Funktionen

- Funktionen sind die grundlegenden Bestandteile eines c-Programms.
- Man kann ein c-Programm auch als eine Sammlung von Funktionen betrachten.
- Ein c-Programm enthält genau eine Funktion mit dem Namen main.

Funktionen

- 2 Gründe, Unterprogramme zu verwenden:
  - Unterprogrammtechniken ermöglichen die Strukturierung eines umfangreichen Programms, ein komplexes Problem wird in überschaubare Einzelaufgaben zerlegt und ist so besser lösbar (Divide et Impera – Teile und herrsche, Cäsar – Römischer Imperator).
  - Mehrfach benötigte Programmpassagen werden in Unterprogrammen nur ein mal codiert. Über Parametrisierung ist die Anpassung an die jeweiligen Erfordernisse möglich.
  - Schaffung wiederverwendbarer Softwarebausteine

Funktionen

- In vielen Programmiersprachen gibt es Prozeduren und Funktionen.
- Man fasst sie auch zusammen unter dem Begriff Unterprogramme (vielleicht etwas old fashioned) oder Routinen.
- In c gibt es formal nur Funktionen.
- Funktionen sind Unterprogramme, die einen Wert produzieren und zurückgeben.

Funktionen

- Im Zusammenhang mit Funktionen spricht man von:
  - Funktionsdeklaration
  - Funktionsdefinition
  - Funktionsaufruf

Funktionsdefinition

- Die Funktionsdefinition beschreibt eine Funktion vollständig und besteht aus zwei Teilen:
  - Funktionskopf
  - Funktionskörper
- Der Funktionskopf beschreibt dabei die wesentlichen Eigenschaften einer Funktion:
  - Name der Funktion
  - Typ des zu produzierenden Wertes (Typ des Returnwertes oder Returntyp der Funktion)
  - Typ und Namen der Parameter mit denen die Funktion ausgeführt werden soll. Die einzelnen Parameterangaben werden durch Komma voneinander getrennt. Die Liste der Parameter wird in runden Klammern angegeben. Hat die Funktion keine Parameter, bleibt die Liste leer, die runden Klammern müssen aber stehen.

Funktionsdefinition

- Der Funktionskörper wird durch einen Block beschrieben.
- Er beinhaltet die Definition der lokalen (automatischen) Variablen und die Anweisungen der Funktion und wird von geschweiften Klammen umschlossen.

```c
#include <stdio.h>

long fakult(int x)
{
  long f=1;
  int j;
  for (j=1; j<=x;j++) f*=j;
//while (x>1)f*=x--; // short version !!
  return f;
}

int main()
{
  int i;
  long f;

  for(i=0; i<10; i++)
  {
    f=fakult(i);
    printf("Fakultaet von %d: %ld\n",i,f);
  }
  return 0;
}
```

Im Beispiel gibt es zwei Funktionsdefinitionen:

Zeile 3-10 fakult

Zeile 12-23 main

Probieren Sie das Programm aus!
Verändern Sie in Zeile 17 den Wert 10 in 5-er Schritten nach oben.
Was passiert?

Erläuterungen dazu

- Der Funktionskopf von fakult (Zeile 3) benennt den Namen der Funktion, eben fakult.
- Der produzierte Wert (Returnwert) soll vom Typ long sein.
- Die Funktion benötigt einen Parameter, hier vom Typ int und dem Namen x.

Erläuterungen dazu

- Der Funktionskörper, der in geschweiften Klammern angegeben ist, enthält zunächst die lokalen Variablen. Diese sind nur innerhalb der Funktion verwendbar (deshalb lokale Variable).
- Es folgen die Anweisungen der Funktion:
  - for-Anweisung
  - return-Anweisung
- Die Anweisung return verlässt die Funktion und übergibt den Returnwert an die Aufrufstelle.

Erläuterungen dazu

- In Zeile 12 beginnt die Funktion main.
- Sie hat hier keine Parameter, es gibt auch eine Form der main-Funktion mit Parametern.
- Die runden Klammern müssen aber immer angegeben werden!
- Main produziert einen Returnwert vom Typ int, der an das aufrufende Betriebsystem übergeben wird.
- 0 heißt, dass das Programm erfolgreich beendet worden ist. Anderenfalls kann ein errorcode zurückgegeben werden (Recherche: c errorcodes).

