L8. Zeiger und Funktionen

- Zeiger und Adressoperator
- Was ist eine Funktion
- Deklaration einer Funktion
- Definition einer Funktion
- Aufruf einer Funktion
- Globale und lokale Variablen
- Rekursive Funktionen

Zeiger und Adressoperator

Jede Speicherzelle in der Arbeitsspeicher trägt eine Nummer — ihre Adresse.

Ein Zeiger ist eine Variable, die die Adresse eines im Speicher gelegten Objektes aufnehmen kann. Ein Zeiger wird normal wie eine Variable definiert, dem Zeigernamen ist lediglich ein Stern vorangestellt:

```
Typname   *Zeigername;
```

(Bedeutung: "`Zeigername`" ist ein Zeiger auf "`Typname`".)

Beispiel:

```
int  *int_pointer;     // int_pointer ist ein Zeiger auf int
float  *float_pointer; // float_pointer ist ein Zeiger auf float
```

Mit dem Adressoperator & erhält man die Adresse eines Objekts:

Ist x eine Variable vom Typ "Typname", so liefert der Ausdruck `&x` einen Zeiger auf das Objekt x vom Typ "Zeiger auf Typname"

Beispiel:

```
int alpha = 1;
int *ptr = &alpha;          // Zeiger ptr erhält die Adresse
                            // der Variablen alpha
scanf("%d", &alpha);
```

Demonstrationsprogramm für Zeiger

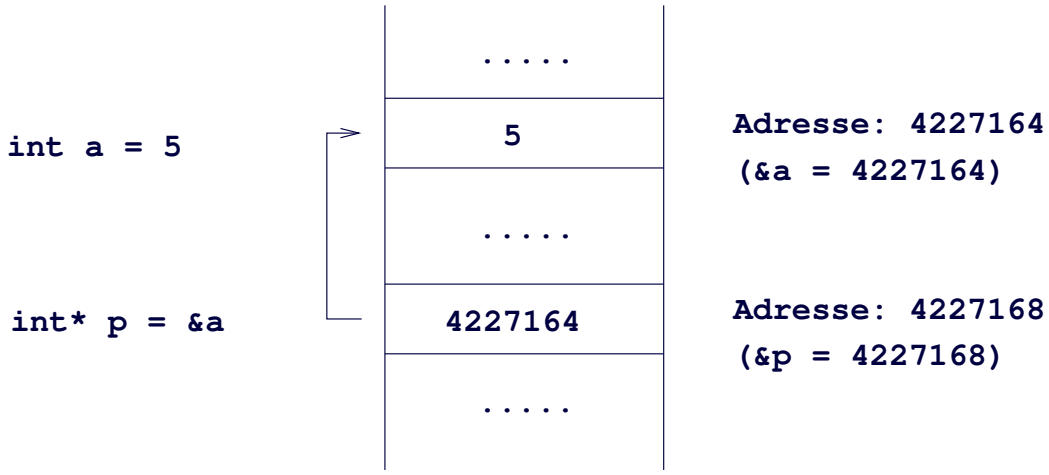
```
#include <stdio.h>

int a,  *p;     // Definition der Variablen a und p
                // oder  int*  p;

int main()      // Werte und Adressen ausgeben {
    a = 5; p = &a;
    printf("Wert von a: \t%7d,   ", a);        // \t Tabulator
    printf("  Adresse von a: \t%7u\n" , &a);
    printf("Wert von p: \t%7u,   ",  p);
    printf("  Adresse von p: \t%7u\n", &p);
    return 0;
}
```

Beispiel einer Bildschirmausgabe:

```
 Wert von a:         5;    Adresse von a:  4227164
 Wert von p:   4227164;    Adresse von p:  4227168
```

Variable	Werte	Adresse
	...	
a	5	4227164
p	4227164	4227168
	...	

Zeiger und Adresse



```
a  = 5;
&a = 4227164;
p  = 4227164;
*p = 5;
&p = 4227168;
```

Was ist eine Funktion

Funktionen sind die Grundbausteine eines C-Programms.

