Kapitel 10

Die Standard C-Library

Standard C-Library

Die **Standard C-Library** (auch C Runtime Library) ist eine Funktionsbibliothek mit einigen hundert Funktionen, die, ebenso wie die Sprache C selbst, durch die ANSI normiert ist. Die Funktionen dieser Bibliothek sind in jeder, dem Standard entsprechenden C-Programmierumgebung verfügbar.

Aufgrund ihres Umfangs können wir diese Funktionsbibliothek nur auszugsweise und anhand von Beispielen besprechen. Alle Details finden Sie in Ihren Compilerhandbüchern, dem Hilfesystem Ihrer Entwicklungsumgebung oder auf zahlreichen Informationsseiten im Internet. Dort finden Sie auch Informationen darüber, welche Headerfiles Sie in Ihrem Quellcode includieren müssen, um die jeweiligen Funktionen, ihren Prototypen entsprechend, korrekt verwenden zu können.

Mathematische Funktionen

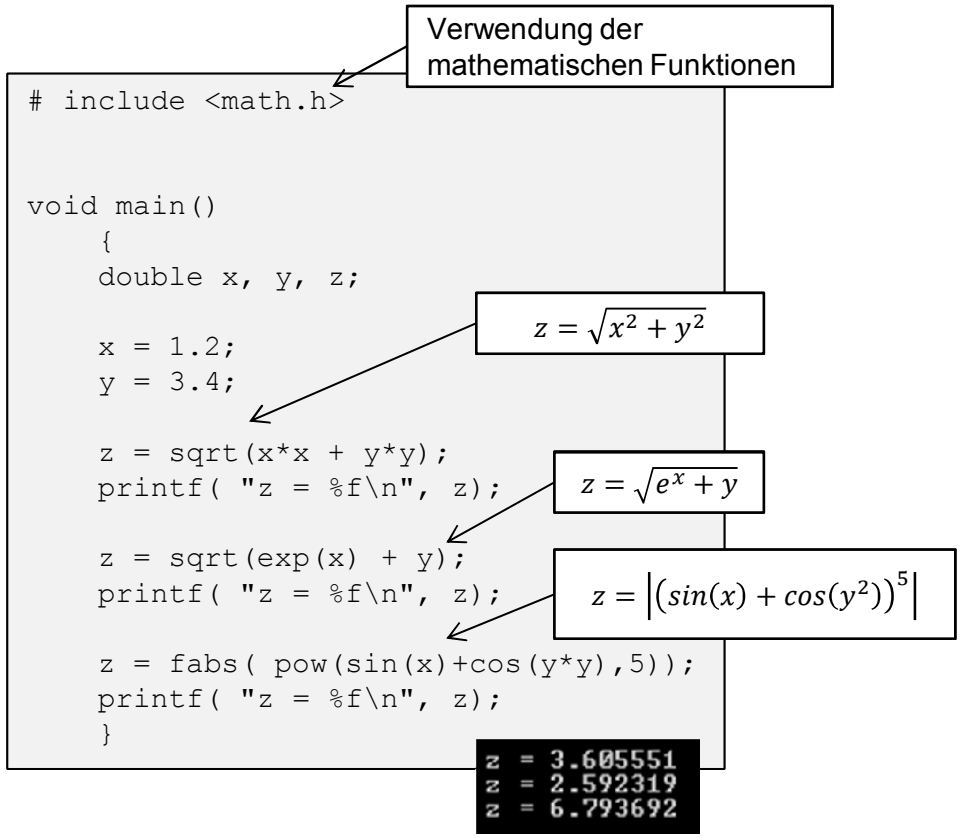
Es gibt in der C-Laufzeitbibliothekt über 50 mathematische Funktionen, von denen die nebenstehende Tabelle einige zeigt.

Um diese Funktionen zu verwenden, müssen Sie `math.h` inkludieren.

Name	Beschreibung	Mathematische Formulierung
`acos`	Arkuskosinus	$\arccos x$
`asin`	Arkussinus	$\arcsin x$
`atan`	Arkustangens	$\arctan x$
`atan2`	„Arkustangens“ mit zwei Argumenten	$\operatorname{atan2}(y, x)$
`ceil`	Aufrundungsfunktion	$\lceil x \rceil$
`cos`	Kosinus	$\cos x$
`cosh`	Kosinus Hyperbolicus	$\cosh x$
`exp`	Exponentialfunktion	e^x
`fabs`	Betragsfunktion	$\lvert x \rvert$
`floor`	Ganzteilfunktion	$\lfloor x \rfloor$
`fmod`	Führt die Modulo Funktion für Gleitkommazahlen durch	$x \bmod y$
`frexp`	Teilt eine Gleitkommazahl in Faktor und Potenz mit der Basis 2 auf	
`ldexp`	Multipliziert den ersten Parameter mit 2 um den zweiten Parameter potenziert	$x2^y$
`log`	Natürlicher Logarithmus	$\ln x$
`log10`	Logarithmus zur Basis 10	$\log_{10} x$
`modf`	Teilt eine Gleitkommazahl in zwei Zahlen auf, vor und nach dem Komma	
`pow`	Potenziert ersten mit dem zweiten Parameter	x^y
`sin`	Sinus	$\sin x$
`sinh`	Sinus Hyperbolicus	$\sinh x$
`sqrt`	Quadratwurzel	$\sqrt{x}$
`tan`	Tangens	$\tan x$
`tanh`	Tangens Hyperbolicus	$\tanh x$