Funktionsaufruf

- In Zeile 19 wird in der Funktion main die Funktion fakult aufgerufen.
- Damit werden die lokalen Variablen von Fakult im Speicher angelegt und das Programm zunächst mit den Anweisungen von Fakult weiter ausgeführt.
- Nach Beendigung der Funktion fakult werden die lokalen Variablen wieder abgebaut und main weiter ausgeführt.
- Im Beispiel hat fakult die Fakultät des Wertes von x berechnet. Dieser Wert – kann man sich vorstellen - steht jetzt an der Stelle des Aufrufes von fakult.

Funktionsaufruf

- Gewissermaßen ersetzt in der Anweisung

  f=fakult(i); mit i=3 das Ergebnis, der Wert 6, den Aufruf, so dass nach Ausführung von fakult(3), f=6; als nächstes ausgeführt würde.

- Man kann sich also vorstellen, dass zur Laufzeit des Programms der Aufruf einer Funktion durch das jeweilige Ergebnis, den Returnwert, temporär substituiert wird.

Beispiel mehrere Parameter

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

char buf[128];
long powInt(int base, int n)
{
    int i;
    long p=base;
    for (i=1; i < n ; i++) p*=base;
    return p;
}

int main()
{
    int b,e;
    long result;
    printf("base    : "); fgets(buf,128,stdin); b=atoi(buf);
    printf("exponent: "); fgets(buf,128,stdin); e=atoi(buf);
    result=powInt(b,e);
    printf("%d hoch %d: %ld\n",b,e,result);
    return 0;
}
```

p wird an die Aufrufstelle kopiert

b wird nach base kopiert

e wird nach n kopiert

Testen Sie das Beispiel!

Parameterübergabe

- Parameter werden in c immer kopiert.

  Im Beispiel wird e aus main nach n in powInt kopiert.

- Die Zuordnung erfolgt in der Reihenfolge von links nach rechts (nicht unbedingt die Ausführungsreihenfolge).
- Es müssen im Funktionsaufruf so viele Werte als Parameter angegeben sein, wie im Funktionskopf angegeben sind.
- Die einander zuzuordnenden Parameter müssen typverträglich sein.

PowInt Var. 2

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

char buf[128];
long powInt(int base, int n)
{
    long p=base;
    while(--n) p*=base;
    return p;
}

int main()
{
    int b,e;
    long result;
    printf("base    : "); fgets(buf,128,stdin); b=atoi(buf);
    printf("exponent: "); fgets(buf,128,stdin); e=atoi(buf);
    result=powInt(b,e);
    printf("%d hoch %d: %ld\n",b,e,result);
    return 0;
}
```

Hier wird n verändert (Zeile 8: `--n`).

Da in der Funktion mit einer Kopie des Wertes der Variablen e in main gearbeitet wird, ändert sich an e in main nichts.

return

- Return verlässt die Funktion und kehrt an die Aufrufstelle zurück.
- Return kann einen Wert an die Aufrufstelle zurückgeben. Der Wert muss vom Returntyp, der im Funktionskopf vor dem Funktionsnamen steht, sein.
- Der Rückgabewert wird an die Aufrufstelle kopiert.

Returntyp void

- Einen Sonderfall bilden Funktionen mit dem Returntyp void.
- Void bedeutet unbestimmt.
- Funktionen mit Returntyp void haben einen unbestimmten Returnwert. Es wird kein Wert über return zurückgegeben.
- Return kann bei solchen Funktionen ganz entfallen, dann endet die Funktion an der schließenden geschweiften Klammer oder über return;
- Funktionen mit Returntyp void werden wie Prozeduren in anderen Sprachen aufgerufen.

Beispiel void-Funktion

- Der Returnwert bleibt unbestimmt.
- Oben: Die Funktion wird an der schließenden Klammer verlassen.
- Unten: Die Funktion wird über die Return-Anweisung verlassen.

```
void printInt(int x)
{
    printf("%d, 0x%08x\n",x,x);
}
```

```
void printInt(int x)
{
    printf("%d, 0x%08x\n",x,x);
    return;
}
```

Die Funktionsdeklaration

- Die Funktionsdeklaration besteht im Prinzip aus dem Funktionskopf und einem Semikolon.
- Im Prinzip?? - in der Parameterliste kann die Angabe der Parameternamen entfallen, man kann auch lediglich die Parametertypen durch Komma getrennt angeben. Die Angabe der Namen fördert aber die Lesbarkeit des Programms, deshalb sollte man sie angeben.

