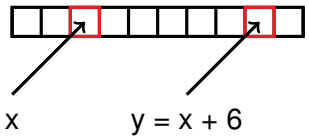
Funktionen

```
#include <math.h>
void init(float a)
{
  if (a <= 0) return;
  printf("%f\n", log(a));
}
float max(float a, float b)
{
  return (a < b) ? b : a;
}
```

- Funktionen werden definiert mit
  `rettyp funktion(typ1 arg1, typ2 arg2,...) {...}`
- Ist der Rückgabetyp **`void`**, gibt die Funktion nichts zurück
- **`return`** verlässt eine Funktion vorzeitig (bei Rückgabetyp **`void`**)
- **`return`** `wert` liefert zusätzlich `wert` zurück

Datentypen – Zeiger



- Zeigervariablen (Pointer) zeigen auf Speicherstellen
- Ihr Datentyp bestimmt, als was der Speicher interpretiert wird (**`void`** $*$ ist unspezifiziert)
- Es gibt keine Kontrolle, ob die Speicherstelle gültig ist (existiert, les-, oder schreibbar)
- Pointer verhalten sich wie Arrays,
  bzw. Arrays wie Pointer auf ihr erstes Element

Datentypen – Zeiger

```
char x[] = "Hallo Welt";
x[5] = 0;
char *y = x + 6, *noch_ein_pointer, kein_pointer;
y[2] = 0;
printf("%s-%s\n", y, x); → We-Hallo
```

- Zeiger werden mit einem führendem Stern deklariert
- Bei Mehrfachdeklarationen: genau die Variablen mit führendem Stern sind Pointer
- `+,-,+=,-=, ++,--` funktionieren wie bei Integern
- `p += n` z.B. versetzt p um n Elemente
- Pointer werden immer um ganze Elemente versetzt
- Datentyp bestimmt, um wieviel sich die Speicheradresse ändert

Zeiger – Referenzieren und Dereferenzieren

```
float *x;
float array[3] = {1, 2, 3};
x = array + 1;
printf("*x = %f\n", *x); // →*x = 2.000000
float wert = 42;
x = &wert;
printf("*x = %f\n", *x); // →*x = 42.000000
printf("*x = %f\n", *(x + 1)); // undefinierter Zugriff
```

- `*p` gibt den Wert an der Speicherstelle, auf die Pointer `p` zeigt
- `*p` ist äquivalent zu `p[0]`
- `*(p + n)` ist äquivalent zu `p[n]`
- `&v` gibt einen Zeiger auf die Variable `v`

Größe von Datentypen – sizeof

```
int x[10], *px = x;
printf("int: %ld int[10]: %ld int *: %ld\n",
       sizeof(int), sizeof(x), sizeof(px));
// → int: 4 int[10]: 40 int *: 8
```

- `sizeof(datentyp)` gibt an, wieviel Speicher eine Variable vom Typ `datentyp` belegt
- `sizeof(variable)` gibt an, wieviel Speicher die Variable `variable` belegt
- Bei Arrays: Länge multipliziert mit der Elementgröße
- Zeiger haben immer dieselbe Größe (8 Byte auf 64-Bit-Systemen)
- Achtung: Das Ergebnis ist 64-bittig (Typ **`long unsigned int`**), daher bei Ausgabe „%ld“ verwenden

Unterschied zwischen Variablen und Zeigern

```
int a, b;
int *ptr1 = &a, *ptr2 = &a;

b = 2; a = b; b = 4;
printf("a=%d b=%d\n", a, b); // →a=2 b=4

*ptr1 = 5; *ptr2 = 3;
printf("*ptr1=%d *ptr2=%d\n", *ptr1, *ptr2);
// →ptr1=3 ptr2=3
```

- Zuweisungen von Variablen in C sind tief, Inhalte werden kopiert
- Entspricht einfachen Datentypen in Python (etwa Zahlen)
- Mit Pointern lassen sich flache Kopien erzeugen, in dem diese auf denselben Speicher zeigen
- Entspricht komplexen Datentypen in Python (etwa Listen)