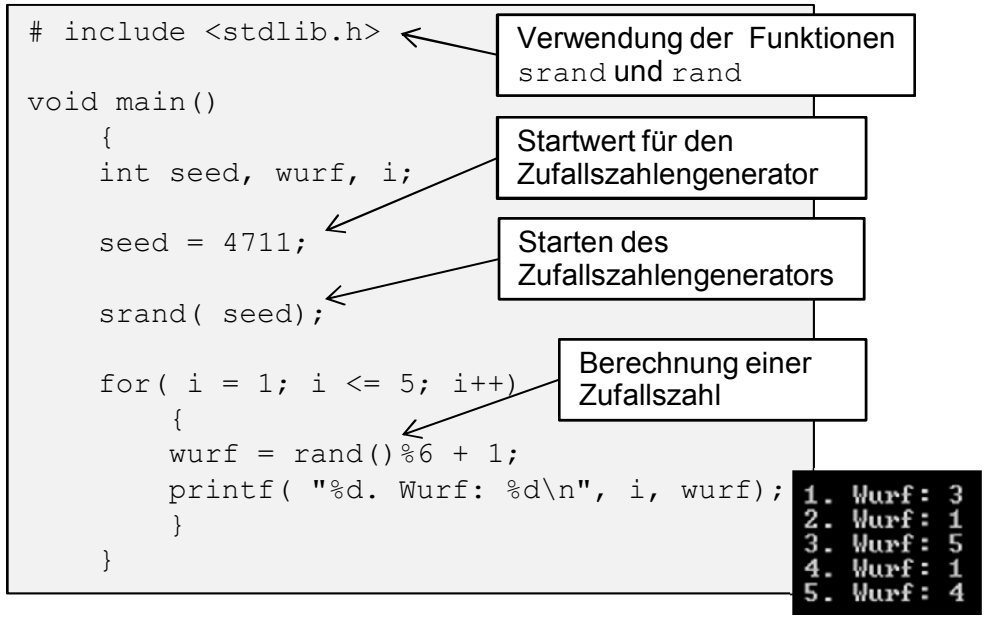
Quelle: Wikipedia

Verwendung einiger mathematischer Funktionen



Zufallszahlen

Mit der Runtime-Library können (Pseudo-) Zufallszahlen erzeugt werden.




```
# include <stdlib.h>

void main()
    {
    int seed, wurf, i;

    seed = 4711;

    srand( seed);

    for( i = 1; i <= 5; i++)
        {
        wurf = rand()%6 + 1;
        printf( "%d. Wurf: %d\n", i, wurf);
        }
    }
```

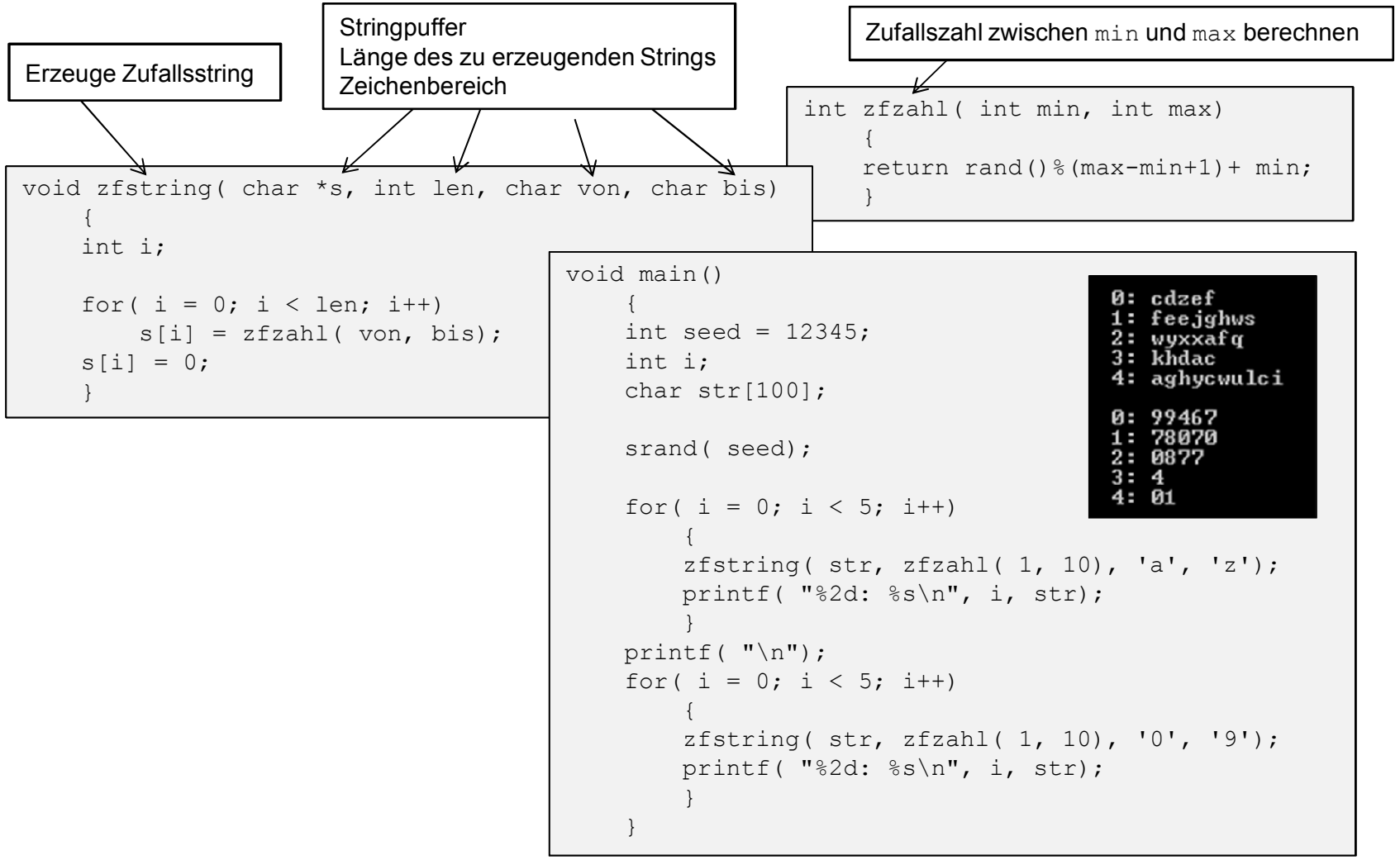
Nachdem der Zufallszahlengenerator mit einem Startwert initialisiert wurde, können mit der Funktion `rand` gleichverteilte Zufallszahlen im Bereich von `0` bis `RAND_MAX` abgerufen werden. Üblicherweise werden diese Zahlen dann noch durch eine Modulo-Operation und eine Verschiebung in den gewünschten Bereich transformiert. Benötigt man etwa Zufallszahlen im Bereich zwischen a und b (einschließlich), so erhält man diese durch die Formel

```
x = rand()%(b-a+1)+a
```

