feld **N** Komponenten hat, kann man in gleicher Form auf die Adresse jeder Komponente zugreifen: **z=&x[i];** (für alle i mit $0 \leq i \leq N-1$).

BSP.12: Nachfolgend ist ein kleines Anwendungsprogramm gezeigt, in dem mit einem Pointer auf die Komponenten eines Feldes zugegriffen wird:

```
/*************************************************************/
/* Verfasser: Prof. Dr. Gregor Büchel                        */
/* Source   : PointFeld.c                                    */
/* Zweck    : Ein Pointer für ein Feld                       */
/* Stand    : 01.06.2012                                     */
/*************************************************************/
#include <stdio.h>
/* Prototyp */
int eiNat();

int main ()
{double *z, w;
 int i,n;
 printf("Ein Pointer für ein Feld: \n");
 n=einNat();
 double x[n];
 for(i=0; i<n; i++)
 { printf("Eingabe x[%d]: \n",i);
   scanf("%lf",&x[i]);
 }
 printf("Zugriff auf die Komponenten mit einem Pointer: \n");
 for(i=0; i<n; i++)
 { z=&x[i];
   w= *z;
   printf("x[%d]: Adresse: %p Wert: %lf \n",i,z,w);
 }
}
```

Im folgenden dokumentieren wir einen Auszug aus dem Runtimeprotokoll, aus dem ersichtlich wird, dass die einzelnen Komponenten des Feldes x[] einen RAM-Platz im Umfang von 8 Byte einnehmen:

```
C:\gbC>PointFeld
Ein Pointer f³r ein Feld:
Bitte geben Sie eine natuerliche Zahl ein:
4
Eingabe x[0]:
3.5
……
Zugriff auf die Komponenten mit einem Pointer:
x[0]: Adresse: 0022FF10 Wert: 3.500000
x[1]: Adresse: 0022FF18 Wert: 2.710000
x[2]: Adresse: 0022FF20 Wert: 1.410000
x[3]: Adresse: 0022FF28 Wert: 11.500000

C:\gbC>
```

9.3 Pointerarithmetik

Man kann den Adresswert eines **Zeigers** mit **arithmetischen Operationen** erhöhen oder erniedrigen: Gegeben: Ein Zeiger **DTYP *z;** Hierbei belegt DTYP im RAM **m** Byte Speicherplatz.

- **Erhöhung: z = z + 1;** /* NeueAdresse = AlteAdresse + m */
- **Erniedrigung: z = z - 1;** /* NeueAdresse = AlteAdresse - m */

BSP.13: Zeigerarithmetik für einen double-Zeiger:

```
int main ()
{double *z, x[]={3.14, 2.71, 1.41, -2.71}, w;
 printf("Pointerarithmetik: \n");
 z=&x[2];
 printf("Startadresse : %p \n",z);
 z=z-1;
 w=*z;
 printf("x[1]: Adresse: %p Wert: %lf \n",z,w);
 z=z+2;
 w=*z;
 printf("x[3]: Adresse: %p Wert: %lf \n",z,w);
}
```

Im Runtime-Protokoll dieser main()-Funktion kann man anhand der Startadresse die durch Pointerarithmetik entstandenen neuen Adresswerte hexadezimal nachrechnen:

```
C:\gbC>a.exe
Pointerarithmetik:
Startadresse : 0022FF50
x[1]: Adresse: 0022FF48 Wert: 2.710000
x[3]: Adresse: 0022FF58 Wert: -2.710000
```

9.4 Pointer und Funktionen

In Unterkapitel 6. hatten wir gesehen, dass C-Funktionen auch mit dem Call by Reference (CbR) Prinzip aufgerufen werden können. Dort wurden als Beispiele von Referenzvariablen Felder übergeben. Nun werden wir Funktionen kennenlernen, bei denen der CbR-Aufruf mittels Pointern erfolgt. Wir betrachten dazu ein Programmsystem, das im Quelltext **VektorP.C** gegeben ist und neben der main()-Funktion aus folgenden Funktionen besteht:

a) int einNat();
b) void vektrech(int n);
c) void vektein(double *x, int n);
d) void vektausp(double *z, int l);
e) double vektbetp(double *x, int m);
f) void vektaddp(double *z, double *x, double *y, int n);

