Hardwarenahe Programmierung I

U. Kaiser, R. Kaiser, M. Stöttinger, S. Reith

(HTTP: http://www.cs.hs-rm.de/~kaiser
EMail: robert.kaiser@hs-rm.de)

Wintersemester 2021/2022

12. C-Bibliothek

http://www.denkschatz.de/zitate/Jorge-Luis-Borges/Das-Paradies-habe-ich-mir-immer-als-eine-Art-Bibliothek-vorgestellt

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 neben- stehende 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


```
# include <math.h>

void main()
{
    double x, y, z;
    x = 1.2;
    y = 3.4;

    z = sqrt(x*x + y*y);
    printf( "z = %f\n", z);

    z = sqrt(exp(x) + y);
    printf( "z = %f\n", z);

    z = fabs( pow(sin(x)+cos(y*y),5));
    printf( "z = %f\n", z);
}
```

```
z = 3.60551
z = 1.592319
z= 6.793692
```

Verwendung einiger mathematischer Funktionen

Verwendung der mathematischen Funktionen

```
# include <math.h>

void main()
{
    double x, y, z;
    x = 1.2;
    y = 3.4;

    z = sqrt(x*x + y*y);
    printf( "z = %f\n", z);

    z = sqrt(exp(x) + y);
    printf( "z = %f\n", z);

    z = fabs( pow(sin(x)+cos(y*y),5));
    printf( "z = %f\n", z);
}
```

```
z = 3.60551
z = 1.592319
z= 6.793692
```

Verwendung einiger mathematischer Funktionen

Verwendung der mathematischen Funktionen

```
# include <math.h>

void main()
{
    double x, y, z;
    x = 1.2;
    y = 3.4;

    z = sqrt(x*x + y*y);
    printf( "z = %f\n", z);

    z = sqrt(exp(x) + y);
    printf( "z = %f\n", z);

    z = fabs( pow(sin(x)+cos(y*y),5));
    printf( "z = %f\n", z);
}
```

$$z = \sqrt{x^2 + y^2}$$

```
z = 3.60551
z = 1.592319
z= 6.793692
```

Verwendung einiger mathematischer Funktionen

Verwendung der mathematischen Funktionen

```
# include <math.h>

void main()
{
    double x, y, z;
    x = 1.2;
    y = 3.4;

    z = sqrt(x*x + y*y);
    printf( "z = %f\n", z);

    z = sqrt(exp(x) + y);
    printf( "z = %f\n", z);

    z = fabs( pow(sin(x)+cos(y*y),5));
    printf( "z = %f\n", z);
}
```

$z = \sqrt{x^2 + y^2}$

$z = \sqrt{e^x + y}$

```
z = 3.60551
z = 1.592319
z= 6.793692
```

Verwendung einiger mathematischer Funktionen

Verwendung der mathematischen Funktionen

```
# include <math.h>

void main()
{
    double x, y, z;
    x = 1.2;
    y = 3.4;

    z = sqrt(x*x + y*y);
    printf( "z = %f\n", z);

    z = sqrt(exp(x) + y);
    printf( "z = %f\n", z);

    z = fabs( pow(sin(x)+cos(y*y),5));
    printf( "z = %f\n", z);
}
```

$z = \sqrt{x^2 + y^2}$

$z = \sqrt{e^x + y}$

$z = |(sin(x) + cos(y^2))^5|$

```
z = 3.60551
z = 1.592319
z= 6.793692
```

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);
      }
}
```

```
1. Wurf: 3
2. Wurf: 1
3. Wurf: 5
4. Wurf: 1
5. Wurf: 4
```

Zufallszahlen

Hochschule RheinMain

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

Verwendung der Funk
srand und rand

```
# 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);
      }
}
```

```
1. Wurf: 3
2. Wurf: 1
3. Wurf: 5
4. Wurf: 1
5. Wurf: 4
```

Zufallszahlen

Hochschule RheinMain

- 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);
      }
}
```

Verwendung der Funk srand und rand

Startwert für den Zufallszahlengenerator

```
1. Wurf: 3
2. Wurf: 1
3. Wurf: 5
4. Wurf: 1
5. Wurf: 4
```

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);
      }
}
```

Verwendung der Funk srand und rand

Startwert für den Zufallszahlengenerator

Starten des Zufallszahlengenerators

```
1. Wurf: 3
2. Wurf: 1
3. Wurf: 5
4. Wurf: 1
5. Wurf: 4
```

Zufallszahlen

Hochschule RheinMain

- 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);
      }
}
```

Verwendung der Funk srand und rand

Startwert für den Zufallszahlengenerator

Starten des Zufallszahlengenerators

Berechnung einer Zufallszahl

```
1. Wurf: 3
2. Wurf: 1
3. Wurf: 5
4. Wurf: 1
5. Wurf: 4
```

Zufallszahlen

- 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.

```
int zfzahl( int min, int max)
{
    return rand()%(max-min+1)+ min;
}
```

```
void zfstr(char *s,int len,char von,char bis)
{
    int i;
    for( i = 0; i < len; i++)
        s[i] = zfzahl( von, bis);
    s[i] = 0;
}
```

```
void main()
{
    int seed = 12345;
    int i;
    char str[100];

    srand( seed);

    for( i = 0; i < 5; i++)
    {
        zfstr(str, zfzahl(1,10),'a','z');
        printf( "%2d: %s\n", i, str);
    }
    printf( "\n");
    for( i = 0; i < 5; i++)
    {
        zfstr(str, zfzahl(1,10),'0','9');
        printf( "%2d: %s\n", i, str);
    }
}
```

```
0: cdzef
1: feejghws
2: wyxxafq
3: khdac
4: aghycwulci

0: 99467
1: 78070
2:0877
3: 4
4: 01
```

Erzeugung von zufälligen Zeichenketten

- Zufallszahlen können verwendet werden, um Programme intensiv zu testen.

Zufallszahl zwischen min und max berechnen

```
int zfzahl( int min, int max)
{
    return rand()%(max-min+1)+ min;
}
```

Erzeuge Zufallsstring

```
void zfstr(char *s,int len,char von,char bis)
{
    int i;
    for( i = 0; i < len; i++)
        s[i] = zfzahl( von, bis);
    s[i] = 0;
}
```

```
void main()
{
    int seed = 12345;
    int i;
    char str[100];

    srand( seed);

    for( i = 0; i < 5; i++)
    {
        zfstr(str, zfzahl(1,10),'a','z');
        printf( "%2d: %s\n", i, str);
    }
    printf( "\n");
    for( i = 0; i < 5; i++)
    {
        zfstr(str, zfzahl(1,10),'0','9');
        printf( "%2d: %s\n", i, str);
    }
}
```

```
0: cdzef
1: feejghws
2: wyxxafq
3: khdac
4: aghycwulci

0: 99467
1: 78070
2:0877
3: 4
4: 01
```

Erzeugung von zufälligen Zeichenketten

Hochschule RheinMain

- Zufallszahlen können verwendet werden, um Programme intensiv zu testen.