- Jedes C-Programm besitzt mindestens eine Funktion, die Funktion main, mit der die Programmausführung beginnt.
- Ein C-Programm kann weitere Funktionen enthalten, die Teile des Programms unter eigenem Namen zusammenzufassen.
- Eine Funktion kann mit Hilfe ihres Namens aufgerufen werden. Dabei kann man auch Parameter mitgeben und Ergebnisse zurück erhalten.
- Eine Funktion kann wieder weitere Funktionen aufrufen. Auf diese Weise wird das gesamte Program bausteinartig zusammengesetzt.
- Viele nützliche Funktionen stehen bereits in den Standardbibliothek zur Verfügung.
- Wichtige Vorteile der Aufteilung eines Programms in Funktionen:
  - ⋆ einmal schreiben, mehrfach benutzen;
  - ⋆ wesentlich übersichtlichere Struktur;
  - ⋆ lokale Variablen mit dem selben Name in verschiedener Funktionen;
  - ⋆ Funktionen können in separaten Quelldateien abgelegt werden und können getrennt übersetzt und getestet werden.
  - ⋆ Fehlerfreie Funktionen können in Bibliotheken gesammelt und weiter benutzt werden.

Bibliotheksfunktionen

Die Sprache C besitzt einen sehr kleinen Sprachkern mit nur knapp 30 reservierten Wörtern. Es gibt aber zahlreiche Bibliotheksfunktionen, die den Befehlswortschatz von C erweitern. Um solche Funktionen aufzurufen, müssen die entsprechende Header-Dateien durch `#include` eingebunden werden (ggf. mehre `#include`). z.B.:

math.h
double cos(double x) (Cosinus von x);
double exp(double x) (Potenz zur Basis e von x);
double log(double x) (Logarithmus von x mit der Basis e);
double pow(double x, double y) (x hoch y);
double sqrt(double x) (Quadratwurzel von x);

stdio.h
int printf(char *Format-String);
int scanf(char *Format-String);

stdlib.h
void srand(unsigned seed) (Initialisiert der Zufallszahlengenerator);
int rand() (Ermittelt eine Zufallszahl zwischen 0 und RAND_MAX);
int abs(int n) (Absolutwert von n);

Deklaration einer Funktion

Jede Funktion muss vor ihrem ersten Aufruf deklariert werden. Die Deklaration einer Funktion legt den Prototyp der Funktion fest:

```
Typ Funktionsname (Typ [Parameter], ..., Typ [Parameter]);
```

Beispiel:
```
void test (int param1, double param2);
void test (int, double);
```

- Der Prototyp liefert alle notwendige Informationen für einen Funktionsaufruf:
  - ⋆ der Typ der Funktion (Rückgabewertstyp).
  - ⋆ der Name der Funktion.
  - ⋆ die Anzahl und die Typen der Argumente
- Eine Deklaration endet immer mit einem Semikolon.
- Die formale Parameter mit Typ und Namen werden in (...) angegeben. Namen können weggelassen. Aber es ist sinnvoll welche einzutragen, und sinnigerweise die gleichen wie bei der Funktionsdefinition.
- Hat eine Funktion keine Parameter, muss der Typ void in den runden Klammern gegeben werden.

```
int TestFunktion();            // falsche Deklaration
int TestFunktion(void);        // korrekte Deklaration
```

Beispiel für Deklaration

```
#include <stdio.h>

void test (int param1, double param2);       //Deklaration

void main() {
    printf("Jetzt folgt der Aufruf von test().\n\n");
    test(2, 3.5);

    printf("Und jetzt wieder in main().\n");
}

------------------------------------------------------------
void test (int param1, double param2)       //Definition
{
    printf("In der Funktion test().\n");
    printf("1. Argument: %d \n",  param1);
    printf("2. Argument: %.2f \n",  param2);
}
```

Definition einer Funktion

Eine Funktion kann wie folgt definiert werden:

```
Typ Name ([Deklarationsliste]) // Funktionskopf
{                              // Beginn des Funktionsblocks
    ....
    Was die Funktion macht
    ....
}                              // Ende der Funktion
```