Bei gleichem Startwert erhält man immer die gleiche Folge von Zufallszahlen. Darum verwendet man häufig flüchtige Systemdaten (z.B. Systemzeit) zur Initialisierung des Zufallszahlengenerators.

Erzeugung von zufälligen Zeichenketten

Zufallszahlen können verwendet werden, um Programme intensiv zu testen.



Funktionen bzw. Makros zur Zeichenklassifizierung und –konvertierung

Zähle alle Klein- bzw. Großbuchstaben in der Benutzereingabe und gib den Eingabetext komplett in Groß- bzw. Kleinschreibung aus.

```
# include <ctype.h>

void main()
   {
   char text[100];
   int u, l;
   char *p;

   printf( "Eingabe: ");
   scanf( "%s", text);

   for( p = text, u = l = 0; *p; p++)
       {
       if( isupper( *p))
           u++;
       if( islower( *p))
           l++;
       }
   printf( "%d Gross-, %d Kleinbuchstaben\n", u, l);

   for( p = text; *p; p++)
       *p = toupper( *p);
   printf( "Gross:   %s\n", text);

   for( p = text; *p; p++)
       *p = tolower( *p);
   printf( "Klein:   %s\n", text);
   }
```

Verwendung der Funktionen zur Zeichenkonvertierung

Zählen der Großbuchstaben

Zählen der Kleinbuchstaben

Zeichenweise Konvertierung in Großbuchstaben

Zeichenweise Konvertierung in Kleinbuchstaben

```
Eingabe: AbCdEfGhIjKlMnOpQrStUvWxYz
13 Gross-, 13 Kleinbuchstaben
Gross:   ABCDEFGHIJKLMNOPQRSTUVWXYZ
Klein:   abcdefghijklmnopqrstuvwxyz
```

Einige wichtige Stringfunktionen

strcpy	Kopiere einen String
strncpy	Kopiere eine bestimmte Anzahl Zeichen aus einem String
strcat	Hänge zwei Strings aneinander
strncat	Hänge eine bestimmte Anzahl Zeichen aus einem String an einen anderen String an
strcmp	Vergleiche zwei Strings
strncmp	Vergleiche eine bestimmter Anzahl Zeichen in zwei Strings
strchr	Finde das erste Vorkommen eines Zeichens in einem String
strrchr	Finde das letzte Vorkommen eines Zeichens in einem String
strstr	Suche einen String in einem String
strlen	Berechne die Länge eines Strings

Ein Beispiel mit Stringfunktionen

Kette alle Benutzereingaben aneinander, bis der Benutzer den Prozess durch Eingabe von "`ende`" abbricht.

```
# include <string.h>

void main()
   {
   char eingabe[100];
   char text[500];

   for( text[0] = 0; ; )
       {
       printf( "Eingabe: ");
       scanf( "%s", eingabe);

       if( strcmp( eingabe, "ende") == 0)
           break;

       if( strlen(text) + strlen(eingabe) < 500)
           strcat( text, eingabe);
       }
   printf( "%s\n", text);
   }
```

