dbis
Institut für Informatik
Humboldt-Universität zu Berlin

Die Programmiersprache C

4. Dynamische Objekte, Präprozessor, Modularisierung

Vorlesung des Grundstudiums
Prof. Johann-Christoph Freytag, Ph.D.
Institut für Informatik, Humboldt-Universität zu Berlin
SoSe 2018

Funktionen, Nachtrag (1)

dbis
INSTITUT FÜR INFORMATIK
HUMBOLDT-UNIVERSITÄT ZU BERLIN

Basissyntax (Definition): wie bei Java-Methoden

```
void readvectors (vector v1, vector v2) {
        int i;      /* zu Beginn */
        ...         /* dann Anweisungen */
                    /* ab C'99 Typ- u. Variablendeklarationen später möglich */
}
```

Besonderheiten Sichtbarkeitsregeln:

- static void readvectors (v1, v2);
  // nur in Übersetzungseinheit (File) sichtbar

- extern void readvectors (v1, v2);
  // auch nach außen sichtbar, extern ist Standardannahme

Funktionen, Nachtrag (2)

dbis
INSTITUT FÜR INFORMATIK
HUMBOLDT-UNIVERSITÄT ZU BERLIN

Verschachtelung von Funktionen

- wie in Java **nicht** erlaubt, in C sind alle Funktionen global, obwohl Blockkonzept (Gültigkeitsbereiche für Bezeichner) seit Algol-60 bekannt

- **Gründe**:
  - Leichter und effektiver durch Compiler zu verarbeiten (Compilezeit)
  - Verwaltungsaufwand für Funktionsrufe geringer (Laufzeit)

- **Kritik**: methodischer Nachteil
  Programmstruktur entspricht nicht der Problemstruktur

Funktionen, Nachtrag (3)

dbis
INSTITUT FÜR INFORMATIK
HUMBOLDT-UNIVERSITÄT ZU BERLIN

Resultattyp von Funktionen

- erlaubt sind: alle Typen auch strukturierte Typen
  (Werte werden in Kopie nach außen gereicht)

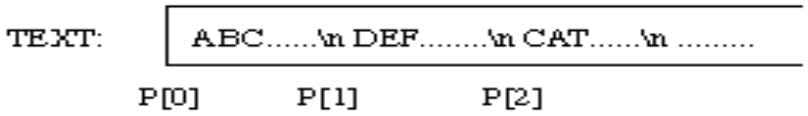
- Vorsicht bei der Rückgabe von Adressen:
  - bei der Rückgabe werden nur „Henkel“ kopiert, nicht aber die Objekte der „Henkel“
  - werden Adressen lokaler Objekte zurückgegeben, ist die weitere Programmausführung nicht definiert

- Rückgabe von Adressen von globalen Objekten oder dynamisch angelegten Objekten auf der Halde ist dagegen der typische Anwendungsfall

Felder von Zeigern

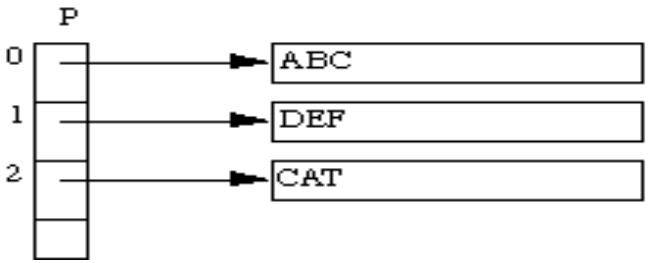
Beispiel: lexikografisches Sortieren der Zeilen eines Textes

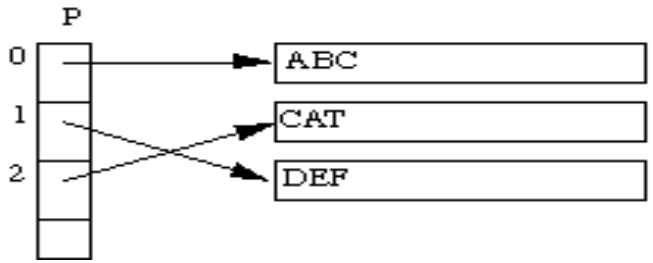
- **Bemerkung**: Text kann nicht mit einer Operation verglichen oder verschoben werden
- Feld von Zeigern ist eine Datenrepräsentation, mit deren Hilfe effizient und elegant mit Textzeilen unterschiedlicher Länge umgegangen werden kann



verhindert:

