[Funktionen] Übersicht Funktionen

- Verwendung
- Vereinbarung
- Wert einer Funktion
- Aufruf einer Funktion
- Parameter
- Rekursion

[Funktionen] Sinn von Funktionen

- Wiederverwendung häufig verwendeter nicht banaler Programmteile
- Wiederverwendung auch in anderen Programmen (Verwendung einer Funktion aus einer „Bibliothek“)
- Bessere Code Struktur
- Einfachere Wartung
- Auflösung von Problemen in Teilproblemen, Lösung in „Unterprogrammen“

Nachteil:
Jeder Funktionsaufruf kostet etwas Rechenzeit (vernachlässigbar)

[Funktionen] Beispiel

```
/* Funktionsdefinition */
double potenz (double basis, int exponent)
{
    double ergebnis = 1;
    for ( ; exponent > 0 ; exponent--)
    {
        ergebnis *= basis;
    }
    return ergebnis;
}
```

```
/* Ein möglicher Funktionsaufruf */
double pot = potenz(2.332,4);
```

[Funktionen] Erweitertes Beispiel

```
/* Funktionsdeklaration (Prototyp) */
double potenz (double, int exponent);

int main()
{
    /* Ein möglicher Funktionsaufruf */
    double pot = potenz(2.332,4);
    return 0;
}


/* Funktionsdefinition */
double potenz (double basis, int exponent)
{
    double ergebnis = 1;
    for ( ; exponent > 0 ; exponent--)
    {
        ergebnis *= basis;
    }
    return ergebnis;
}
```

[Funktionen] Vereinbarung: Deklaration

- Deklaration: Der Prototyp

```
typ                 name(              typ1 [name], ... , typN [name] );

(Funktionswert)     (Funktionsname)    (Parameterliste)
```

```
double potenz (double, int);                    /* eine Möglichkeit */
double potenz (double basis, int exponent);     /* ausführlicher */
```

  - Der Funktionswert entspricht dem Rückgabewert
  - Jede Funktion hat einen (eindeutigen) Namen
  - In Klammern werden die Parameter gelistet
    (auch Typen ohne Bezeichner
    oder sogar leere Parameterliste)
  - Die Deklaration wird mit einem Semikolon abgeschlossen
  - Im obigen Beispiel war die Deklaration global, d.h. für das ganze Programm gültig.

[Funktionen] Vereinbarung: Definition

- Definition
  - Entspricht ihrer Deklaration
  - Eine Funktionsdefinition kann nicht innerhalb einer anderen gemacht werden
  - Eine Funktion hat keinen, einen oder mehrere Parameter. Sie kann nur mit der exakten Anzahl Parameter aufgerufen werden.
  - Eine Funktion kann einen Typ haben, der in der Definition vorangestellt wird. Hat sie keinen Typ, wird dort *void* eingesetzt.

[Funktionen] Funktionswert

- Eine Funktion hat entweder keinen oder einen Funktionswert

```
void funktion_ohne_wert()
{
    printf(“Auch ohne Funktionswert ist eine Funktion “
            “wertvoll!\n“);
}

int funktion_mit_wert()
{
    /* Es muss ein Wert zurückgegeben werden */
    return 0;
}
```

- Der Funktionswert wird also durch die *return*-Anweisung festgelegt.
- Diese darf fehlen bei funktionswertlosen Funktionen (erreicht eine Funktion ihr Ende, springt der Programm-ablauf automatisch zurück zur rufenden Funktion)

[Funktionen] Funktionswert und *return*-Anweisung

```
/* implizites return */
void funktion1()
{
    anweisung1;
}

/* explizites return */
void funktioin2()
{
    anweisung1;
    return;
}

/* return-Anweisung mit Ausdruck */
typ funktion3()
{
    anweisung1;
    return ausdruck;
}
```

- Stimmt der Typ des Ausdrucks der Return-Anweisung nicht mit dem Typ der Funktion überein, erfolgt eine Typumwandlung.

[Funktionen] Aufruf

- Sprung an die Definitionsstelle
- Abarbeitung der Anweisungen
- Fortsetzung des Programms in der rufenden Funktion mit dem ersten Befehl nach dem Aufruf

[Funktionen] Parameter und Argumente

- *Parameter* innerhalb der Funktion wie (lokale) Variablen
- *Argumente* beim Funktionsaufruf sind den Parametern linear zugeordnet
- call-by-value vs. call-by-reference

```
double bas=2.34, int exp=4;
potenz(bas,exp);              /* call-by-value */

scanf(“%d“,&i);               /* call-by-reference */
```

  - call by value: Der Wert wird an die Funktion übergeben und dort in eine eigene Speicherzelle kopiert. Das Original bleibt davon unbeeinflusst
  - call by reference: Die Adresse der Speicherzelle, welche den Wert beinhaltet, wird übergeben. Nun kann direkt dieser Wert verändert werden.