Zufallszahl zwischen min und max berechnen

```
int zfzahl( int min, int max)
{
    return rand()%(max-min+1)+ min;
}
```

Erzeuge Zufallsstring

Stringpuffer Länge des zu erzeugenden Strings Zeichenbereich

```
void zfstr(char *s,int len,char von,char bis)
{
    int i;
    for( i = 0; i < len; i++)
        s[i] = zfzahl( von, bis);
    s[i] = 0;
}
```

```
void main()
{
    int seed = 12345;
    int i;
    char str[100];

    srand( seed);

    for( i = 0; i < 5; i++)
    {
        zfstr(str, zfzahl(1,10),'a','z');
        printf( "%2d: %s\n", i, str);
    }
    printf( "\n");
    for( i = 0; i < 5; i++)
    {
        zfstr(str, zfzahl(1,10),'0','9');
        printf( "%2d: %s\n", i, str);
    }
}
```

```
0: cdzef
1: feejghws
2: wyxxafq
3: khdac
4: aghycwulci

0: 99467
1: 78070
2:0877
3: 4
4: 01
```

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);
}
```

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

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

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

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

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

Zeichenklassifizierung und –konvertierung

Hochschule RheinMain

- 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

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

Zählen der Großbuchstaben

Zählen der Kleinbuchstaben

Zeichenweise Konvertierung in Großbuchstaben

Zeichenklassifizierung und –konvertierung

Hochschule RheinMain

- 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

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

Zählen der Großbuchstaben

Zählen der Kleinbuchstaben

Zeichenweise Konvertierung in Großbuchstaben

Zeichenweise Konvertierung in Kleinbuchstaben

Einige wichtige Stringfunktionen

Befehl	Funktionsbeschreibung
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);
}
```

```
Eingabe: Das
Eingabe: Ist
Eingabe: Ein
Eingabe: Test!
Eingabe: ende
DasIstEinTest!
```

- Achten Sie darauf, dass Strings immer konsistent sind und bleiben!
- Prüfen Sie bei Veränderungen, ob das zugrunde liegende Array ausreichend groß ist und der Terminator korrekt gesetzt ist!

Ein Beispiel mit Stringfunktionen

- Kette alle Benutzereingaben aneinander, bis der Benutzer den Prozess durch Eingabe von „ende“ abbricht.

Verwendung der Stringfunktionen

```
# 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);
}
```

```
Eingabe: Das
Eingabe: Ist
Eingabe: Ein
Eingabe: Test!
Eingabe: ende
DasIstEinTest!
```

- Achten Sie darauf, dass Strings immer konsistent sind und bleiben!
- Prüfen Sie bei Veränderungen, ob das zugrunde liegende Array ausreichend groß ist und der Terminator korrekt gesetzt ist!

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

```
Eingabe: Das
Eingabe: Ist
Eingabe: Ein
Eingabe: Test!
Eingabe: ende
DasIstEinTest!
```

- Achten Sie darauf, dass Strings immer konsistent sind und bleiben!
- Prüfen Sie bei Veränderungen, ob das zugrunde liegende Array ausreichend groß ist und der Terminator korrekt gesetzt ist!

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

```
Eingabe: Das
Eingabe: Ist
Eingabe: Ein
Eingabe: Test!
Eingabe: ende
DasIstEinTest!
```

- Achten Sie darauf, dass Strings immer konsistent sind und bleiben!
- Prüfen Sie bei Veränderungen, ob das zugrunde liegende Array ausreichend groß ist und der Terminator korrekt gesetzt ist!

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

```
Eingabe: Das
Eingabe: Ist
Eingabe: Ein
Eingabe: Test!
Eingabe: ende
DasIstEinTest!
```

- Achten Sie darauf, dass Strings immer konsistent sind und bleiben!
- Prüfen Sie bei Veränderungen, ob das zugrunde liegende Array ausreichend groß ist und der Terminator korrekt gesetzt ist!

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

```
Eingabe: Das
Eingabe: Ist
Eingabe: Ein
Eingabe: Test!
Eingabe: ende
DasIstEinTest!
```

- Achten Sie darauf, dass Strings immer konsistent sind und bleiben!
- Prüfen Sie bei Veränderungen, ob das zugrunde liegende Array ausreichend groß ist und der Terminator korrekt gesetzt ist!

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: Das
Eingabe: Ist
Eingabe: Ein
Eingabe: Test!
Eingabe: ende
DasIstEinTest!
```

- Achten Sie darauf, dass Strings immer konsistent sind und bleiben!
- Prüfen Sie bei Veränderungen, ob das 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:

```
for( i= 0; str[i] != 0; i++)
{
    ...
}

len = strlen( str);
for( i = 0; i < len; i++)
{
    ...
}

for( i = 0; i < strlen(str); i++)
{
   ...
}
```

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

Stringfunktionen

Hochschule RheinMain

- 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:

Effizient, da nur einmal über den String iteriert wird

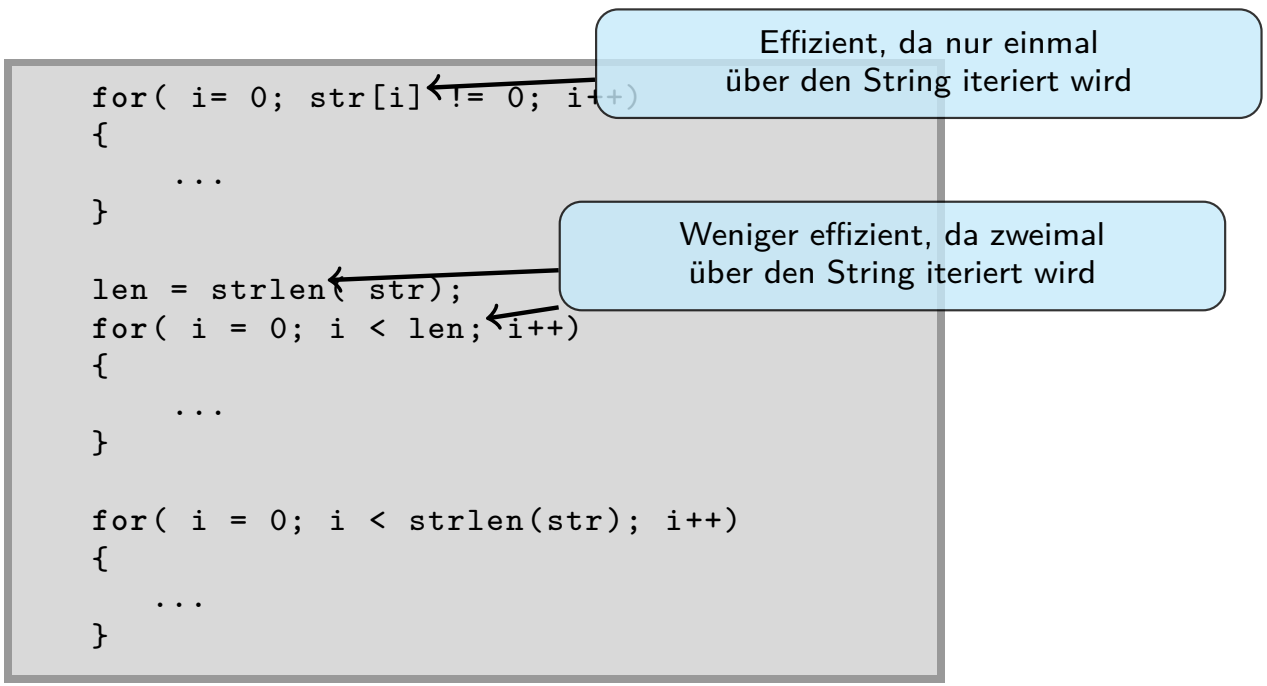
```
for( i= 0; str[i] != 0; i++)
{
    ...
}

len = strlen( str);
for( i = 0; i < len; i++)
{
    ...
}

for( i = 0; i < strlen(str); i++)
{
   ...
}
```

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

Stringfunktionen

Hochschule RheinMain

- 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:




```
for( i= 0; str[i] != 0; i++)
{
    ...
}

len = strlen( str);
for( i = 0; i < len; i++)
{
    ...
}

for( i = 0; i < strlen(str); i++)
{
   ...
}
```

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

Stringfunktionen

Hochschule RheinMain

- 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:

```
for( i= 0; str[i] != 0; i++)
{
    ...
}

len = strlen( str);
for( i = 0; i < len; i++)
{
    ...
}

for( i = 0; i < strlen(str); i++)
{
   ...
}
```

Effizient, da nur einmal über den String iteriert wird

Weniger effizient, da zweimal über den String iteriert wird

Ineffizient, da oft über den String iteriert wird, wie der String Zeichen hat

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

Dateioperationen

- Dateien sind aus Sicht eines C-Programms sogenannte *Streams*, aus denen gelesen oder in die geschrieben werden kann.
- Aus Dateien kann gelesen oder in Dateien kann geschrieben werden, wenn 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.

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

void main()
 {
    char token[100];
    int counter = 0;
    FILE *pf;

    pf = fopen( "Test.c", "r");

    if( pf == 0)
        return;
    for( ; ; )
    {
        fscanf( pf, "%s", token);
        if( feof( pf))
            break;
        counter++;
        printf( "Token %3d: %s \n", counter, token);
    }
    fclose(pf);
}
```

```
Token:   1: #
Token:   2: include
Token:   3: <stdio.h>
Token:   4: #
Token:   5: Include
Token:   6: #
Token:   7: include
Token:   8: <stdlib.h>
Token:   9: void
...
```

Anwendung von Dateioperationen

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

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

void main()
 {
    char token[100];
    int counter = 0;
    FILE *pf;

    pf = fopen( "Test.c", "r");

    if( pf == 0)
        return;
    for( ; ; )
    {
        fscanf( pf, "%s", token);
        if( feof( pf))
            break;
        counter++;
        printf( "Token %3d: %s \n", counter, token);
    }
    fclose(pf);
}
```

Datentyp für einen Stream

```
Token:   1: #
Token:   2: include
Token:   3: <stdio.h>
Token:   4: #
Token:   5: Include
Token:   6: #
Token:   7: include
Token:   8: <stdlib.h>
Token:   9: void
...
```

Anwendung von Dateioperationen

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

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

void main()
 {
    char token[100];
    int counter = 0;
    FILE *pf;

    pf = fopen( "Test.c", "r");

    if( pf == 0)
        return;
    for( ; ; )
    {
        fscanf( pf, "%s", token);
        if( feof( pf))
            break;
        counter++;
        printf( "Token %3d: %s \n", counter, token);
    }
    fclose(pf);
}
```

Datentyp für einen Stream

fopen versucht die Datei zu öffnen und gibt den Stream zurück.

```
Token:   1: #
Token:   2: include
Token:   3: <stdio.h>
Token:   4: #
Token:   5: Include
Token:   6: #
Token:   7: include
Token:   8: <stdlib.h>
Token:   9: void
...
```

Anwendung von Dateioperationen

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

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

void main()
 {
    char token[100];
    int counter = 0;
    FILE *pf;

    pf = fopen( "Test.c", "r");

    if( pf == 0)
        return;
    for( ; ; )
    {
        fscanf( pf, "%s", token);
        if( feof( pf))
            break;
        counter++;
        printf( "Token %3d: %s \n", counter, token);
    }
    fclose(pf);
}
```

Datentyp für einen Stream

fopen versucht die Datei zu öffnen und gibt den Stream zurück.

Das Programm öffnet seine eigene Quellcodedatei zum Lesen (*r* steht für read)

```
Token:   1: #
Token:   2:  include
Token:   3: <stdio.h>
Token:   4: #
Token:   5: Include
Token:   6: #
Token:   7:  include
Token:   8: <stdlib.h>
Token:   9: void
...
```

Anwendung von Dateioperationen

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

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

void main()
 {
    char token[100];
    int counter = 0;
    FILE *pf;

    pf = fopen( "Test.c", "r");

    if( pf == 0)
        return;
    for( ; ; )
    {
        fscanf( pf, "%s", token);
        if( feof( pf))
            break;
        counter++;
        printf( "Token %3d: %s \n", counter, token);
    }
    fclose(pf);
}
```

Datentyp für einen Stream

fopen versucht die Datei zu öffnen und gibt den Stream zurück.