- **komplexes Speichermanagement**
- **Zusatzaufwand beim Kopieren**





Multidimensionale Felder

- In C: 2D-Felder sind in Wirklichkeit 1D-Felder, bei dem jedes Element wiederum ein Feld ist
- Notation: a[n][m]

  Feldelemente werden zeilenweise gespeichert
- werden 2D-Felder an Funktionen übergeben, so muss stets die Anzahl der Spalten angegeben werden
  - Entweder sind alle Dimensionen bekannt oder nur die erste (d.h. Zeilenzahl) nicht
  - Bei unbekannter Zeilenzahl ist zusätzlicher Parameter erforderlich
  - Grund: C muss “wissen” , wie viele Spalten es gibt, um von einer Zeile zur nächsten springen zu können

Multidimensionale Felder (2)

Beispiel: a[5][35] wird wie folgt übergeben:

- Erste Alternative:

  f(a); void f(int vec[5][35]) {.....}

  f(a, 5); void f(int vec[][35], int dim) {.....}

- Zweite Alternative:

  f(a, 5); void f(int (*a)[35], int dim) {.....}

- Bemerkung: wir brauchen Klammern (*a), da [] eine höhere Präzedenz hat als *

  Warum?

  - int (*a)[35] deklariert einen Zeiger auf ein Feld mit 35 integer-Werten
  - int *a[35] deklariert ein Feld von 35 Zeigern vom Typ int

Multidimensionale Felder (3)

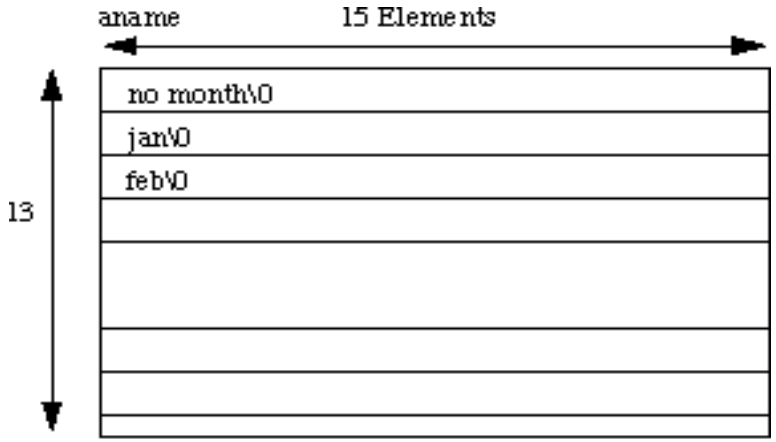
- Beispiel:
  - char *name[10];
  - char aname[10][20];
  - Legal sind: name[3][4] und aname[3][4]

  Achtung:
  - name ist ein Feld mit 10 Zeigerelementen (ohne Speicher für Objekte)
  - aname ist ein 200 elementiges 2D-Feld vom Typ *char* (mit Speicher für Objekte)
  - Zugriff auf Elemente erfolgt im Speicher durch Basisadresse + Spalten-Nr + 20*Zeilen-Nr
- Bemerkung:
  - falls jeder Zeiger in name auf ein 20-elementiges Feld zeigt, werden Speicherzellen für 200 char bereitgestellt (+10 Elemente für die Adressen)
    - char *name[10]= { ... };

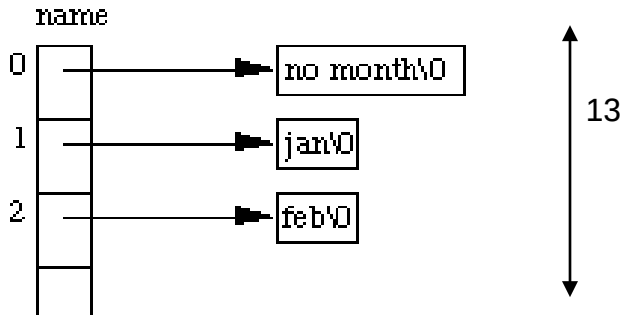
Multidimensionale Felder (4)

- Beispiel:

```
char *name[] = {"no month", "jan", "feb", ... };
char aname[][15] = {"no month", "jan", "feb", ... };
```