Verwendung der Stringfunktionen

Puffer für die Eingabe und den kumulierten Text

Kumulierter Text ist leer

Wenn "ende" eingegeben wird, endet die Schleife

Prüfen, ob ausreichend Platz vorhanden

Anfügen der Eingabe an den kumulierten Text

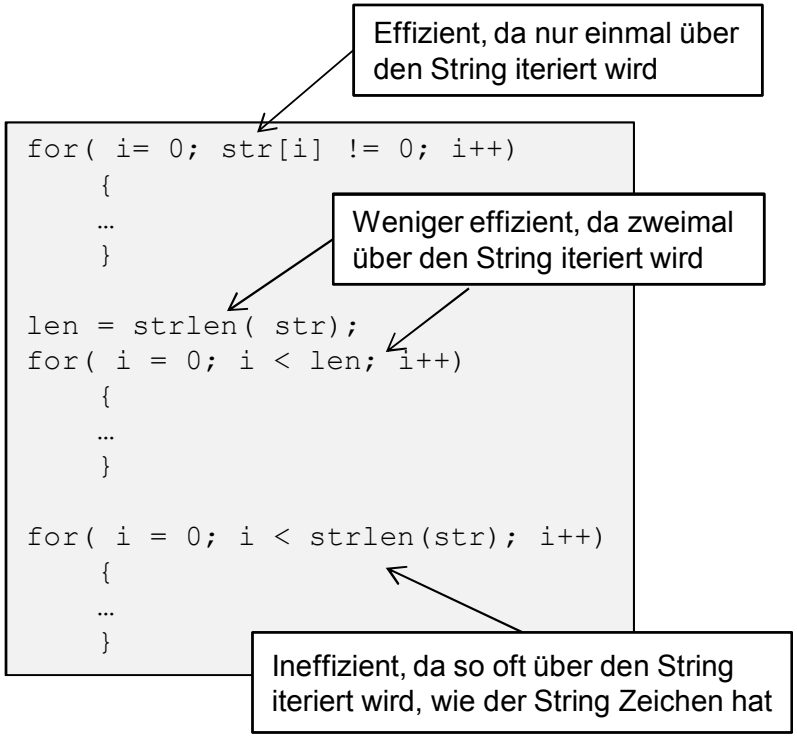
```
Eingabe: the
Eingabe: quick
Eingabe: brown
Eingabe: fox
Eingabe: jumps
Eingabe: over
Eingabe: the
Eingabe: lazy
Eingabe: dog
Eingabe: ende
thequickbrownfoxjumpsoverthelazydog
```

Achten Sie darauf, dass Strings immer konsistent sind und bleiben!

Prüfen Sie bei Veränderungen, ob der zugrunde liegende Array ausreichend groß ist und der Terminator korrekt gesetzt ist!

Stringfunktionen

Achten Sie auf die Effizienz Ihre Codes, da Funktionsaufrufe immer mit Laufzeitkosten verbunden sind. Dies ist besonders bei Stringfunktionen zu beachten, da diese Funktionen in der Regel Zeichen für Zeichen über den String iterieren, um ihre Aufgabe zu erledigen:



Der Umfang des Quellcodes ist kein Maß für die Effizienz des compilierten Codes!

Dateioperationen

Dateien sind aus Sicht eines C-Programms sogenannte Streams (Datenströme), aus denen gelesen oder in die geschrieben werden kann.

Bevor aus Dateien gelesen oder in Dateien geschrieben werden kann, müssen die Dateien als Stream zum Lesen oder Schreiben geöffnet (`fopen`) werden.

