Advanced Programming in C

Pointer und Listen

Institut für Numerische Simulation
Rheinische Friedrich-Wilhelms-Universität Bonn

Oktober 2013

Überblick

1. Variablen vs. Pointer
   - Statischer und dynamischer Speicher
2. Structs
   - Eigene Datentypen aus Kombination anderer Datentypen
3. Listen
   - Verkettung mehrerer Elemente

Variablen vs. Pointer

– Statischer und dynamischer Speicher

Variablen

Eigenschaften

- Haben einen **bestimmten Datentyp** (**int**, **float**, ...).
- **Speicherplatzbedarf** der Variable ist **bekannt**.
- Müssen **im Quellcode deklariert** werden.
- **Zu jeder Zeit bekannt** wie viel und welcher **Speicherplatz** verwendet wird.

Pointer

Eigenschaften

- **Keine** direkte **Speicherzelle**.
- Lediglich **Adresse einer Speicherzelle**.
- In der **Speicherzelle können Daten** stehen.
- Es können **beliebig viele Pointer** auf **eine Speicherzelle** zeigen.
- Es ist **nicht bekannt**, wie viel **Speicherplatz zu einem Zeitpunkt** belegt ist.

Beispiel: Addition mit Variablen und Pointern

Listing 1: Addition mit Variablen

```
#include <stdio.h>

int
main (void)
{

  int var1 = 21;
  int var2 = 21;

  printf ("%d \n", var1 + var2);

  return 0;
}
```

Listing 2: Addition mit Pointern

```
#include <stdio.h>

int
main (void)
{

  int var1 = 21;
  int* var2 = &var1;

  printf ("%d \n", var1 + *var2);

  return 0;
}
```

Zusammenfassung: Pointer

- Anlegen eines Pointers:
  double $*$ pointer;

- Zuweisen der Daten in der Speicherzelle (&-Operator):
  double value $=$ 42;
  pointer $=$ &value;

- Zugriff auf Daten in der Speicherzelle ($*$-Operator):
  double value $=$ $*$pointer;

Beispiel: Statische und dynamische Arrays

Listing 3: Statisches Array

```
#include <stdio.h>
int
main (void)
{
  int a [] = {1,90,20};

  int sum = 0;
  for(int i = 0 ; i < 3 ; ++i)
    sum += a[i];

  return 0;
}
```

Listing 4: Dynamsiches Array

```
#include <stdio.h>
#include <stdlib.h>
int
main (void)
{
  int* a;
  a = (int*)malloc(3 * sizeof(int));
  a[0] = 1; a[1] = 90; a[2] = 20;

  int sum = 0;
  for(int i = 0 ; i < 3 ; ++i)
    sum += a[i];
  free(a);
  return 0;
}
```

Dokumentation: malloc und free

malloc(3) - Linux manual page - Chromi

malloc(3) - Linux manual p

man7.org/linux/man-pages/man3/malloc.3.html

man7.org > Linux > man-pages

NAME | SYNOPSIS | DESCRIPTION | RETURN VALUE | CONFORMING TO | NOTES | SEE ALSO | COLOPHON

MALLOC(3) Linux Programmer's Manual MALLOC(3)

NAME top

malloc, free, calloc, realloc - allocate and free dynamic memory

SYNOPSIS top

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

DESCRIPTION top

The **malloc**() function allocates *size* bytes and returns a pointer to the allocated memory. *The memory is not initialized*. If *size* is 0, then **malloc**() returns either NULL, or a unique pointer value that can later be successfully passed to **free**().

The **free**() function frees the memory space pointed to by *ptr*, which must have been returned by a previous call to **malloc**(), **calloc**() or **realloc**(). Otherwise, or if *free(ptr)* has already been called before, undefined behavior occurs. If *ptr* is NULL, no operation is performed.

Reine Dokumentation: `http://man7.org/linux/man-pages/man3/malloc.3.html`

Dokumentation mit Beispielen: `http://www.cplusplus.com/reference/cstdlib/malloc/`

Zusammenfassung: Dynamisches Array

- Anlegen eines Pointers:
  double $*$ array;
- Allozieren von Speicher:
  array $=$ (**double**$*$)malloc(n $*$ **sizeof**(**double**));
- Zugriff auf Daten in den Speicherzellen:
  array[0]$=$ 1.5;
  double value $=$ array [0];
- Freigeben von alloziertem Speicher:
  free (array)

Was passiert, wenn der Aufruf von free() vergessen wird?

Listing 5: Codeauszug

```
int* a = (int*)malloc(3 * sizeof(int));
//free(a);
```

Listing 6: Analyse mit valgrind

```
diehl@ossus $ valgrind ./run
==11846== HEAP SUMMARY:
==11846== in use at exit: 12 bytes in 1 blocks
==11846== total heap usage: 1 allocs, 0 frees, 12 bytes allocated
==11846==
==11846== LEAK SUMMARY:
==11846== definitely lost: 12 bytes in 1 blocks
```

Structs

– Eigene Datentypen aus Kombination anderer Datentypen

Datentyp: **struct**

Deklaration

- **struct** name {var1 , ... , varN};

Beispiel

- **struct** student {**int** fachsemester ; **char** name[25]};

Initialisierung

- **struct** name = {var1 , ... , varN};

Beispiel

- **struct** student = { 1, 'Karl Coder' };