Vorteil von name: jeder Zeiger kann auf ein Feld unterschiedlicher Länge zeigen



Zeiger und Strukturen

- Beispiel: Punkt

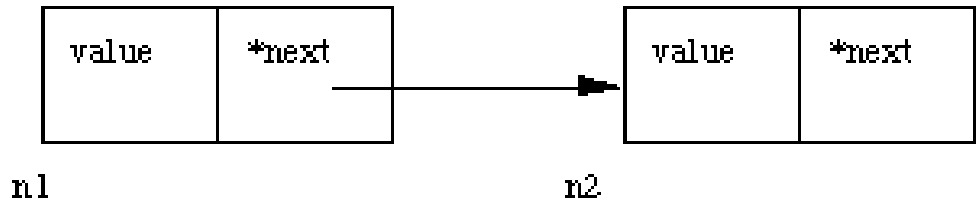
```
struct Punkt {float x, y, z; } p;
struct Punkt *p_ptr;
p_ptr = &p; /* Adresse von p */
```

- Operator -> erlaubt den Zugriff auf Elemente einer Struktur, auf die der Zeiger zeigt:

```
p_ptr->x = 1.0; /* Kurznotation für (*p_ptr).x= 1.0, Klammerung notwendig */
p_ptr->y = p_ptr->y - 3.0;
```

- Beispiel: Verkette Liste

```
struct ELEMENT {  int value; struct ELEMENT *next; };
struct ELEMENT n1, n2; /* ohne struct-Angabe ist ELEMENT kein gültiger Typname */
n1.next = &n2;
```

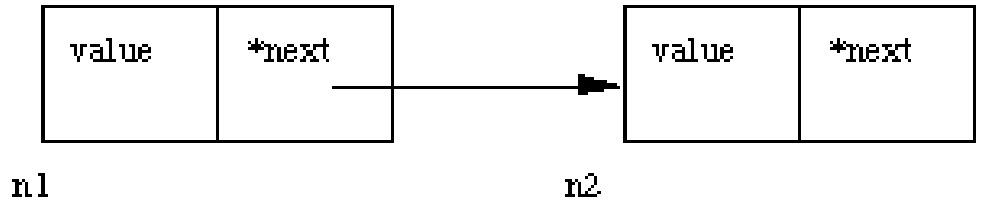


Struktur als Typdefinition

- nochmal: Verkette Liste

```
struct ELEMENT {
   int value;
   struct ELEMENT *next;
} ;

struct ELEMENT n1, n2; /* ELEMENT ist immer noch kein
                          gültiger Typname */
```

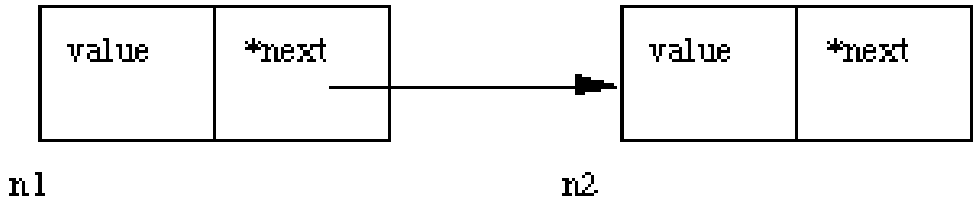


Struktur als Typdefinition

- nochmal: Verkette Liste

```
typedef struct ELEMENT {
   int value;
   struct ELEMENT *next;
} ELEMENT;

ELEMENT n1, n2; /* ELEMENT ist jetzt ohne struct-Angabe ein
                   gültiger Typname */
```



Umgang mit dynamischen Speicher

```
#include <stdlib.h>      /*Systembibliothek: Speicherverwaltung u. mehr */

struct datum {
    int jahr;            /* 4 Bytes */
    char monat[3];       /* 3 Bytes */
    int tag;             /* 4 Bytes */
};
```

Schritte von malloc:

1. sizeof: Größe des Objekts
2. Speicherplatzreservierung
3. Rückgabe eines typlosen Zeigers (void*)
   (zeigt auf Anfang des reserv. Bereichs)
4. entstehender Wert erhält einen Typ
   struct datum * (Typumwandlung)

danach kann Speicherplatz belegt werden

```
struct datum *t1;
t1= (struct datum *) malloc(sizeof(struct datum));
      /* t1 zeigt auf eine nichtinitialisierte Speicherfläche auf der Halde */

      /* wie groß ist die Speicherfläche ? 12!!! Speicherausrichtung */
```