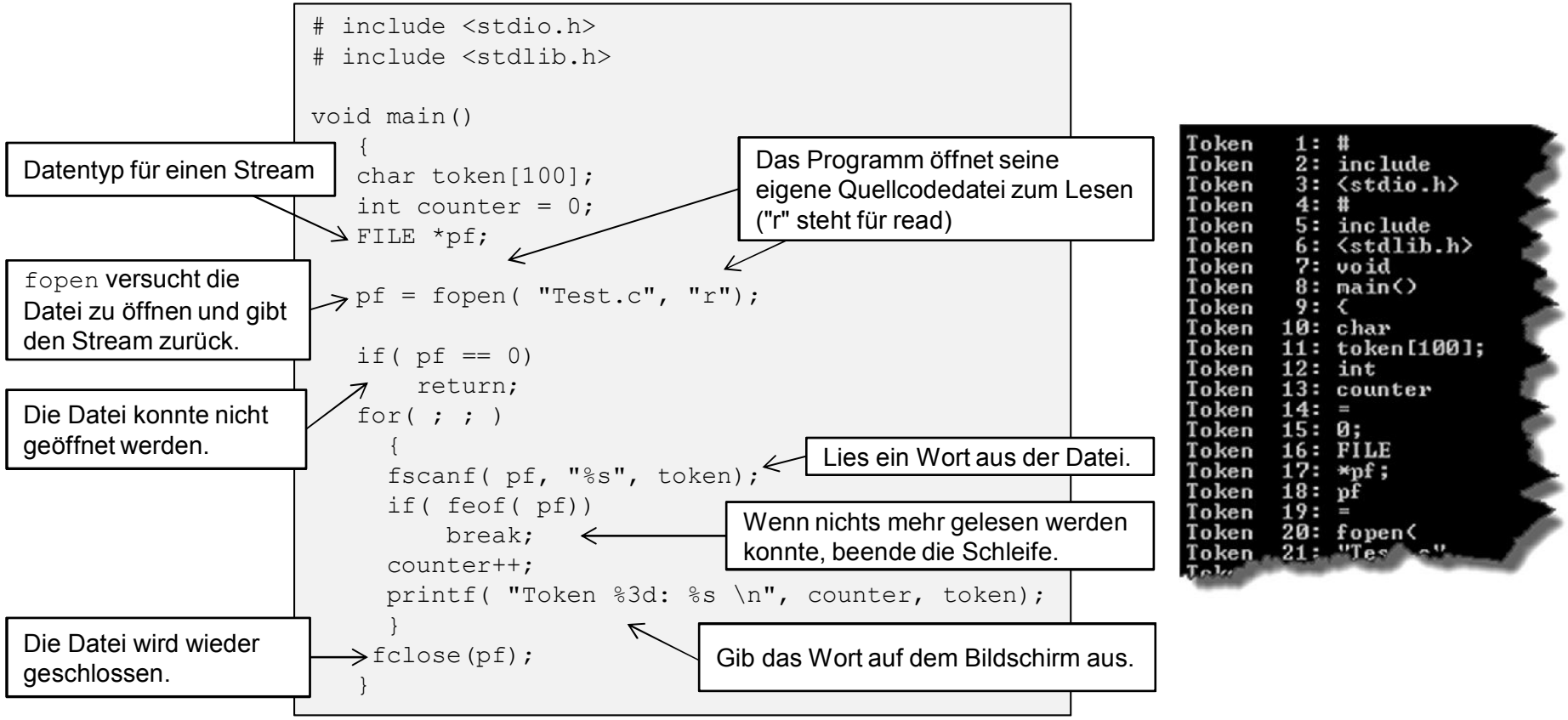
Danach kann aus dem Stream gelesen oder in den Stream geschrieben werden. Es gibt zahlreiche Funktionen zum Lesen und Schreiben. Zur formatierten Ein- bzw. Ausgabe verwendet man die Funktionen `fscanf` bzw. `fprintf`, die den Funktionen `scanf` und `printf` sehr ähnlich sind.

Abweichend von Tastatur und Bildschirm kann man sich in Dateien "frei" bewegen (`fseek`, `ftell`) und an beliebigen Positionen schreiben oder lesen.

Es können parallel mehrere Dateien geöffnet sein. Geöffnete Dateien, auf die nicht mehr zugegriffen wird, sollten geschlossen werden (`fclose`), um die damit verbundenen Systemressourcen wieder freizugeben.

Anwendung von Dateioperationen

Das folgende Programm liest seinen eigenen Quellcode ein und gibt ihn Wort für Wort (Token) auf dem Bildschirm aus.



Standardstreams

Tastatur und Bildschirm sind Lesen bzw. Schreiben geöffnete Streams und können daher auch mit den Dateioperationen bearbeitet werden.

```
void main()
   {
   char name[100];
   int alter;

   fprintf( stdout, "Bitte gib deinen Namen und dein Alter an: " );
   fscanf( stdin, "%s %d", name, &alter);
   fprintf( stdout, "Du heisst %s und bist %d Jahre alt.\n", name, alter);
   }
```

Wie `printf` und `scanf`, wobei im ersten Parameter der Stream steht.

```
Bitte gib deinen Namen und dein Alter an: Otto 42
Du heisst Otto und bist 42 Jahre alt.
```

Es gibt drei vordefinierte Streams

- `stdin` Standardeingabe (Tastatur)
- `stdout` Standardausgabe (Bildschirm)
- `stderr` Standardfehlerausgabe (Bildschirm)

Diese Streams (`stdin`, `stdout`) werden bei Programmstart vom Laufzeitsystem geöffnet und bei Programmende vom Laufzeitsystem wieder geschlossen.

Streams können bei Bedarf mit `freopen` umgelenkt werden. Zum Beispiel kann die Fehlerausgabe in eine Datei umgelenkt werden oder die Tastatureingeben können aus einer Datei gelesen werden.

Funktionen mit variabler Argumentzahl

Funktionen können eine unbestimmte Anzahl Parameter haben.

Es gibt Makros, mit denen man zur Laufzeit die effektiv übergeben Parameter vom Stack holen kann.