Die Funktionsdeklaration

- Funktionsdeklarationen sind immer dann notwendig, wenn Funktionen verwendet werden sollen, die an der betreffenden Stelle noch gar nicht definiert worden oder anderweitig bekannt gemacht sind.
- Man spricht auch von einem Funktionsprototyp
- Headerfiles (das sind die Dateien mit der Dateiendung .h) enthalten hauptsächlich Funktionsprotypen. Man braucht sie, wenn ein c-Programm aus mehreren Quelldateien besteht.

```c
#include <stdio.h>

long fakult(int i);

int main()
{
  int i;
  long f;

  for(i=0; i<10; i++)
  {
    f=fakult(i);
    printf("Fakultaet von %d: %ld\n",i,f);
  }
  return 0;
}

long fakult(int i)
{
  long f=1;
  int j;
  for (j=1; j<=i;j++) f*=j;
//while (i>1)f*=i--; // short version !!
  return f;
}
```

Zeile 1 enthält eine include-Anweisung, die ein Bibliotheksheaderfile importiert.
Zeile 3 enthält die Funktionsdeklaration zu fakult.
In Zeile 12 wird fakult verwendet.
Ab Zeile 18 haben wir die Funktionsdefinition von fakult.

Probieren Sie das Programm aus!
Kommentieren Sie Zeile 3 aus:
// long fakult(int i);
Was passiert?

Programme aus mehreren Quelldateien

- Große Programme bestehen oft aus vielen c-Quelldateien.
- Dabei wird jede Quelldatei für sich allein compiliert.
- Headerfiles, die Funktionsprototypen beinhalten, schaffen für den Compiler die Verbindung zwischen den c-Modulen.
- Am Beispiel fakult soll das demonstriert werden.

FakultaetFunk.h

```
1  long fakult(int i);
2
```

FakultaetMain.c

```
1  #include <stdio.h>
2
3  #include "FakultaetFunk.h"
4
5  // long fakult(int i);
6
7  int main()
8  {
9    int i;
10   long f;
11
12   for(i=0; i<10; i++)
13   {
14     f=fakult(i);
15     printf("Fakultaet von %d: %ld\n",i,f);
16   }
17   return 0;
18 }
```

FakultaetFunk.c

```
1  #include <stdio.h>
2  // Jeder Modul muss sein eigenes headerfile beinhalten
3  #include "FakultaetFunk.h"
4
5  long fakult(int i)
6  {
7    long f=1;
8    int j;
9    while (i>1)f*=i--; // short version !!
10   return f;
11 }
```

- Compiliert werden die .c-Files, die .h-Files werden über include in die c-Files eingefügt.
- Jedes c-File soll sein eigens Headerfile per include einschließen, damit bei Änderungen an der Definition oder Deklaration von Funktionen die Konsistenz gesichert wird.

Programmbuild

- Die Anwendung wird am einfachsten gebaut (buildcommand) mit:

  gcc -o fakult FakultaetFunk.c FakultaetMain.c

- Man kann auch die build-Schritte einzeln ausführen:
  - gcc -c FakultaetFunk.c
  - gcc -c FakultaetMain.c
  - gcc FakultaetFunk.o FakultaetMain.o
- gcc -c erzeugt die Objektfiles (.o)

gcc -c :
Es wird nur compiliert,
Nicht aber gelinkt

Hier wird der Linker (Programmverbinder) ausgeführt, der Die Module und die Standardbibliothek zu einemProgramm verbindet.

Erfassen Sie die drei Dateien und
Probieren Sie es aus!

Gültigkeit und Lebensdauer

- In den vergangenen Beispielen war zu sehen, dass Variablen innerhalb von Funktionen oder aber auch außerhalb von Funktionen definiert werden.
- Der Ort einer Variablendefinition bestimmt die Eigenschaften der
  - Gültigkeit (der Teil eine Programms, in dem ein Bezeichner (hier der Variablen) bekannt ist.
  - Lebensdauer bezieht sich auf die Laufzeit, sie bezeichnet, den Zeitraum zwischen der Erzeugung einer Variablen und dem Ende ihrer Existenz.

Gültigkeit und Lebensdauer

- Variable, die innerhalb von Funktionen vereinbart werden, lokale Variable, werden beim Betreten der Funktion, wenn die Funktion aufgerufen wird, auf dem Stack angelegt. Wird die Funktion verlassen, werden diese Variablen wieder abgebaut. Sie existieren dann nicht mehr, ihre Lebenszeit ist dann beendet. Der Bezeichner lokaler Variablen ist ab der Variablendefinition bis zum Ende des umgebenden Blockes gültig.