Funktionen – main

```
int main(int argc, char **argv)
{
  printf("der Programmname ist %s\n", argv[0]);
  for(int i = 1; i < argc; ++i) {
    printf("Argument %d ist %s\n", i, argv[i]);
  }
  return 0;
}
```

- `main` ist die Hauptroutine
- erhält als **`int`** die Anzahl der Argumente
- und als **`char`** `**` die Argumente
- Zeiger auf Zeiger auf Zeichen ≙ Array von Strings
- Rückgabewert geht an die Shell

Schleifen – while und do ... while

```
for(int i = 0; i < 10; ++i) { summe += i; }
```

```
int i = 0;
while (i < 10) { summe += i; ++i; }
```

```
int i = 0;
do { summe += i; ++i; } while (i < 10);
```

- **`while`** `(cond) block` und **`do`** `block` **`while`** `(cond);` führen `block` aus, solange die Bedingung `cond` wahr ist
- Unterschied zwischen den beiden Formen:
  - **`while`** überprüft die Bedingung `cond` *vor* Ausführung von `block`
  - **`do ... while`** erst danach
- Die drei Beispiele sind äquivalent
- Jede Schleife kann äquivalent als **`for`**-, **`while`**- oder **`do...while`**-Schleife geschrieben werden

Bedingte Ausführung – switch

```
char buch = argv[1][0];
switch (buch) {
case 'a':
  printf("a, rutscht durch zu b\n");
case 'b':
  printf("b, hier geht es nicht weiter\n");
  break;
default:
  printf("Buchstabe '%c' ist unerwartet \n", buch);
}
```

- Das Argument von **`switch`** `(wert)` muss ganzzahlig sein
- Die Ausführung geht bei **`case`** `konst:` weiter, wenn `wert=konst`
- **`default`**: wird angesprungen, wenn kein Wert passt
- Der **`switch`**-Block wird ganz abgearbeitet
- Kann explizit durch **`break`** verlassen werden

struct – Datenverbünde

```
struct Position {
  float x, y, z;
};
struct Particle {
  struct Position position;
  float ladung;
  int identitaet;
};
```

- **`struct`** definiert einen Verbund
- Ein Verbund fasst mehrere Variablen zusammen
- Die Größe von Feldern in Verbünden muss konstant sein
- Ein Verbund kann Verbünde enthalten

Variablen mit einem struct-Typ

```
struct Particle part, *ptr = &part;

part.identitaet = 42;
ptr->ladung = 0;

struct Particle part1 = { {1, 0, 0}, 0, 42};
struct Particle part2 = { .position = { 1, 0, 0 },
                          .ladung = 0,
                          .identitaet = 43};
struct Particle part3 = { .identitaet = 44};
```

- Elemente des Verbunds werden durch „.“ angesprochen
- Kurzschreibweise für Zeiger: `(*pointer).x = pointer->x`
- Verbünde können wie Arrays initialisiert werden
- Initialisieren einzelner Elemente mit Punktnotation

typedef

```
typedef float real;
typedef struct Particle Particle;
typedef struct { real v[3]; } Vektor3D

struct Particle part;
Particle part1;       // beides ist ok, selber Typ

Vektor3D vektor; // auch ok
struct Vektor3D vektor; // nicht ok, struct Vektor3D fehlt
```

- **typedef** definiert neue Namen für Datentypen
- **typedef** ist nützlich, um Datentypen auszutauschen, z.B. double anstatt float
- Achtung: **struct** `Particle` und `Particle` können auch verschiedene Typen bezeichnen!
- **typedef struct** `{...} typ` erzeugt keinen Typ **struct** `typ`

#define – Makros

```
#define PI 3.14
/* Position eines Teilchens
   x sollte Zeiger auf Particle sein */
#define POSITION(x) ((x)->position)
POSITION(part).z = PI;
#undef PI
float test = PI; // Fehler, PI undefiniert
```

- **#define** definiert Makros
- **#undef** entfernt Definition wieder
- Ist Präprozessorbefehl, d.h. Makros werden *textuell* ersetzt
- Ohne Parameter wie Konstanten einsetzbar
- Makros mit Parametern nur sparsam einsetzen!
- Klammern vermeiden unerwartete Ergebnisse
- **#ifdef** testet, ob eine Makro definiert ist

const – unveränderbare Variablen

```
static const float pi = 3.14;

pi = 5; // Fehler, pi ist nicht schreibbar

// Funktion aendert nur, worauf ziel zeigt, nicht quelle
void strcpy(char *ziel, const char *quelle);
```