Die Funktionen `printf` und `scanf` arbeiten nach diesem Prinzip.

```
void main()
   {
   int a=1, b=2, c=3, d=4;
   int x;

   x = summe( 2, a, b);
   printf( "%d\n", x);
   x = summe( 3, a, b, c);
   printf( "%d\n", x);
   x = summe( 4, a, b, c, d);
   printf( "%d\n", x);
}
```

```
3
6
10
```

Die Funktion `summe` wird mit unterschiedlicher Parameterzahl gerufen

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

# include <stdarg.h>

int summe( int anz,...)
    {
    va_list ap;
    int sum;
    int summand;

    va_start( ap, anz);
    for( sum = 0; anz; anz--)
        {
        summand = va_arg( ap, int);
        sum += summand;
        }
    va_end( ap);
    return sum;
    }
```

Notwendiges Include für Funktionen mit variabler Argumentzahl

Der erste Parameter gibt an, wie viele Parameter folgen. Alle weiteren Parameter sind unspezifiziert

Stackpointer für den Parameterzugriff

Initialisieren des Stackpointers hinter dem Parameter `anz`

Schleife über alle unspezifizierten Parameter

Lesen des nächsten `int`-Parameters vom Stack*

Ende der Stackoperationen

* Die Anweisung `va_arg( ap, int)` ist etwas verwirrend, da es so wirkt, als würde der Datentyp `int` als Parameter an eine Funktion übergeben, was natürlich nicht möglich ist. Tatsächlich handelt es sich bei `va_arg` aber um einen Macro der zu einem Stackzugriff mit dem Datentyp `int` aufgelöst wird. Die konkrete Definition des Macros finden Sie in `stdarg.h`.

Dynamische Speicherverwaltung

Mit den Funktionen zur dynamischen Speicherverwaltung kann man sich von den Fesseln der zur Compilezeit fest angelegten Daten befreien.

Mit `malloc` und `calloc` kann man zur Laufzeit dynamisch Speicher holen (allokieren),