Umgang mit dynamischen Speicher(2)

- Weitere Beispiele

```
int i;
struct PUNKT {float x, y, z};
typedef struct PUNKT p;
sizeof(int), sizeof(i), sizeof(struct PUNKT) oder sizeof(p)
```

  sind alle korrekt

Häufige Probleme mit Zeigern (2)

- Illegale Indirektionen

  Funktion void *malloc()

  stellt Speicher dynamisch zur Laufzeit eines Programms zur Verfügung; gibt Zeiger auf den gewünschten Speicherblock zurück -

  falls erfolgreich, oder sonst einen NULL-Zeiger

  ein gutes C-Programm sollte immer prüfen

```
char *p;
p = (char *) malloc(100);
if ( p == NULL)   { printf("Error: Out of Memory  \n"); exit(1); }
*p = 'y';
```

Häufige Probleme mit Zeigern (3)

- Effekt?

  ```
  p = (char *) malloc(100);
  ```

  - Annahme: es konnte kein Speicher reserviert werden, dann führt

    ```
    *p = 'y';
    ```

    zu einem Laufzeitfehler (SIGSEGV)

Häufige Probleme mit Zeigern (2)

- folgendes Programmfragment enthält Fehler:

```
char *p;
*p = (char *) malloc(100);  /* Anforderung von 100 Bytes */
*p = 'y';
```

Welchen?

Antwort

- keine Angabe von * in *p = (char *) malloc(100);
  - malloc gibt einen Zeiger zurück
  - da p bereits auf einen (nicht legalen!!!) Speicherbereich zeigt, wird in diesem der Rückgabewert von malloc eingetragen
  - p zeigt weiterhin auf die nicht legale Speicheradresse
- korrekter Code:

```
p = (char *) malloc(100);
```

Dynamische Speicheranforderung & dynamische Strukturen

dbis
Institut für Informatik
Humboldt-Universität zu Berlin

```
p = (char *) malloc(100);
```

- Da Zeiger und Felder in C miteinander eng “verwandt” sind, können wir den Speicher wie ein Feld behandeln:

  ```
  p[0] = 'A';
  ```

  oder

  ```
  for(i=0; i<100; ++i) scanf("%c", &p[i]);
  ```

- Weitere Beispiele

  ```
  int i;
  struct PUNKT {float x, y, z};
  typedef struct PUNKT p;
  sizeof(int), sizeof(i), sizeof(struct PUNKT) oder sizeof(p)
  ```

  sind alle korrekt

Dynamische Speicheranforderung & dynamische Strukturen (2)

dbis
INSTITUT FÜR INFORMATIK
HUMBOLDT-UNIVERSITÄT ZU BERLIN

Funktion void free(void* ptr)

- falls ein früher reservierter Speicherblock nicht mehr benötigt wird, sollte dieser “frei”-gegeben werden
- Funktion free leistet dies:
  - Argument ist ein Zeiger auf den Speicherblock
  - Effekt ist die Freigabe des Speichers, auf den der Zeiger verweist
  - leere Anweisung falls ptr ein NULL-Zeiger
- ptr hat aber immer noch die ursprüngliche Adresse
  Was passiert beim erneuten Versuch einer Freigabe ?

Dynamische Speicheranforderung & dynamische Strukturen (3)

dbis
INSTITUT FÜR INFORMATIK
HUMBOLDT-UNIVERSITÄT ZU BERLIN

Funktion calloc

```
void *calloc(size_t   num_elements,  size_t  element_size);
```

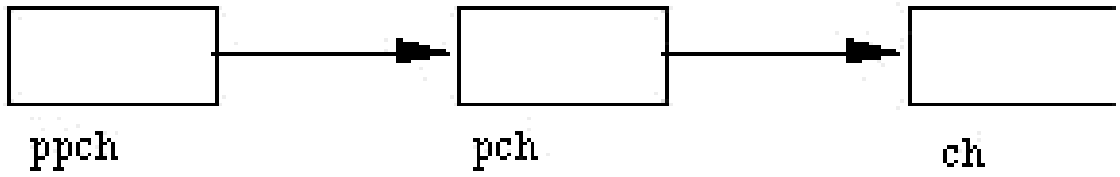
- calloc initialisiert den Speicher mit “Nullen” (im Gegensatz zu malloc !)
- Anwendung von calloc:

```
int *ip;
ip = (int *) calloc(100, sizeof(int));
```