Das Programm öffnet seine eigene Quellcodedatei zum Lesen (*r* steht für read)

Die Datei konnte nicht geöffnet werden.

```
Token:   1: #
Token:   2:  include
Token:   3: <stdio.h>
Token:   4: #
Token:   5: Include
Token:   6: #
Token:   7:  include
Token:   8: <stdlib.h>
Token:   9: void
...
```

Anwendung von Dateioperationen

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

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

void main()
 {
    char token[100];
    int counter = 0;
    FILE *pf;

    pf = fopen( "Test.c", "r");

    if( pf == 0)
        return;
    for( ; ; )
    {
        fscanf( pf, "%s", token);
        if( feof( pf))
            break;
        counter++;
        printf( "Token %3d: %s \n", counter, token);
    }
    fclose(pf);
}
```

Datentyp für einen Stream

fopen versucht die Datei zu öffnen und gibt den Stream zurück.

Das Programm öffnet seine eigene Quellcodedatei zum Lesen (*r* steht für read)

Die Datei konnte nicht geöffnet werden.

Lies ein Wort aus der Datei.

```
Token:   1: #
Token:   2:  include
Token:   3: <stdio.h>
Token:   4: #
Token:   5: Include
Token:   6: #
Token:   7:  include
Token:   8: <stdlib.h>
Token:   9: void
...
```

Anwendung von Dateioperationen

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

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

void main()
 {
    char token[100];
    int counter = 0;
    FILE *pf;

    pf = fopen( "Test.c", "r");

    if( pf == 0)
        return;
    for( ; ; )
    {
        fscanf( pf, "%s", token);
        if( feof( pf))
            break;
        counter++;
        printf( "Token %3d: %s \n", counter, token);
    }
    fclose(pf);
}
```

Datentyp für einen Stream

fopen versucht die Datei zu öffnen und gibt den Stream zurück.

Das Programm öffnet seine eigene Quellcodedatei zum Lesen (*r* steht für read)

Die Datei konnte nicht geöffnet werden.

Lies ein Wort aus der Datei.

Wenn nichts mehr gelesen werden konnte, beende die Schleife.

Gib das Wort auf dem Bildschirm aus.

```
Token:   1: #
Token:   2:  include
Token:   3:  <stdio.h>
Token:   4: #
Token:   5: Include
Token:   6: #
Token:   7:  include
Token:   8:  <stdlib.h>
Token:   9: void
...
```

Anwendung von Dateioperationen

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

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

void main()
 {
    char token[100];
    int counter = 0;
    FILE *pf;

    pf = fopen( "Test.c", "r");

    if( pf == 0)
        return;
    for( ; ; )
    {
        fscanf( pf, "%s", token);
        if( feof( pf))
            break;
        counter++;
        printf( "Token %3d: %s \n", counter, token);
    }
    fclose(pf);
}
```

Datentyp für einen Stream

fopen versucht die Datei zu öffnen und gibt den Stream zurück.

Das Programm öffnet seine eigene Quellcodedatei zum Lesen (*r* steht für read)

Die Datei konnte nicht geöffnet werden.

Lies ein Wort aus der Datei.

Wenn nichts mehr gelesen werden konnte, beende die Schleife.

Gib das Wort auf dem Bildschirm aus.

Die Datei wird wieder geschlossen.

```
Token:   1: #
Token:   2:  include
Token:   3:  <stdio.h>
Token:   4: #
Token:   5: Include
Token:   6: #
Token:   7:  include
Token:   8:  <stdlib.h>
Token:   9: void
...
```

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);
}
```

```
Bitte gibt 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.

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 gibt 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 übergebenen Parameter vom Stack holen kann.
- `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
```

```
# 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;
}
```

Funktionen mit variabler Argumentzahl

Hochschule RheinMain

- Funktionen können eine unbestimmte Anzahl Parameter haben.
- Es gibt Makros, mit denen man zur Laufzeit die effektiv übergebenen Parameter vom Stack holen kann.
- `printf` und `scanf` arbeiten nach diesem Prinzip.

Notwendiges Include für Funktionen mit variabler Argumentzahl

```
# 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;
}
```

```
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

Funktionen mit variabler Argumentzahl

Hochschule RheinMain

- Funktionen können eine unbestimmte Anzahl Parameter haben.
- Es gibt Makros, mit denen man zur Laufzeit die effektiv übergebenen Parameter vom Stack holen kann.
- `printf` und `scanf` arbeiten nach diesem Prinzip.

Notwendiges Include für Funktionen mit variabler Argumentzahl

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

```
# 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;
}
```

```
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

Funktionen mit variabler Argumentzahl

Hochschule RheinMain

- Funktionen können eine unbestimmte Anzahl Parameter haben.
- Es gibt Makros, mit denen man zur Laufzeit die effektiv übergebenen Parameter vom Stack holen kann.
- `printf` und `scanf` arbeiten nach diesem Prinzip.

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

```
# 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;
}
```

```
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

Funktionen mit variabler Argumentzahl

Hochschule RheinMain

- Funktionen können eine unbestimmte Anzahl Parameter haben.
- Es gibt Makros, mit denen man zur Laufzeit die effektiv übergebenen Parameter vom Stack holen kann.
- `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

Notwendiges Include für Funktionen mit variabler Argumentzahl

```
# 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;
}
```

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

Funktionen mit variabler Argumentzahl

Hochschule RheinMain

- Funktionen können eine unbestimmte Anzahl Parameter haben.
- Es gibt Makros, mit denen man zur Laufzeit die effektiv übergebenen Parameter vom Stack holen kann.
- `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

Notwendiges Include für Funktionen mit variabler Argumentzahl

```
# 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;
}
```

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

Funktionen mit variabler Argumentzahl

Hochschule RheinMain

- Funktionen können eine unbestimmte Anzahl Parameter haben.
- Es gibt Makros, mit denen man zur Laufzeit die effektiv übergebenen Parameter vom Stack holen kann.
- `printf` und `scanf` arbeiten nach diesem Prinzip.

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

Lesen des nächsten int-Parameters vom Stack*

Schleife über alle unspezifizierten Parameter

```
# 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;
}
```

```
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

Funktionen mit variabler Argumentzahl

Hochschule RheinMain

- Funktionen können eine unbestimmte Anzahl Parameter haben.
- Es gibt Makros, mit denen man zur Laufzeit die effektiv übergebenen Parameter vom Stack holen kann.
- `printf` und `scanf` arbeiten nach diesem Prinzip.

```
# 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

Lesen des nächsten int-Parameters vom Stack*

Schleife über alle unspezifizierten Parameter

Ende der Stackpointeroperaion

```
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

Funktionen mit variabler Argumentzahl

- Anmerkung: 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 ein Macro das zu einem Stackzugriff mit dem Datentyp `int` aufgelöst wird. Die konkrete Definition des Macros finden Sie in `stdarg.h`.