Mit `realloc` kann man Speicher allokieren bzw. allokierten Speicher vergrößern
Mit `free` kann man allokierten Speicher freigeben.

Bevor man Speicher allokiert, muss man den Speicherbedarf ermitteln

Speicher wird in der Regel für eine bestimmte Anzahl von Daten eines bestimmten Typs (z.B. `int`) benötigt. Den Speicherbedarf eines Datentyps bestimmt man mit dem `sizeof`-Operator. Für `100 float`-Zahlen benötigt man zum Beispiel `100*sizeof(float)` Bytes.

Wenn man Speicher mit `malloc`, `calloc` oder `realloc` allokiert, erhält man als Rückgabewert die Adresse des allokierten Speichers.

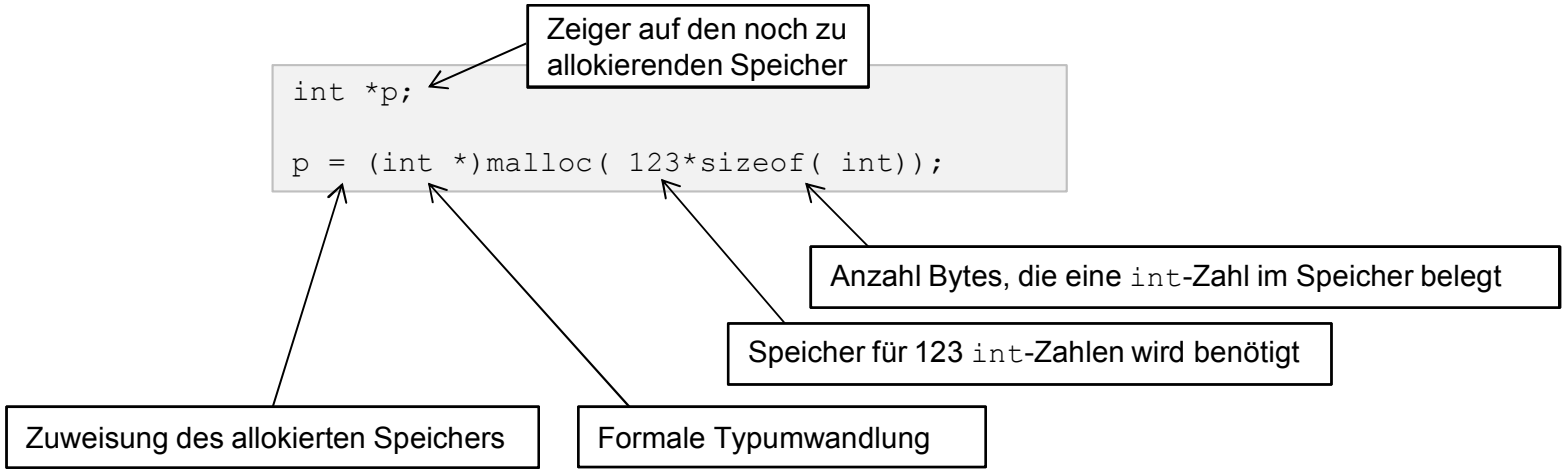
Man benötigt einen Zeiger passenden Typs (z.B. `float *`), um die Adresse zu speichern. Dazu weist man dem Zeiger den Rückgabewert von `malloc`, `calloc` oder `realloc` zu. Über diesen Zeiger kann man dann auf den Speicher zugreifen. Der Programmierer muss "verantwortlich" mit dem Zeiger umgehen und darauf achten, dass er mit dem Zeiger nur innerhalb des ihm zugewiesenen Speicherbereichs zugreift.

Wenn man allokierten Speicher nicht mehr benötigt, muss man ihn wieder freigeben

Zur Freigabe gibt man den beim Allokieren erhaltenen Zeiger wieder ab. Nach der Freigabe darf der Zeiger nicht mehr verwendet werden, es sei denn, dass ihm erneut Speicher zugewiesen wird.

Dynamische Speicherallokierung mit `malloc`

Bevor man Speicher allokiert, muss man wissen, wie viel Speicher man für welchen Datentyp benötigt. Im folgenden Beispiel wird Speicher für `123` Integer-Zahlen allokiert:



Über den Zeiger kann dann auf den Speicher zugegriffen werden. Im Beispiel wird der Speicher mit 0 initialisiert:

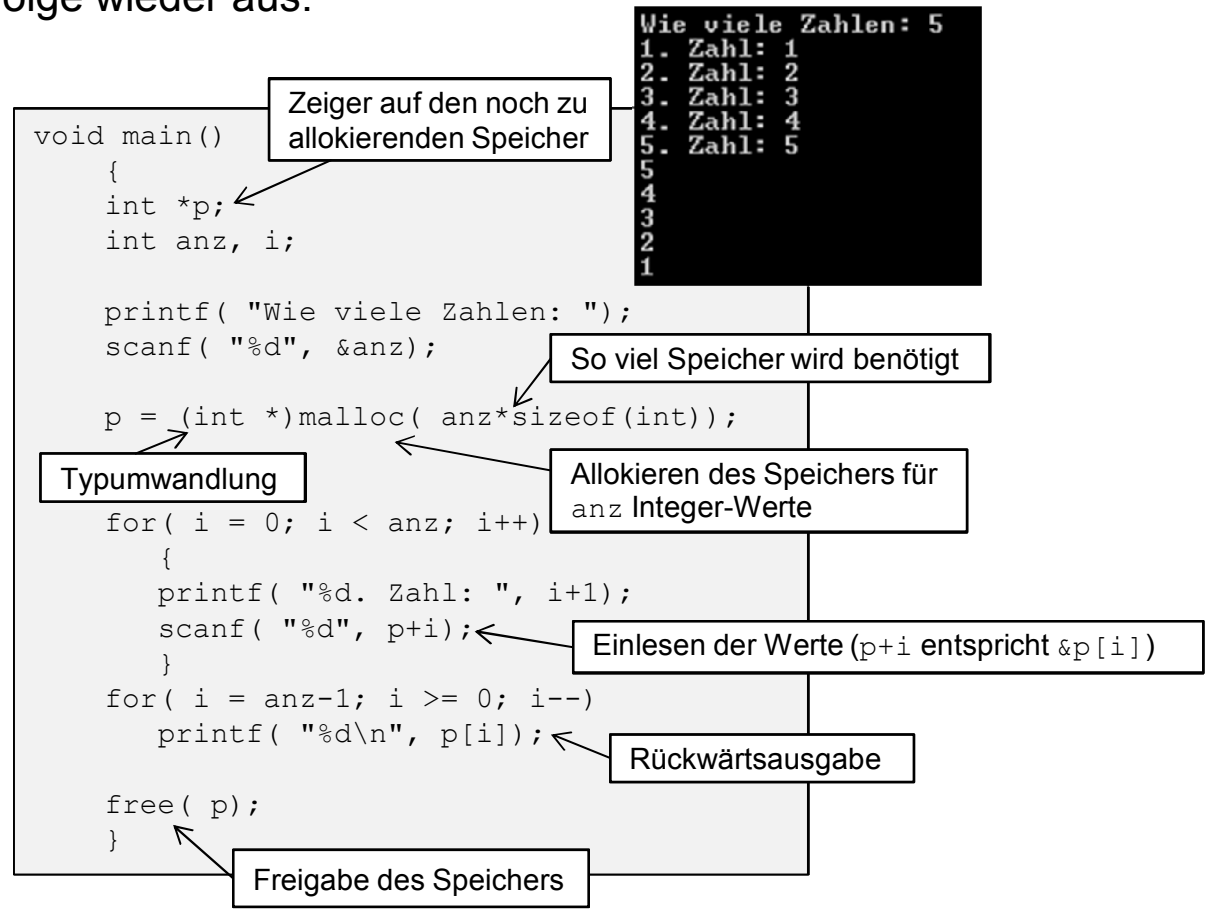
```
int i;

