Vorlesung 2 (4.12.2014)

2.4 Unterprogramme, Funktionen

Es gibt vordefinierte Funktionen, z.B. mathematische Funktionen. Man benötigt header-file, z.B. : `math.h` (seltsame Resultat falls nicht verwendet)

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

int main()
{
  double argument;

  argument = 0.0;
  while (argument < 2*M_PI)
  {
    printf("sin(%f)=%f\n", argument, sin(argument));
    argument += 0.1;
  }
  return(0);
}
```

Linke Bibliothek beim Compilieren: `-l<lib-name>`

```
cc -o first first.c -Wall -lm
```

Zufallszahlen

```
#include <stdlib.h>

...

value = drand48();
```

erzeugt (pseudo) Zufallszahl in $[0, 1)$
Eigene Funktionen definieren:

```
#include <stdio.h>

/*************** calc_average()**************/
```

```
/** Calculates average of values passed      **/
/** uses periodic boundary conditions        **/
/** PARAMETERS: (*)= return-paramter         **/
/**       number: .. of values               **/
/**     value: array of values               **/
/** RETURNS:                                 **/
/**    average                               **/
/*********************************************/
double calc_average(int number, double value[])
{
  int counter;  /* local variable, not visible e.g. in main() */
  double sum;   /* the same */
  sum = 0.0;
  for(counter=0; counter<number; counter++)
    sum += value[counter];
  return(sum/number);
}

int main()
{
  int counter;
  double value[10];
  double average;
  for(counter=0; counter<10; counter++)
    value[counter]=counter*counter;
  average = calc_average(10, value);
  printf("avg=%f\n", average);
  return(0);
}
```

Unterprogramme = Funktionen ohne Rückgabewert:

```
void print_value_and_flowers(int x)
{
  printf("flowers\n, value=%d, flowers\n", x);
}
```

Mehr als ein Rückgabewert: benutze Strukturen oder Zeiger (siehe später)

2.5 Geltungsbereich von Variablen

Variablen sind lokal

```
#include <stdio.h>

void change(int z)
{
  int x;
  x= 100;
  printf("local x=%d, z=%d\n", x, z);
  z= 101;
  printf("local x=%d, z=%d\n", x, z);
}

int main()
{
  int x, z;

  x = 200; z =201;     /* one can have several commands in a line */
  printf("x=%d, z=%d\n", x, z);
  change(z);
  printf("x=%d, z=%d\n", x, z);

  return(0);
}
```

[Selbsttest]

Frage: Was wird ausgegeben (1-2 min nachdenken)

2.6 Strings

Strings = Array von `char`. Ende eines String: Zahl 0.

```
char name[100];            /* up to 99 characters */
```

Stringfunktionen in `string.h` deklariert.
Bsp.:

```
  strcpy(name, "Robert Smith");               /* copies text into string */
  printf("length(%s)=%d\n", name, strlen(name));         /* prints length */
```

Nützlich: `sprintf(string, <format string>, ....)` (definiert in `stdlib.h`)
geht wie `printf` aber schreibt in String anstatt auf Standardausgabe.

Hinweis: `valgrind` findet Speicherfehler, wie z.B. Schreiben über das Ende des reservierten Bereichs bei einem String/Array.

2.7 Strukturen, selbst-definierte Datentypen

Gruppiere mehrere Elemente in einen Datentyp:
Bsp.:

```
struct particle
{
  double        mass;         /* in kg                         */
  int         charge;         /* in units of e                 */
  double position[3];         /* position in space. in meters */
}
```

Variablen-Deklaration:

```
struct particle particle1;
```

Zugriff

```
particle1.mass = 9.109e-31;
particle1.charge = 1;
particle1.position[0] = -2.3e-3;
```

Bequem: eigene Datentypen. Schreibe: `typedef` gefolgt von einer "normal" Deklaration, z.B.:

```
typedef double vector_t[3];                   /* new type 'vector_t' */
typedef struct particle particle_t;         /* new type 'particle_t' */
...

vector_t velocity;                 /* velocity is of type 'vector_t' */
particle_t electron;  /* variable 'electron' is of type 'particle_t' */
```

Konvention: sammle alle Typen in extra Header (.h) File.
Damit kann man leicht Funktionen implementieren, die mehrere Daten zurückgeben, z.B.

```
particle_t initialise_atom()
{
  particle_t atom;
  atom.mass = 1;
  ...
  return(atom);
}
```

2.8 Pointer

Pointer (Zeiger) = Adresse einer Variable im Speicher.

Deklaration: `<type> *ptr` erzeugt `ptr` als Adresse einer Variablen des Typs `<type>`.

&-Operator gibt Adresse der Variablen: `& <variable>`.

`*ptr` = Inhalt der Variablen auf die `ptr` zeigt, d.h. man kann den Inhalt setzen durch `*prt= <expression>`. Bsp:

```
int number, *address;

number = 50;
address = & number;
*address = 100;
printf("%d\n", number);
```

ergibt: 100.
Arrays = Pointer, `int value[10]` ⇒ `value`= Adresse des Anfangs des Arrays, d.h. von `value[0]`. Beides `int value[0]` und `int *value2` = Pointer auf `int`, aber für `value` wird Array der Länge 10 reserviert und `value` zeigt auf den Array Anfang. `value2` erhält anfangs KEINEN Wert.

Zugriff: `value[5]` ist äquivalent zu `*(value+5)`.

Fall: Zeiger auf Struktur zeigt: Zugriff auf Elemente durch`->` Operator.

```
struct particle *atom;
...
atom->mass = 2.0;
```

(äquivalent `(*atom).mass=2.0`)
Pointer: verwendbar um komplexe Beziehungen zwischen Variablen herzustellen, z.B. um komplexe Datentypen zu bauen (Listen oder Bäume, s.u.).

Pointer: verwendbar um aus Funktion Daten ohne `return` zurückzugeben. (Nützlich, falls viele Variablen zurückgegeben werden)

```
void add_numbers(int n1, int n2, int *result_p)
{
  *result_p = n1+n2;
}
```

Hinweis: Pointer `result_p` kann in `add_numbers()`, nur den Inhalt des Speichers ändern, auf den `result_p` zeigt, nicht aber den (externen) Wert von `result_p`.

TAKE HOME WORK

Besorgen Sie sich (falls noch nicht vorhanden) das des C Tutorials vom StudIP, und lesen sie die restlichen Seiten. Vollziehen sie die Programmteile praktisch nach!