Dynamische Speicherallokierung

- 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.

Dynamische Speicherallokierung

- 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:

```
int *p;
p = (int *)malloc( 123*sizeof( int));
```

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

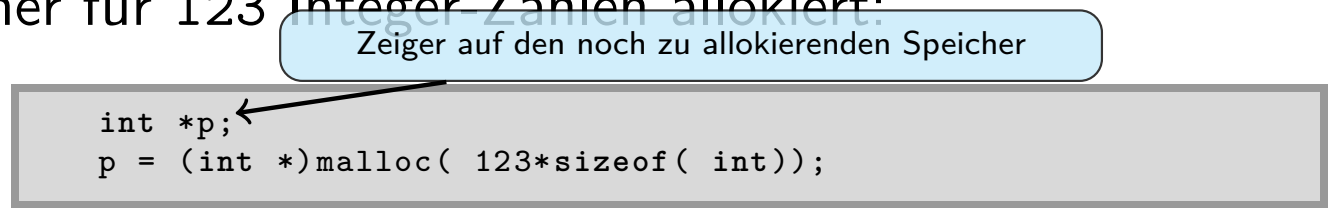
```
int i;

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

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

```
free(p);
```

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:




```
int *p;
p = (int *)malloc( 123*sizeof( int));
```

- 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);
```

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:

```
int *p;
p = (int *)malloc( 123*sizeof( int));
```

Zeiger auf den noch zu allokierenden Speicher

Speicher für 123 int-Zahlen wird benötigt

Zuweisung des allokierten Speichers

Formale Typumwandlung

Anzahl Bytes, die eine int-Zahl im Speicher belegt

- 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);
}
```

```
Wie viele Zahlen: 5
1. Zahl: 1
2. Zahl: 2
3. Zahl: 3
4. Zahl: 4
5. Zahl: 5
5
4
3
2
1
```

- 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 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);
}
```

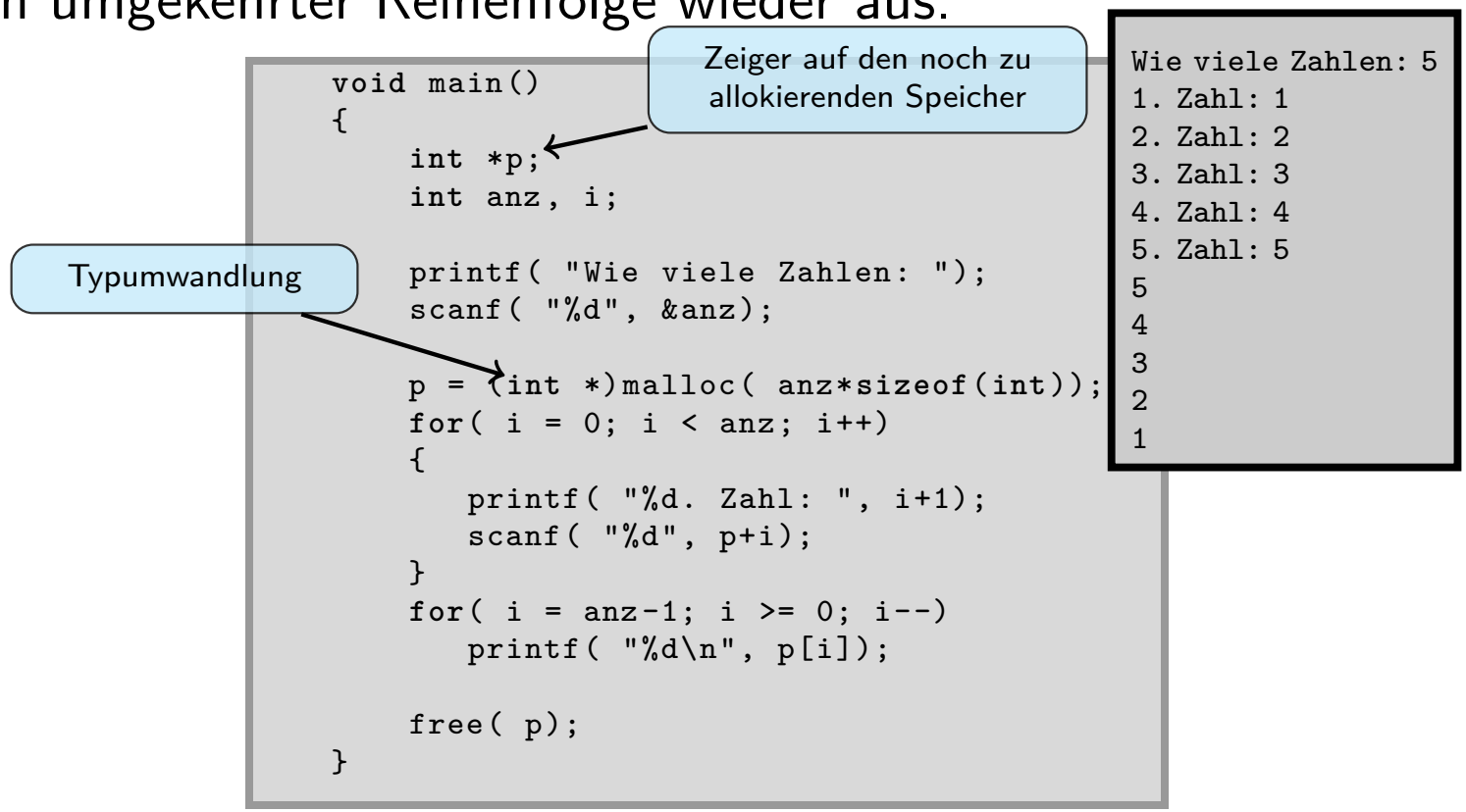
Zeiger auf den noch zu allokierenden Speicher

```
Wie viele Zahlen: 5
1. Zahl: 1
2. Zahl: 2
3. Zahl: 3
4. Zahl: 4
5. Zahl: 5
5
4
3
2
1
```