Zeiger auf Zeiger

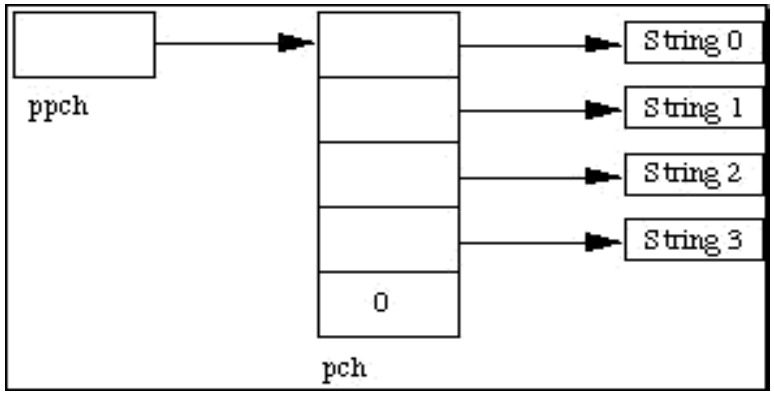
- Beispiel:

```
char ch;          /* ein Zeichen */
char *pch;        /* ein Zeiger auf ein Zeichen */
char **ppch;      /* ein Zeiger auf einen Zeiger, welcher auf ein
                     Zeichen zeigt */
```



Zeiger auf Zeiger (2)

- Zeiger auf mehr als einen String:



- definiert als:

  char *ppch[]

Kommandozeileneingabe

- C erlaubt das Lesen von Eingabeargumenten von der Kommandozeile, die im Programm verarbeitet werden können
  - werden bei Aufruf des Programms nach dem Namen angegeben
  - Beispiel: cc -o prog prog.c
  - Definition von main kann wie folgt erfolgen:

    int main(int argc, char *argv[]) ...

- Main-Funktion hat ihre eigenen Argumente:
  - argc gibt die Anzahl der eingegebenen Argumente an (einschließlich des Programmnamens)
  - argv ist ein Feld von Strings: Jedes Element enthält ein Argument aus der Eingabe (einschließlich des Programmnamens als erstes Feldelement)

Kommandozeileneingabe (2)

- Beispiel:

```
#include <stdio.h>
int main (int argc, char **argv)
    { /* program to print arguments from command line */
      int i;
      printf("argc = %d\n",argc);
      for (i=0; i<argc; ++i)
            printf("argv[%d]: %s\n", i, argv[i]);

   return 0; /* sonst ist der Rückgabewert undefiniert */
}
```

Überblick

- Bit-Operationen
- C-Präprozessor
- Systemaufrufe
- Standardbibliotheken (libraries)

Bitorientierte Operatoren

&	AND
|	OR
^	XOR
~	Einer-Komplement
<<	Shift left
>>	Shift right

- Bemerkung:
  - Nicht zu verwechseln: & und && - & ist bitorientiertes AND, && logisches UND.
  - Ähnlich für | und ||
  - Unärer Operator
  - Die Schiebeoperatoren schieben den linken Operanden um den Wert des rechten Operanden
  - Der rechte Operand muss positiv sein. Die ‘neuen’ Bits werden mit ‘0’ aufgefüllt (***i.e.*** Es gibt **kein** “wrap around”)

Bitorientierte Operatoren (2)

- Beispiel:
  - Falls x = 00000010 (binär) oder 2 (dezimal):
    - x>>=2 : x= 00000000 oder x = 0 (dezimal)
    - x<<=2 : x= 00001000 oder x = 8 (dezimal)
  - Bemerkung:
    - “shift left” ist zur Multiplikation mit 2 äquivalent
    - Ähnlich: “shift right” ist äquivalent zur Division mit 2
    - Die shift Operation ist wesentlich schneller als die Multiplikation (*) oder Division (/) durch 2

C-Präprozessor

- Motivation
  - Substitution von sich häufig wiederholenden Mustern
  - Substitution wird vor der Compilation ausgeführt
  - Sollte Lesbarkeit des Codes verbessern und Code vereinfachen (aber nicht übertreiben !)
- “Direktiven” beginnen immer mit einem ”#”

C-Präprozessor (1)