for( i = 0; i < 123; i++)
    p[i] = 0;
```

Der Speicher wird mit `free` wieder freigegeben, wenn er nicht mehr benötigt wird:

```
free(p);
```

Beispiel mit dynamischer Speicherverwaltung 1

Lies eine vom Benutzer vorab gewählte Anzahl an Zahlen ein und gib sie in umgekehrter Reihenfolge wieder aus.

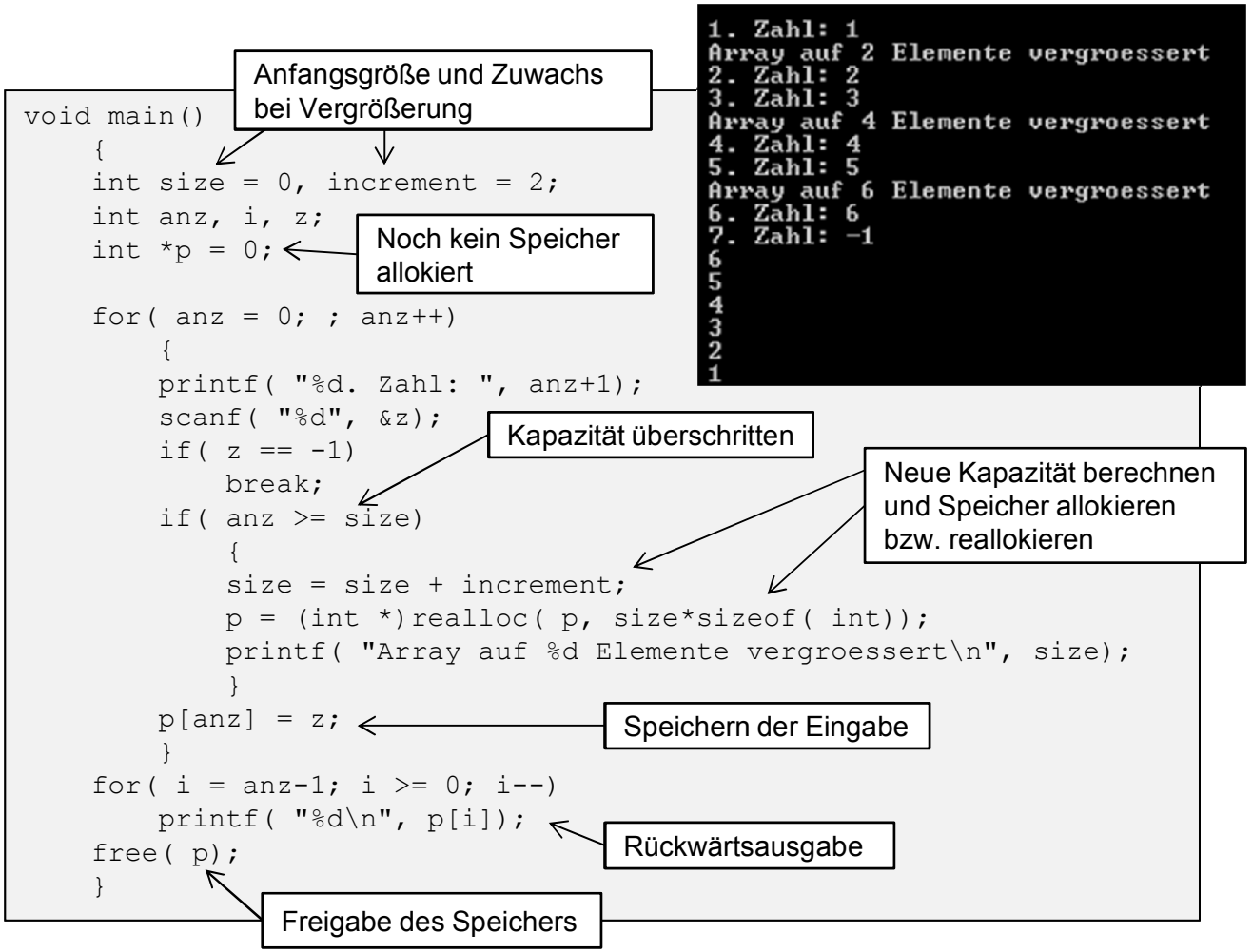



```
void main()
    {
    int *p;
    int anz, i;

    printf( "Wie viele Zahlen: ");
    scanf( "%d", &anz);

    p = (int *)malloc( anz*sizeof(int));

    for( i = 0; i < anz; i++)
       {
       printf( "%d. Zahl: ", i+1);
       scanf( "%d", p+i);
       }
    for( i = anz-1; i >= 0; i--)
       printf( "%d\n", p[i]);

    free( p);
    }
```

Dieses Programm ist nur eingeschränkt flexibel. Die Größe des Arrays wird zwar nicht mehr zur Compilezeit festgelegt, aber der Benutzer muss zur Laufzeit vorab festlegen, wie viele Zahlen er eingeben will.

Beispiel mit dynamischer Speicherverwaltung 2

Lies eine beliebige Anzahl Zahlen ein und gib sie in umgekehrter Reihenfolge wieder aus.




```
void main()
    {
    int size = 0, increment = 2;
    int anz, i, z;
    int *p = 0;

    for( anz = 0; ; anz++)
        {
        printf( "%d. Zahl: ", anz+1);
        scanf( "%d", &z);
        if( z == -1)
            break;
        if( anz >= size)
            {
            size = size + increment;
            p = (int *)realloc( p, size*sizeof( int));
            printf( "Array auf %d Elemente vergroessert\n", size);
            }
        p[anz] = z;
        }
    for( i = anz-1; i >= 0; i--)
        printf( "%d\n", p[i]);
    free( p);
    }
```

Dieses Programm ist voll flexibel, da der Array bei Bedarf mit `realloc` vergrößert wird. Normalerweise arbeitet man natürlich nicht so "kleinschrittig" wie hier gezeigt.

Einige wichtige Funktionen für Speicheroperationen

memcpy	Kopiere einen Speicherblock
memmove	Verschiebe einen Speicherblock
memcmp	Vergleiche zwei Speicherblöcke
memchr	Finde ein Zeichen in einem Speicherblock
memset	Initialisiere einen Speicherblock