- 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 1

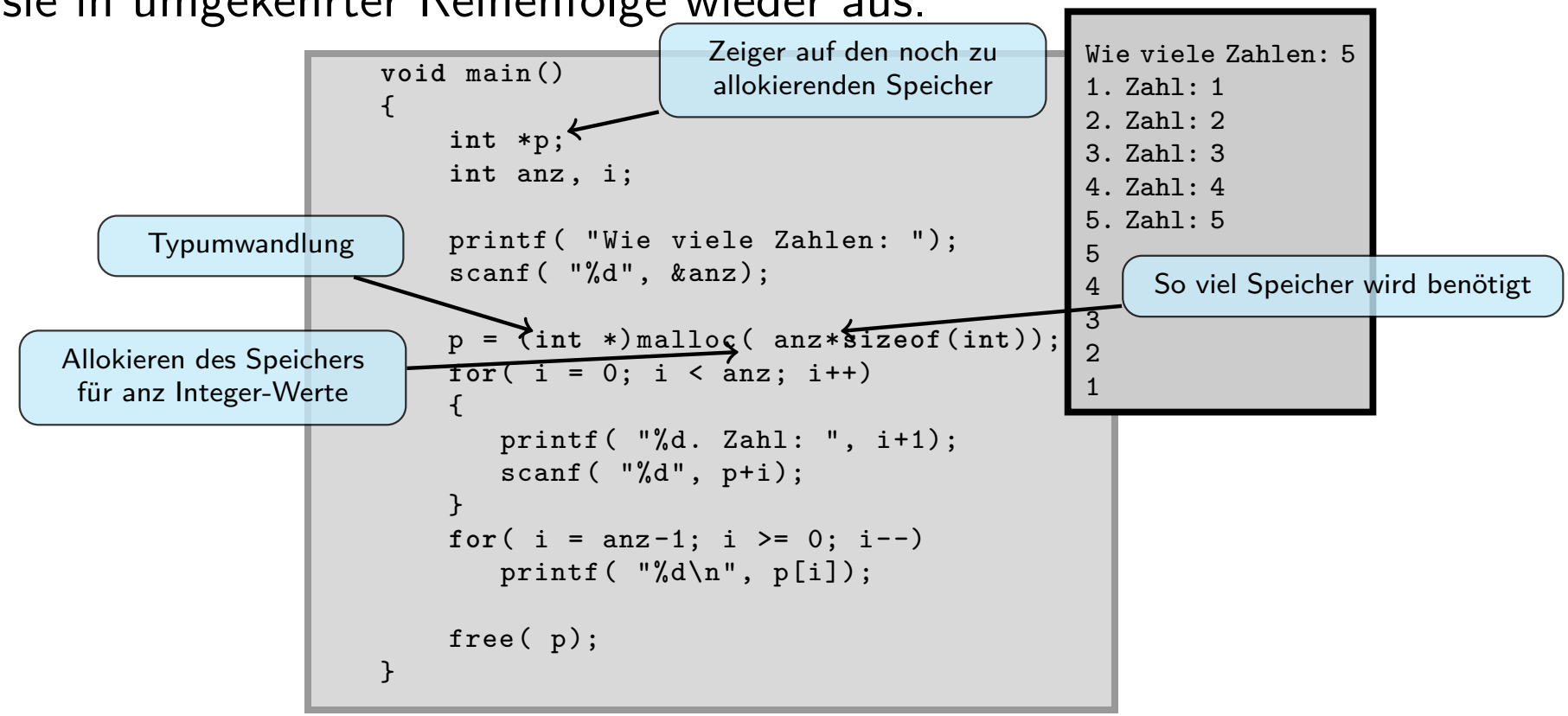
- Lies eine vom Benutzer vorab gewählte Anzahl an Zahlen ein und gib sie in umgekehrter Reihenfolge wieder aus.



- 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 1

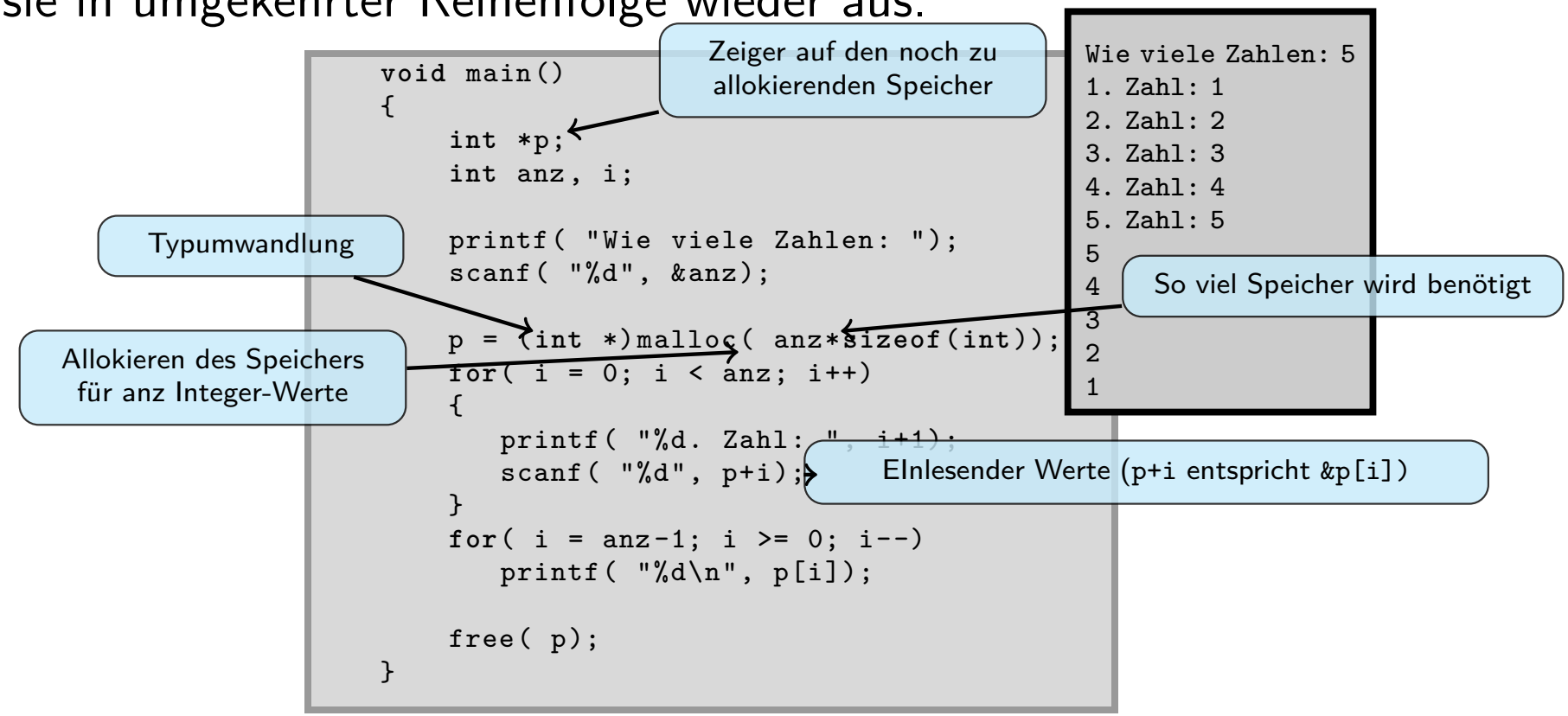
- Lies eine vom Benutzer vorab gewählte Anzahl an Zahlen ein und gib sie in umgekehrter Reihenfolge wieder aus.