#define <macro> <replacement name>

- definiert ein Makro: <macro> String soll durch <replacement name> ersetzt werden

- Beispiel:
  - #define begin {
  - #define end }
  - #define max(A, B) ((A) > (B) ? (A):(B))
    - dies definiert keine Funktion, sondern nur Text
    - A and B werden durch “aktuelle” Parameter ersetzt:
    - max(a,3) : max(a,3) ((a) > (3) ? (a):(3))

C-Präprozessor (2)

#undef <name> : macht die Makrodefinition “rückgängig”

#include: fügt eine Datei mit in den Code ein

- **#include** hat zwei mögliche Formen:
  - **#include** <file> oder **#include** "file"
  - <file> gibt dem Compiler an, wo er im System nach den Dateien zu suchen hat.
    Im UNIX-System werden die Dateien normalerweise im Verzeichnis /usr/include gehalten
  - "file" sucht nach der Datei im aktuellen Verzeichnis (in dem das Programm ausgeführt wird)

- Include-Dateien enthalten normalerweise C- Deklarationen und keinen (algorithmischen) C-Code

C-Präprozessor (3)

#if <Conditional> <inclusion>

- **#if** evaluiert einen konstanten Integer-Ausdruck.
- es wird ein **#endif** benötigt, um das Ende zu signalisieren
- else Teil: **#else** and **#elif** -- else if

Weitere Nutzung von **#if** :

#ifdef

-- if defined

#ifndef

-- if not defined

- Sie sind nützlich, um zu überprüfen, ob Makros definiert sind möglicherweise in verschiedenen Programmen oder “include”-Dateien

C-Präprozessor (4)

Weitere Kommandos:

#error text of error message

- Generiert eine entsprechende Fehlermeldung des Compilers

#line number "string"

- informiert den Präprozessor, dass die Zeilennummer der nächsten Zeile number sein soll und in welcher Datei mit Namen string diese Zeile auftritt

Standardbibliotheken

Viele “vorprogrammierte” Funktionen sind verfügbar:

- stdio.h: Standard- Ein-/Ausgabe-Bibliothek
  - Terminal- & Dateifunktionen
  - printf, scanf, sprintf, sscanf, ...
- math.h: mathematische Funktionen
- string.h: Zeichenkettenoperationen
- Bibliothek an Systemaufrufen (BS-abhängig):
  - Prozessfunktionen, “message queues”, “interrupts”, “shared memory”, “threads”, “remote procedure calls”, “mutex”, ...

Modularisierung von C-Programmen

Programm als Menge von Bausteinen (Modulen)

Modul hat zwei Seiten

- technischer Aspekt: Compilationseinheit
- semantischer Aspekt: begründete Teilaufgabe

Beispiel für Modularisierung (abstrakter Datentyp Keller)

- Interface des Kellers
- Implementation des Kellers
- Nutzung des Kellers

File stack.h (Interfaces des Kellers)

```
#define bool short          /* bool gibt es bereits in C'99 */

extern void push (char e); /* speichere e im Keller*/
extern char poptop ();      /* oberstes Kellerelement  als Wert u. im Keller streichen*/
extern bool isempty ();     /* testet, ob Keller leer ist*/
```

File stack.c (Implementation des Kellers)

```
#include <stdio.h>
#include “stack.h“          /* nicht zwingend, aber Funktionen können nun in beliebiger
                                Reihenfolge def. werden */

/* Repraesentation der Daten im Stack */
static int const N=100;
static char a[N];            /* Keller mit maximal 100 Elementen */
static int k=0;              /* Keller enthaelt k Elemente, a bis zur Stelle k-1 gefüllt */

void push (char e) {
    if (k<N) {
        a[k]=e;
        k++
    }
    else printf(“Keller voll\n“);
}
```

File stack.c (Implementation des Kellers)

```
char poptop () {
    if (k>0) return a [--k];
    else return '\0';
}

bool isempty () {
    return (k==0);
}
```

File main.c (Anwendung)

```
#include <stdio.h>
#include “stack.h“                    /* hier zwingend */

int main() {
    char x;
    while ((x=getchar()) != ‘\n‘) push(x);
    while ( ! isempty()) {
            x= poptop();
            putchar(x);
    }
    putchar(‘\n‘);
    return 0;
}
```

Viel Spaß – und bis zum Semesterbeginn...