Gültigkeit und Lebensdauer

```
1  #include <stdio.h>
2  // Jeder Modul muss sein eigenes headerfile
3  #include "FakultaetFunk.h"
4
5  long fakult(int i)
6  {
7    long f=1;
8    int j;
9    while (i>1)f*=i--; // short version !!
10   return f;
11 }
```

Im Beispiel der Funktion fakult sind die Bezeichner der Variablen f und j innerhalb der Fuktion ab ihrer Definition gültig. Sie werden angelegt, wenn die Funktion betreten wird und wieder automatisch abgebaut beim Verlassen der Funktion. Sie gehören der Speicherklasse auto (automatische Variable) an. Der Parameter i wird auch wie eine automatische, lokale Variable behandelt. Sie wird beim Funktionsaufruf durch die übergebenen Parameter initialisiert.

Gültigkeit und Lebensdauer

- Lokale Variable können mit dem Attribut static versehen sein.
  - **`static char[16+1]=“1234567812345678“;`**
- Eine lokale statische Variable existiert weiter, wenn die umgebende Funktion verlassen wird und behält ihren letzten Wert.
- Sie wird nur einmalig beim ersten Aufruf der umgebenden Funktion initialisiert.
- Man kann einen Pointer auf eine lokale statische Variable zurückgeben, auf automatische, lokale Variable niemals!

Gültigkeit und Lebensdauer

- Globale Variable sind außerhalb aller Funktionen vereinbart.
- Sie existieren über die gesamte Laufzeit des Programms.
- Sie sind ab ihrer Vereinbarung im gesamten weiteren Quelltext gültig (sichtbar).
- Globale Variable sollen sparsam verwendet werden.
- In mehreren Beispielen haben wir den Eingabepuffer für Tastatureingaben (char buf [128]) als globale Variable definiert.
- Globale Variablen werden im Unterschied zu lokalen Variablen, wenn sie nicht explizit initialisiert werden, mit 0 intialisiert.

Gültigkeit und Lebensdauer

- Gibt es Bezeichner in unterschiedlichen Ebenen
  - Globale Variable, lokale Variable
  - Variable in einem äußeren/inneren Block einer Funktion

  kommt es zur Überdeckung.
- Der Bezeichner der inneren Vereinbarung überdeckt die der äußeren Vereinbarung.

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

int x=99;

int main(int argc, char**argv)
{
    {
        int x=12;
        printf("local  x: %d\n",x);
    }
    printf("global x: %d\n",x);
    return 0;
}
```

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

int main(int argc, char**argv)
{
    int x=99;
    {
        int x=12;
        printf("inner x: %d\n",x);
    }
    printf("outer x: %d\n",x);
    return 0;
}
```

Gültigkeit und Lebensdauer

- Besteht ein Programm aus mehreren Quelltexten, so kann der Gültigkeitsbereich einer Globalen Variable auf andere c-Module ausgedehnt werden.
- Dann muss zusätzlich zur Variablendefinition im c-File eine Externvereinbarung üblicherweise im Headerfile angegeben werden.
- Für eine int-Variable x lautet die Externvereinbarung:

```
extern int i; // ohne Initialisierung!
```

Gültigkeit und Lebensdauer

- Soll eine externe, globale Variable gundsätzlich nur in ihrem Modul bekannt sein, so wird sie durch static gekennzeichnet.
- Statische, globale Variable werden nicht an den Linker bekanntgegeben.
- Statische, globale Variable sind nur im eigenen c-Quelltext sichtbar (information hiding).

Ort der Definition	Gültigkeit	Lebensdauer	Initialisierung
Außerhalb von Funktionen	Im gesamten C-Quelltext und darüber hinaus	Existieren bis zum Programmende	Einmalige Initialisierung beim Programmstart, Defaultwert: 0
static, außerhalb von Funktionen	Im gesamten C-Quelltext	Existieren bis zum Programmende	Einmalige Initialisierung beim Programmstart, Defaultwert: 0
Innerhalb einer Funktion/eines Blocks	Von der Variablendefinition bis zum Blockende	Vom Betreten des umgebenden Blocks bis zum seinem Verlassen	Bei jedem Betreten des umgebenden Blocks
static, innerhalb einer Funktion/eines Blocks	Von der Variablendefinition bis zum Blockende	Vom ersten Betreten des umgebenden Blocks bis zum Programmende	Einmalige Initialisierung beim erstmaligen Betreten des umgebenden Blocks