- Mit der Funktion **einNat()** wird eine natürliche Zahl **n** eingelesen.
- In der Funktion **vektrech()** werden dann double-Felder **u[]**, **v[]**, **w[]** mit einer Länge **n** angelegt.
- **vektrech()** ruft die Funktion **vektein()** mit dem bereits existierenden Feld **u[]** auf (**x** ist der Zeiger auf die Adresse **&u[0]**).
- **vektein()** erfasst mit scanf() und Mitteln der Pointerarithmetik die Komponenten des Feldes, auf das **x** zeigt.
- **vektausp()** gibt das Feld aus, auf das **z** zeigt.
- **vektbetp()** berechnet den Betrag von dem Feld, auf das **x** zeigt.

- Die Berechnung des Summenvektors wird in der Funktion **vektaddp()** ausgeführt. Der Summenvektor, auf den **z** zeigt, wird mit Mitteln der Pointerarithmetik aus den Komponenten der Felder berechnet, auf die **x** und **y** zeigen.

```
/*************************************************************/
/* Verfasser: Prof. Dr. Gregor Büchel                        */
/* Source   : VektorP.c                                      */
/* Zweck    : Anlegen und Einlesen eines Vektors mittels Poin- */
/*            tern auf Feldern, Berechnung seines Betrages,    */
/*            Berechnung der Vektorsumme                       */
/* Stand    : 03.06.2012                                     */
/*************************************************************/

#include <stdio.h>
#include <math.h>

/* Prototypen */
int einNat();
void vektrech(int n);
void vektein(double *x, int n);
void vektausp(double *z, int l);
double vektbetp(double *x, int m);
void vektaddp(double *z, double *x, double *y, int n);

int main ()
{int n;
 printf("Vektorrechnung:\n");
 printf("Eingabe der Vektorlaenge:\n");
 n=einNat();
 printf("Vektorlaenge=%d\n",n);
 vektrech(n);
 return 0;
}

int einNat()
{int k;
 do
 { printf("Bitte geben Sie eine natuerliche Zahl ein:\n");
   scanf("%d",&k);
   if (k<1)
   { printf("%d ist keine natuerliche Zahl. Neueingabe!\n",k);
   }
 } while(k<1);
 return k;
}

void vektrech(int n)
{double a;
 double u[n], v[n], w[n];
 int i;
 printf("Eingabe: Vektor x[]: \n");
 vektein(u,n);
 printf("Ausgabe: Vektor x[]: \n");
 vektausp(u,n);
 a=vektbetp(u,n);
 printf("|x|=%lf \n",a);
 printf("Berechnung der Vektorsumme: \n");
```

```
 printf("Eingabe: Vektor y[]: \n");
 vektein(v,n);
 vektaddp(w,u,v,n);
 printf("Ausgabe: Summenvektor z[]: \n");
 vektausp(w,n);
}


void vektein(double *x, int n)
{ int i;
  for(i=0; i<n; i++)
  { printf("Eingabe[%d]: \n",i);
    scanf("%lf",x);
    x=x+1;
  }
}


void vektausp(double *x, int n)
{ int i;
  double w, *s;
  for(i=0; i<n; i++)
  { w=*x;
    printf("[%d]=%lf \n",i,w);
    x=x+1;
  }
}

double vektbetp(double *x, int n)
{ double w=0., u;
  int i;
  for(i=0; i<n; i++)
  { u=*x;
    w=w+u*u;
    x=x+1;
  }
  w=sqrt(w);
  return w;
}

void vektaddp(double *z, double *x, double *y, int n)
{ double w1, w2, *p;
  int i;
  for (i=0; i<n; i++)
  { w1=*x;
    w2=*y;
    *z=w1+w2;
    x=x+1;
    y=y+1;
    z=z+1;
  }
}
```