- 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 1

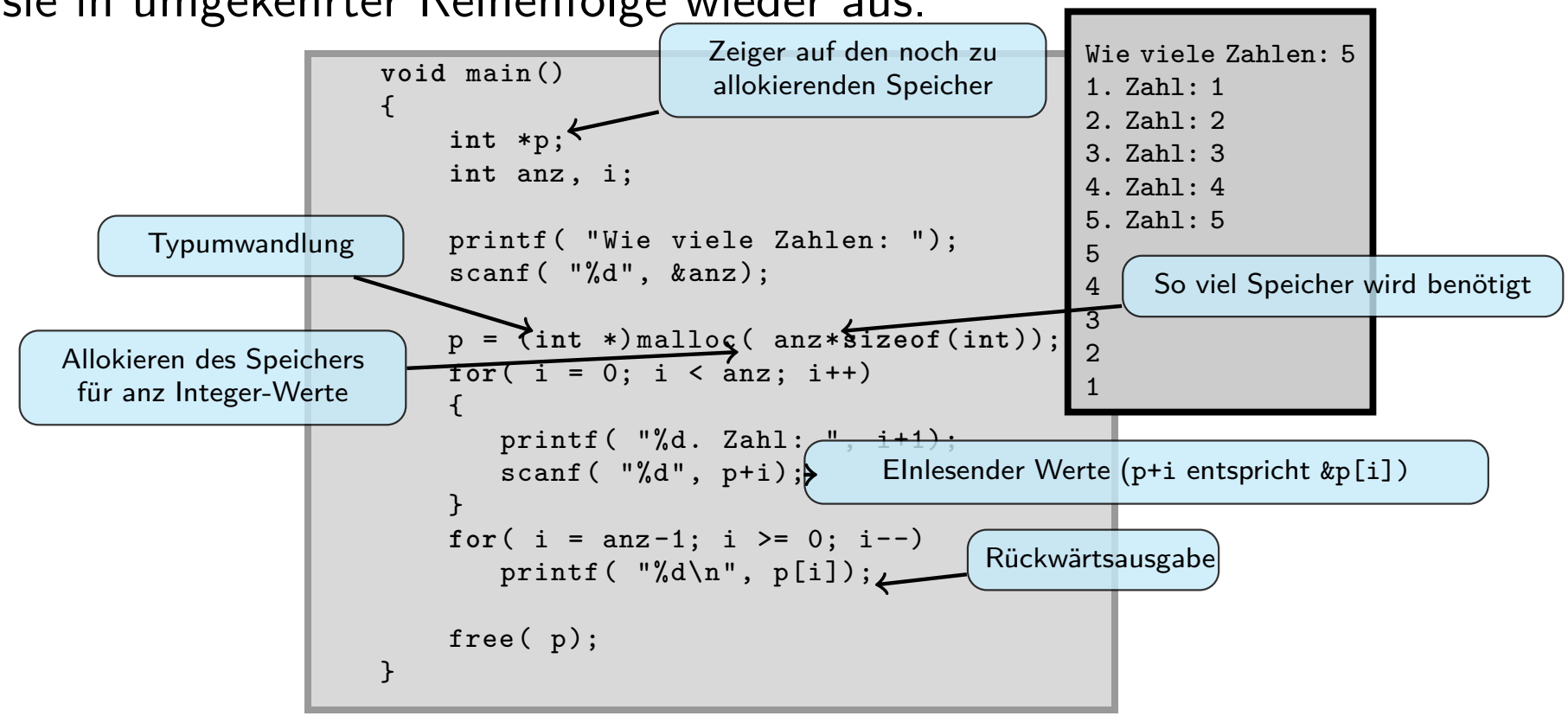
- Lies eine vom Benutzer vorab gewählte Anzahl an Zahlen ein und gib sie in umgekehrter Reihenfolge wieder aus.



- 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 1

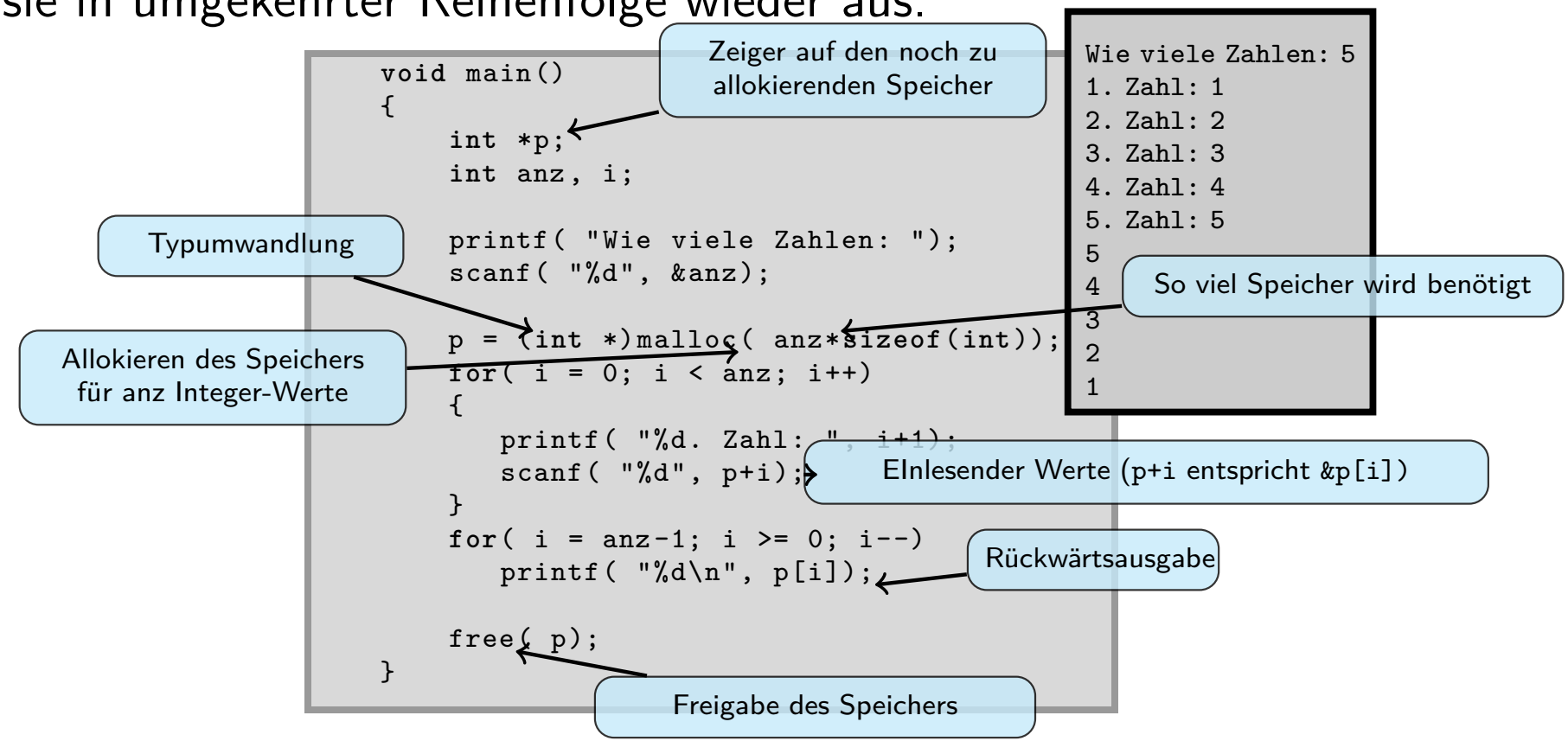
- Lies eine vom Benutzer vorab gewählte Anzahl an Zahlen ein und gib sie in umgekehrter Reihenfolge wieder aus.