Speicherklassen

- auto: Variablen werden automatisch auf dem Stack auf- und abgebaut, auto ist die Standardannahme.
- static: Variablen existieren über die gesamte Laufzeit.
- extern: Variable ist in einem anderen Modul global definiert, sie wird hier nur bekannt gemacht.
- register: Die Variable soll besonders schnell sein, es wird versucht, sie in einem Prozessorregister zu halten.
- volatile: Die Variable wird von parallel (quaisiparallel) ablaufenden Programmteilen (z.Bsp. Interruptroutine) benutzt. Sie wird nicht (auch nicht temporär) in Registern gehalten.
- Die Speicherklasse wird, wenn sie angegeben werden soll, vor dem Typ der Variablen angegeben.

```
int main()
{
  register int i;
  auto long f;
```

Arrays als Parameter

- **Arrays können nicht als Parameter an Funktionen übergeben werden.**
- Soll ein Array an eine Funktion übergeben werden, so wird grundsätzlich nur die Anfangsadresse übergeben.
- Der Typ des Arrays ist natürlich bekannt.
- Die Anzahl der Arrayelemente bleibt der Funktion dabei unbekannt.

Arrays als Parameter

- Deshalb ist die Länge des Arrays grundsätzlich gesondert zu übergeben (Strings bilden hier eine Ausnahme, da die Länge durch die terminierende 0 bestimmt ist).
- Es ist zu beachten, dass Arrays bei der Übermittlung an Funktionen nicht kopiert werden.
- Alle Änderungen an einem Array in einer Funktion vollziehen sich an den originalen Daten.

```c
#include <stdio.h>

char buf[128];

void sort(int* parray, int n)
{
  int i,j;
  for(i=0; i<n-1; i++)
    for(j=i+1; j<n; j++)
      if (parray[i]>parray[j])
      {
        parray[i]^=parray[j];
        parray[j]^=parray[i];
        parray[i]^=parray[j];
      }
}
```

```c
int main()
{
  int array[]={9,5,1,2,7,3};
  int n=sizeof array/sizeof(int);
  int i;

  for(i=0; i<n; i++)
    printf("array[%d]:%d\n",
                        i,array[i]);
  puts("==========");
  sort(array,n);
  for(i=0; i<n; i++)
    printf("array[%d]:%d\n",
                        i,array[i]);
  return 0;
}
```

Die Änderungen vollziehen sich im originalen Array array, dessen Adresse an die Funktion Sort übergeben wird. Die Anzahl der Arrayelemente wird separat übergeben.

Arrays als Returnwerte

- Ähnliches gilt für Returnwerte.
- Auch hier wird nur die Adresse des zurückzugebenden Arrays übermittelt.
- Hierbei ist auf die Lebensdauer des Arrays, dessen Adresse zurückgegeben wird, zu achten.

Parameter von main

- Die main-Funktion kann auch Parameter haben.
- Es sind die Kommandozeilenparameter, sie erlauben Argumente beim Start des Programms an das Programm zu übergeben.
- Der erste Parameter ist vom Typ int und beinhaltet die Anzahl der Kommandozeilenparameter.
- Der zweite Parameter ist ein Pointer auf ein Array von Zeichenketten.

Kommandozeilenparameter

```c
#include <stdio.h>

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

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

```
$ ./a.out hans otto 73
argv[0]: ./a.out
argv[1]: hans
argv[2]: otto
argv[3]: 73
```

Das Argument mit dem Index 0 ist immer der Programmaufruf selbst, dann folgen die Argumente

Berechnung der Fakultät mit Kommandozeilenparameter

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

long fakult(int x)
{
  long f=1;
  int j;
  for (j=1; j<=x;j++) f*=j;
//while (x>1)f*=x--; // short version !!
  return f;
}

int main(int argc, char* argv[])
{
  long f;
  int  x;
  if (argc != 2)
    {printf("usage: %s <numeral>\n",argv[0]); exit (-1);}
  x=atoi(argv[1]);
  f=fakult(x);
  printf("Fakultaet von %d: %ld\n",x,f);
  return 0;
}
```

Kommando-zeilenarg.

```
$ ./a.out 3
Fakultaet von 3: 6
$
```

Test, ob Argumente Angegeben worden sind

Fehelermeldung und Programmabbruch.

Kommandozeilenparameter

- Das erste Kommandozeilenargument ist immer der Programmname selbst.
- Sollen Kommandozeilenargumente verwendet werden, so ist als erstes immer zu prüfen, ob diese beim Aufruf an angegeben worden sind.
- Ist das nicht der Fall, sollte eine Fehler- oder Usage-Meldung ausgegeben werden (Zeile 17,18).