- Typ ist der Typ der Funktion (= der Typ vom Rückgabewert der Funktion). Eine Funktion ohne Rückgabewert hat den Typ void.
- Name ist der Funktionsname, der wie ein Variablenname gebildet wird. Sinnigerweise soll der Funktionsname auf den Zweck der Funktion hinweisen.
- Deklarationsliste enthält die Namen der Parameter und deklariert ihren Typ. Die Liste kann auch leer sein. wie z.B. main()
- Funktionsblock steht immer in {...}. Dort können neue Variablen vereinbart werden und Anweisungen formuliert werden, die festlegen, was die Funktion macht.
- Eine Funktion kann an beliebiger Stelle außerhalb jeder anderen Funktion definiert werden. Sie darf aber nur einmal definiert werden.

Rückgabewert von einer Funktion

Eine Funktion muss keinen Rückgabewert liefern. Sie hat dann den Typ void.

Hat eine Funktion einen von void verschiedenen Typ, so muss die Ausführung mit der Anweisung "return Ausdruck" enden. Der Wert von dem Ausdruck ist dann der Rückgabewert der Funktion.

```
int Summe (int Zahl1, int Zahl2)
{
   return (Zahl1 + Zahl2);
}
```

- Wie immer: wenn der Typ des Ausdrucks nicht mit dem Typ der Funktion übereinstimmt, führt der Compiler eine implizite Typenumwandlung durch, falls es möglich ist (sonst Fehlermeldung).
- Eine Funktion mit einem Rückgabewert kann man sich als Variablen vorstellen, deren Wert beim Aufruf der Funktion neu berechnet wird. Z.B.

```
Zahl3 = Summe ( 3, 5 );
```

Der Aufruf einer Funktion

Eine Funktion kann wie folgt aufgerufen werden:

```
Funktionsname ( [Aktuelle-Argument-Liste] );
```

Beispiel:
```
printf("Hello, World");
test(2, 3.5);
```

- Der Aufruf einer Funktion erfolgt durch deren Funktionsname. Dahinter setzt man in Klammern die Liste von Argumenten. Die Klammern muss man auch für eine leere Argumentenliste setzen.
- Der Reihenfolge, die Anzahl und der Typ der Argumente muss mit der bei der Definition und Deklaration der Funktion festgelegte Parameterliste übereinstimmen. Die Argumente können beliebige Ausdrücke sein.
- Beim Aufruf wird ausschließlich der "**Call-by-Value**"-Mechanismus verwendet, d.h., der Wert des Argumentes wird als Kopie übergeben. Für die Funktion sind die Argumente lokale Daten.
- Vor ihrem ersten Aufruf muss eine Funktion im Quelltext deklariert werden.

Programmablauf beim Funktionsaufruf

```
func1()
{
  long    a,     func2(int, double);
  int   x;     double    y;
   .....
  a =  func2(x+4, x*y);          // Aufruf von func2()
   .....
}

                              // Wertuebergabe:
                              //  a = x+4;  b = x*y;

long func2(int a, double b)
{
   double x;
   long  result;
   ....            //  Hier folgt die
   ....            //  Berechnung von result
   return  (result);
}
```

Beispiel für Definition und Deklaration

```
 #include <stdio.h>
 int test(int param1, double param2, int* param3);  // prototyp

 void main() {
  int zwei = 2;   int wert = 2;   int ergebnis;

  printf("vor Aufruf von test:\n  zwei=%d, wert=%d, ergebnis=%d\n\n",
                zwei, wert, ergebnis);
  ergebnis = test(zwei, 3.5, &wert);                 // Aufruf
  printf("wieder in main:\n zwei=%d, wert=%d, ergebnis=%d\n\n",
                zwei, wert, ergebnis);
}

int test(int param1, double param2, int* param3) {
  printf(" in 'test':\n param1=%d, param2=%f, param3=%p, *param3=%d\n\n",
                param1, param2, param3, *param3);
  param1 = 5;   param2 = 4.5;   *param3 = 5;
  printf(" neue Werte:\n param1=%d, param2=%f, param3=%p, *param3=%d\n\n",
                param1, param2, param3, *param3);
  return 4;
}
```