- Datentypen mit **const** sind konstant
- Variablen mit solchen Typen können nicht geändert werden
- Statt Makros besser Variablen mit konstantem Typ
- Diese können nicht verändert werden
- Anders als Makros haben sie einen Typ
- Vermeidet seltsame Fehler, etwa wenn einem Zeiger ein **float**-Wert zugewiesen werden soll
- **static** ist nur bei Verwendung mehrerer Quelldateien wichtig

Bibliotheksfunktionen

- In C sind viele Funktionen in Bibliotheken realisiert
- Diese sind selber in C / Assembler geschrieben
- Basisfunktionen sind Teil der C-Standardbibliothek
- Andere Bibliotheken müssen mit -l geladen werden, z.B.
  `gcc -Wall -O3 -std=c99 -o mathe mathe.c -lm`
  zum Laden der Mathematik-Bibliothek „libm“
- Um die Funktionen benutzen zu können, sind außerdem Headerdateien notwendig

Speicherverwaltung – malloc und free

```
#include <stdlib.h>
// Array mit Platz fuer 10000 integers
int *vek = (int *)malloc(10000*sizeof(int));
for(int i = 0; i < 10000; ++i) vek[i] = 0;
// Platz verdoppeln
vek = (int *)realloc(20000*sizeof(int));
for(int i = anzahl; i < 20000; ++i) vek[i] = 0;
free(vek);
```

- Speicherverwaltung für variabel große Bereiche im *Freispeicher*
- `malloc` reserviert Speicher
- `realloc` verändert die Größe eines reservierten Bereichs
- `free` gibt einen Bereich wieder frei

Speicherverwaltung – malloc und free

```
#include <stdlib.h>
// Array mit Platz fuer 10000 integers
int *vek = (int *)malloc(10000*sizeof(int));
for(int i = 0; i < 10000; ++i) vek[i] = 0;
// Platz verdoppeln
vek = (int *)realloc(20000*sizeof(int));
for(int i = anzahl; i < 20000; ++i) vek[i] = 0;
free(vek);
```

- Wird dauernd Speicher belegt und nicht freigegeben, geht irgendwann der Speicher aus („Speicherleck“)
- Dies kann z.B. so passieren:

```
int *vek = (int *)malloc(100*sizeof(int));
vek = (int *)malloc(200*sizeof(int));
```

- Ein Bereich darf auch nur einmal freigegeben werden

math.h – mathematische Funktionen

```
#include <math.h>

float pi = 2*asin(1);

for (float x = 0; x < 2*pi; x += 0.01) {
  printf("%f %f\n", x, pow(sin(x), 3)); // x, sin(x)^3
}
```

- `math.h` stellt mathematische Standardoperationen zur Verfügung
- Bibliothek einbinden mit
  `gcc -Wall -O3 -std=c99 -o mathe mathe.c -lm`
- Beispiel erstellt Tabelle mit Werten x und $sin(x)^3$

string.h – Stringfunktionen


```
#include <string.h>

char test[1024];
strcpy(test, "Hallo");
strcat(test, " Welt!");
// jetzt ist test = "Hallo Welt!"
if (strcmp(test, argv[1]) == 0)
  printf("%s = %s (%d Zeichen)\n", test, argv[1],
         strlen(test));
```

- `strlen(quelle)`: Länge eines 0-terminierten Strings
- `strcat(ziel, quelle)`: kopiert Zeichenketten
- `strcpy(ziel, quelle)`: kopiert eine Zeichenkette
- `strcmp(quelle1, quelle2)`: vergleicht zwei Zeichenketten, Ergebnis ist < 0, wenn lexikalisch `quelle1` $<$ `quelle2`, $= 0$, wenn gleich, und > 0 wenn `quelle1` $>$ `quelle2`