- 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 1

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



- 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()

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);
}
```

```
1. Zahl: 1
Array auf 2 Elemente vergroessert
2. Zahl: 2
3. Zahl: 3
Array auf 4 Elemente vergroessert
4. Zahl: 4
5. Zahl: 5
Array auf 6 Elemente vergroessert
6. Zahl: 6
7. Zahl: -1
6
5
4
3
2
1
```

Beispiel mit dynamischer Speicherverwaltung 2

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

```
void main()

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);
}
```

Anfangsgröße und Zuwachs bei Vergrößerung

```
1. Zahl: 1
Array auf 2 Elemente vergroessert
2. Zahl: 2
3. Zahl: 3
Array auf 4 Elemente vergroessert
4. Zahl: 4
5. Zahl: 5
Array auf 6 Elemente vergroessert
6. Zahl: 6
7. Zahl: -1
6
5
4
3
2
1
```

Beispiel mit dynamischer Speicherverwaltung 2

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

```
void main()

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);
}
```

Anfangsgröße und Zuwachs bei Vergrößerung

Noch kein Speicher allokiert

```
1. Zahl: 1
Array auf 2 Elemente vergroessert
2. Zahl: 2
3. Zahl: 3
Array auf 4 Elemente vergroessert
4. Zahl: 4
5. Zahl: 5
Array auf 6 Elemente vergroessert
6. Zahl: 6
7. Zahl: -1
6
5
4
3
2
1
```

Beispiel mit dynamischer Speicherverwaltung 2

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

```
void main()

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);
}
```

Anfangsgröße und Zuwachs bei Vergrößerung

Noch kein Speicher allokiert

Kapazität überschritten

Neue Kapazität berechne und Speicher allokieren bzw. reallokieren

```
1. Zahl: 1
Array auf 2 Elemente vergroessert
2. Zahl: 2
3. Zahl: 3
Array auf 4 Elemente vergroessert
4. Zahl: 4
5. Zahl: 5
Array auf 6 Elemente vergroessert
6. Zahl: 6
7. Zahl: -1
6
5
4
3
2
1
```

Beispiel mit dynamischer Speicherverwaltung 2

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

```
void main()

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);
}
```

Anfangsgröße und Zuwachs bei Vergrößerung

Noch kein Speicher allokiert

Kapazität überschritten

Neue Kapazität berechne und Speicher allokieren bzw. reallokieren

Speichern der Eingabe

```
1. Zahl: 1
Array auf 2 Elemente vergroessert
2. Zahl: 2
3. Zahl: 3
Array auf 4 Elemente vergroessert
4. Zahl: 4
5. Zahl: 5
Array auf 6 Elemente vergroessert
6. Zahl: 6
7. Zahl: -1
6
5
4
3
2
1
```

Beispiel mit dynamischer Speicherverwaltung 2

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

```
void main()

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);
}
```

Anfangsgröße und Zuwachs bei Vergrößerung

Noch kein Speicher allokiert

Kapazität überschritten

Neue Kapazität berechne und Speicher allokieren bzw. reallokieren

Speichern der Eingabe

Rückwärtsausgabe

```
1. Zahl: 1
Array auf 2 Elemente vergroessert
2. Zahl: 2
3. Zahl: 3
Array auf 4 Elemente vergroessert
4. Zahl: 4
5. Zahl: 5
Array auf 6 Elemente vergroessert
6. Zahl: 6
7. Zahl: -1
6
5
4
3
2
1
```

Beispiel mit dynamischer Speicherverwaltung 2

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

```
void main()

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);
}
```

Anfangsgröße und Zuwachs bei Vergrößerung

Noch kein Speicher allokiert

Kapazität überschritten

Neue Kapazität berechne und Speicher allokieren bzw. reallokieren

Speichern der Eingabe

Rückwärtsausgabe

Freigabe des Speichers

```
1. Zahl: 1
Array auf 2 Elemente vergroessert
2. Zahl: 2
3. Zahl: 3
Array auf 4 Elemente vergroessert
4. Zahl: 4
5. Zahl: 5
Array auf 6 Elemente vergroessert
6. Zahl: 6
7. Zahl: -1
6
5
4
3
2
1
```

Beispiel mit dynamischer Speicherverwaltung 2

- Das Programm auf der vorherigen Folie ist voll flexibel, da das Array bei Bedarf mit `realloc` vergrößert wird. Normalerweise arbeitet man natürlich nicht so *kleinschrittig* wie hier gezeigt.

Einige wichtige Speicheroperationen

Befehl	Funktionsbeschreibung
memcpy	Kopiere einen Speicherblock
memmove	Verschiebe einen Speicherblock
memcmp	Vergleiche zwei Speicherblöcke
memchr	Finde ein Zeichen in einem Speicherblock
memset	Initialisiere einen Speicherblock
strlen	Berechne die Länge eines Strings