Ausgabe des Beispiels

```
vor Aufruf von test:
 zwei=2, wert=2, ergebnis=-370086

 in ‘test‘:
param1=2, param2=3.500000, param3=0x0012ff68, *param3=2

 neue Werte:
param1=5, param2=4.500000, param3=0x0012ff68, *param3=5

wieder in main:
 zwei=2, wert=5, ergebnis=4
```

Globale und lokale Variablen

Die in einem Funktionskopf deklarierten Variablen und die in einer Funktion definierten Variablen heißen lokale Variablen.

Außerhalb jeder Funktion definierte Variablen heißen globale Variablen.

Die Sichtbarkeit von Variablen:

- Lokale Variablen sind nach außen nicht sichtbar.
- Globale Variablen sind überall nach ihre Definition sichtbar.
- Wird eine lokale Variable mit demselben Namen wie eine globale Variable definiert, so ist innerhalb der Funktion nur die lokale Variable sichtbar. Die globale Variable wird durch Namensgleichheit verdeckt.

Die Lebensdauer von Variablen:

- Globale Variablen leben solange wie das Programm;
- Lokale Variablen werden beim Aufruf der Funktion neu angelegt und verschwinden wieder beim Verlassen der Funktion.

Sichtbarkeit und Lebensdauer der Variablen

```
#include <stdio.h>
int main(void)
{
...
}

int x;

void sub1(void)
{
int alpha = 4;
...
}

void sub2 (void)
{
...
}
```

```
#include <stdio.h>
int main(void)
{
...
}

int x;

void sub1(void)
{
int x = 4;
...
}

void sub2 (int y)
{
...
}
```

Ein Beispiel

```
#include <sdtio.h>
void DruckeSumme( void );
void main() {
   int Summe = 0, Zahl1 = 1, Zahl2 = 2, Zahl3 = 3;
   Summe = Zahl1 + Zahl2 + Zahl3;
   DruckeSumme();
}
void DruckeSumme() {
   printf("Die Summe ist: %d", Summe);
}
```

```
==========================================================================
```

```
#include <sdtio.h>
void DruckeSumme( int );
void main() {
   int Summe, Zahl1 = 1, Zahl2 = 2, Zahl3 = 3;
   Summe = Zahl1 + Zahl2 + Zahl3;
   DruckeSumme(Summe);
}
void DruckeSumme( Ergebnis ){
   printf("Die Summe ist: %d", Ergebnis);
}
```

Rekursive Funktionen

In der Mathematik werden viele Funktionen rekursiv definiert, d.h., bei der Definition einer Funktion f werden die Werte von f verwendet. Z.B., Die Fakultätsfunktion $f(n) := n!$ wird definiert durch:

$$\begin{cases} f(0) & := & 1 \\ f(n+1) & := & (n+1) \cdot f(n) \end{cases}$$

$$f(4) = 4 \cdot f(3) = 4 \cdot 3 \cdot f(2) = 4 \cdot 3 \cdot 2 \cdot f(1) = 4 \cdot 3 \cdot 2 \cdot 1 \cdot f(0) = 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 = 24$$

In C dürfen Funktionen auch rekursiv benutzt werden, d.h. eine Funktion darf sich direkt oder indirekt selbst aufrufen. Z.B, die Fakultätsfunktion kann durch folgender C-Funktion berechnet werden:

```
int fak(int n)
{
   if (n == 0)
       return (1);
   else
       return n * fak (n-1);
}
```

Programm für Fakultätsfunktion

```
#include <stdio.h>

int fak( int x );

int main(void) {
    int n, z;
    printf("Geben Sie eine natuerliche Zahl ein: \n");
    scanf("%d", &n);

    z = fak( n );
    printf("Die Fakultaet von %d ist %d! = %d\n", n, n, z);
    return 0;
}

int fak( int n ) {
    if (n == 0)
        return (1);
    else
        return n * fak(n-1);
}
```