Institute for Computational Physics
University of Stuttgart Germany

string.h – Stringfunktionen

http://www.icp.uni-stuttgart.de

```
#include <string.h>

char test[1024];
strncpy(test, "Hallo", 1024);
strncat(test, " Welt!", 1023 - strlen(test));
if (strncmp(test, argv[1],2) == 0)
  printf("die 1. 2 Zeichen von %s und %s sind gleich\n",
    test, argv[1]);
```

- Zu allen `str`-Funktionen gibt es auch `strn`-Versionen
- Die `strn`-Versionen überprüfen auf Überläufe
- Die Größe des Zielbereichs muss *korrekt* angegeben werden
- Die terminierende 0 benötigt ein Extra-Zeichen!
- Die `str`-Versionen sind oft potentielle Sicherheitslücken!

A. Arnold Computergrundlagen 29/33

string.h – memcpy und memset

```c
#include <string.h>

float test[1024];
memset(test, 0, 1024*sizeof(float));
// erste Haelfte in die zweite kopieren
memcpy(test, test + 512, 512*sizeof(float));
```

- Lowlevel-Funktionen zum Setzen und Kopieren von Speicherbereichen
- Z.B. zum initalieseren oder kopieren von Arrays
- `memset(ziel, wert, groesse)`: füllt Speicher byteweise mit dem *Byte* `wert`
- `memcpy(ziel, quelle, groesse)`: kopiert `groesse` viele *Bytes* von `quelle` nach `ziel`

stdio.h – printf und scanf

```c
#include <stdio.h>
char  kette[11];
float fliess;
int   ganz;
if (scanf("%10s %f %d", kette, &fliess, &ganz) == 3) {
  printf("%s %f %d\n", kette, fliess, ganz);
}
```

- `printf` gibt formatierten Text auf Standardausgabe aus
- `printf` liest Werte von der Standardeingabe
- Beide benutzen ähnliche Formatangaben wie auch Python

scanf

- Benötigt *Zeiger* auf die Variablen, um die Werte zu setzen
- Bei Zeichenketten unbedingt die Größe des Puffers angeben!
- Gibt zurück, wieviele Werte gelesen werden konnten

stdio.h – Datei-Ein-/Ausgabe

```
#include <stdio.h>
FILE *datei = fopen("test.txt", "r");
if (datei == 0) {
  fprintf(stderr, "Kann Datei nicht oeffnen\n");
  return -1;
}
if (fscanf(datei, "%f %d", &fliess, &ganz) == 2) {
  fprintf(stdout, "%f %d\n", fliess, ganz);
}
fclose(datei);
```

- `stdio.h` stellt Dateien durch *Filehandles* vom Typ `FILE *` dar
- Handle ist Zeiger auf 0, wenn Datei nicht geöffnet werden kann
- Dateien öffnen mit `fopen`, schließen mit `fclose`
- Zum Schreiben öffnen mit Modus „w“ oder „a“ statt „r“

INSTITUTE FOR COMPUTATIONAL PHYSICS
University of Stuttgart Germany
http://www.icp.uni-stuttgart.de

stdio.h – Datei-Ein-/Ausgabe


```
#include <stdio.h>
FILE *datei = fopen("test.txt", "r");
if (datei == 0) {
  fprintf(stderr, "Kann Datei nicht oeffnen\n");
  return -1;
}
if (fscanf(datei, "%f %d", &fliess, &ganz) == 2) {
  fprintf(stdout, "%f %d\n", fliess, ganz);
}
fclose(datei);
```


- `fprintf` und `fscanf` funktionieren wie `printf` und `scanf`
- Aber auf beliebigen Dateien statt `stdout` und `stdin`
- `stdin`, `stdout` und `stderr` sind Handles für die Standardgeräte

stdio.h – sscanf

```c
#include <stdio.h>

int main(int argc, char **argv) {
  if (argc < 2) {
    fprintf(stderr, "Kein Parameter gegeben\n");
    return -1;
  }
  if (sscanf(argv[1], "%f", &fliess) == 1) {
    fprintf(stdout, "%f\n", fliess);
  }
  return 0;
}
```

- `sscanf` liest statt aus einer Datei aus einem 0-terminierten String
- Dies ist nützlich, um Argumente umzuwandeln
- Es gibt auch `sprintf` (gefährlich!) und `snprintf`