[Funktionen] Felder als Parameter

- Felder werden als Parameter (und Argument) anders behandelt (dazu später mehr):
  Es wird nie eine Kopie der Werte der Komponenten übermittelt, sondern immer die Adresse der Anfangskomponente. (call-by-reference)

- Eine Funktion kann also als Nebeneffekt immer den Feldinhalt verändern
  (ausser dies wird ausdrücklich verboten, dazu gleich mehr)

[Funktionen] Felder als Parameter - Beispiel

```
#include <stdio.h>
#define LAENGE 20

void invert(char str[]);
int strlen(char str[]);

int main(void)
{
    char string[LAENGE+1];
    printf("String eingeben: ");
    scanf("%s",string);
    invert(string);
    printf("Invertiert: %s\n",string);

    return 0;
}
```

```
void invert(char str[])
{
    int i,j,n;
    char c;

    n = strlen(str);
    for (i=0 , j=n-1; i<j ; i++,j--)
    {
        c=str[i];
        str[i] = str[j];
        str[j]=c;
    }
}
```

```
int strlen(char str[])
{
    int i =0;
    while(str[i] != '\0')
        i++;
    return i;
}
```

[Funktionen] Felder als Parameter - *const*

- Um zu verhindern, daß ein Parameter innerhalb einer Funktion verändert wird, kann das Attribut *const* vorangestellt werden.

```
void keinesinnvollebeispielfunktion(const int i)
{
    i=4;         /* hier wird der Compiler eine
                    Fehlermeldung ausgeben */
}

int strlen(const char str[])
{
...
}
```

- nützliche Zusicherung, dass ein Fehler gemeldet wird, wenn man im Programm einen Parameter ändern würde, ohne das womöglich zu wollen.

[Funktionen] Leere Parameterliste

- Bei der Deklaration einer Funktion gibt es mehrere Möglichkeiten, keine Parameter anzugeben:

```
/* Funktionsdeklaration mit leerer Parameterliste: */
int testfunktion();
```

Damit ist vor allem gesagt, dass über die Parameter nichts bekannt ist. Hingegen...

```
int testfunktion(void);
```

... besagt, daß keine Parameter übergeben werden können.

[Funktionen] exit()

```
#include <stdlib.h>

exit( int status);

in stdlib.h definierte Konstanten:
EXIT_SUCCESS : 0
EXIT_FAILURE : 1
```

Verlassen des Programms an beliebiger Stelle möglich. Dabei wird ein Wert mit zurückgegeben, welcher meist besagt, ob das Programm erfolgreich oder mit welchem Fehler beendet wurde.

[Funktionen] Rekursion

- Rekursion = Selbstbezüglichkeit
- z.B. das rekursive Akronym GNU :

```
GNU
GNUS's not UNIX
GNUS's not UNIX's not UNIX
GNUS's not UNIX's not UNIX's not UNIX
...
```

- in mathematischen Definitionen z.B. Fakultät:

```
0! =1
n! = n(n-1)! für n>0
```

- Entscheidender Unterschied: Abbruchkriterium (fehlt im rekursiven Akronym)

[Funktionen] Rekursion Beispiel

- Lösung des Fakultätsbeispiels Iterativ (schrittweise)

```
long int fakultaet(int n)
{
    long int fak = 1;
    while ( n > 1)
        fak *= n--;
    return fak;
}
```

- oder Rekursiv

```
long int fakultaet_rek_lang(int n)
{
        if ( n == 0)
                return 1;
        else
            return ( n * fakultaet_rek_lang( a – 1 ));
}
```

```
long int fakultaet_rek_kurz(int n)
{
    return ( n > 0 ? n * fakultaet_rek_kurz(n-1) : 1 );
}
```

[Funktionen] Rekursion Beispiel

- Rekursionen sind zwar schön, aber auch aufwendig: Jeder einzelne Funktionsaufruf ist kostspielig (in Bezug auf Rechenoperationen)
- Stark vom Einzelfall abhängig, wann sich eine rekursive Lösung lohnt.
- Rekursionen sind nicht immer leichter verstehbar als die iterativen Formulierungen.

[Funktionen] Rekursion nochn Beispiel (für Experten)

```
void hanoi(int hoehe, char von, char nach, char ueber);

int main()
{
    int n;
    scanf("%d",&n);
    hanoi(n,1,3,2);
    return 0;
}

void hanoi(int hoehe, char von, char nach, char ueber)
{

    if (hoehe >1)
        hanoi(hoehe-1, von, ueber, nach);
    printf("Bewege Scheibe %d nach Feld %c!\n",hoehe,nach);
    if (hoehe>1)
        hanoi(hoehe-1,ueber,nach,von);

    return;
}
```

```
    x
   xxx
  xxxxx
--------- -------- ---------
    1        2        3
```