Beispiel

Listing 7: Länge zwischen zwei Punkten

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

int
main (void)
{
  struct point {double x ; double y; } p1 = {0.0 ,0.0};
  struct point p2 = {0.0,1.5};

  double length = sqrt(pow((p1.x-p2.x),2.0) + pow(p1.y-p2.y,2.0));

  printf("Length = %f \n" , length);

  return 0;
}
```

Listen

– Verkettung mehrerer Elemente

Was sind Listen?

Allgemein:

Listen sind quasi „**unbegrenzte Arrays**“.

Sie sind eine **fundamentale Idee der Informatik**.

Die jeweils benötigten **Operationen** sind **entscheidend für** das **Verhalten** / die **Semantik** der Liste, z.B.

- LIFO = Last-In-First-Out: Stack / Keller
- FIFO = First-In-First-Out: Queue / Warteschlange
- sortiert

...

Vor-/Nachteile von Listen

Vorteile

- **Anzahl** der zu speichernden Elemente muss vorher **nicht bekannt** sein bzw. **darf sogar schwanken**.
- **Zusammenhang** der Elemente / Reihenfolge **kann leicht modifiziert werden**.

Nachteile

- **Zugriff** auf ein bestimmtes Element **relativ langsam**.
  - ★ Arrays $O(1)$
  - ★ Listen $O(n)$
- Speicherbedarf = Anzahl der Elemente + Verwaltung

Grundstruktur/Datentyp (einfachverkettete Liste)

Zunächst die **konkrete Deklaration** eines **Listen-Elements**:

- **struct** element
  {
    int value; // *Wert des Elements*
    struct element *next; // *Zeiger auf Nachfolger*
  };

Jedes Element enthält einen **Wert** sowie einen Zeiger auf das jeweilige Nachfolger-Element. Das letzte Listenelement enthält den NULL-Pointer als Nachfolger-Zeiger.
Die Liste wird über einen **Zeiger auf das erste Element**, den sog. „**Anker**“ angesprochen:

- **struct** element *anchor = NULL; // *leere Liste*

Operationen: **leere Liste**

Eine **Liste auf „Leerheit“ überprüfen**.

- **int** isEmpty (**struct** element *l)
  {
  int empty = 0;
  if (l == NULL)
  empty = 1;
  return empty;
  }

- **int** test = isEmpty(anchor);

Listenoperationen und -Durchläufe

- Für manche Listenoperationen werden sog. **Listendurchläufe** benötigt.
- Listendurchläufe sind **rekursiv** (oft eleganter) und **iterativ** (oft leichter verständlich) möglich.
- In beiden Fällen ist der Listendurchlauf für den **relativ langsamen** Zugriff in der Größenordnung $O(n)$ verantwortlich.

Operationen: **einfügen**

Element hinten anhängen.

- **void** append (**struct** element **l, **int** value) {
  struct element *tmp = *l;
  struct element *new =
  (**struct** element *) malloc (**sizeof** (**struct** element));
  new->value = value; // *Wert setzen*
  new->next = NULL; // *neues letztes Listenelement*
  // *new ans Listenende anhaengen*
  if (tmp == NULL)
  *l = new;
  else
  {
  while (tmp->next != NULL)
  tmp = tmp->next;
  tmp->next = new;
  }
  }

Operationen: **einfügen** (cont. 1)

Hinweis: Setzt man Anker für das **erste** („Head“) und das **letzte** Element („Tail“), kann auch beim hinten Einfügen auf den Listendurchlauf verzichtet werden.

Problem: Nach wie vor mehrfache Vorkommen von Item möglich!

Operationen: **einfügen** (cont. 2)

Nach definiertem Vergleich einfügen.

- Vergleich mit Schlüssel.
  - **struct** element
    {
      int key; // *Schluessel*
      double item; // *Wert*
      struct element *next; // *Nachfolger*
    };

Hinweis: Ist Key eindeutig, dann kommt die Kombination aus Key und Item genau einmal vor.
Problem: Schlüsselverwaltung nötig (Datenbank) und nach wie vor mehrfache Vorkommen von Item möglich!

Operationen: **erstes bzw. letztes Element**

Rückgabe des ersten bzw. letzten Elements
$\Rightarrow$ mit „**Head**“ und „**Tail**“ kein Listendurchlauf nötig.
Varianten:

- Item zurückgeben.
  - **double** elem_Item(**struct** element *l);
- Pointer auf element zurückgeben.
  - **struct** element *elem_Pointer(**struct** element *l);

Operationen: **suchen**

Um ein Element zu **suchen** wird wieder ein **Listendurchlauf** benötigt.
Varianten:

- Item „ist enthalten“/„ist nicht enthalten“ zurückgeben.
- Item selbst zurückgeben.
- Pointer auf **struct** element zurückgeben.

Hinweis: Führt man beim Einfügen zunächst eine Suche durch und fügt Item nur dann ein, wenn „ist nicht enthalten“ zurückgegeben wird, dann kommt Item immer genau einmal vor. (Menge)

Problem: Vergleich mit generischem Item nötig!

Operationen: **suchen** (cont.)

Rekursiv suchen mit Rückgabe von „ist enthalten“/„ist nicht enthalten“.

- **int** contains_Rek(**struct** element *l, **int** i)
  {
  if (isEmpty(l))
  return(1); // *false*
  else if (l->item == i)
  return(0); // *true*
  else
  return(contains_Rek(l->next, i);
  }

Operationen: **löschen**

Vorne/Hinten löschen $\Rightarrow$ kein Listendurchlauf nötig.

Varianten:

- Ohne Rückgabe..
  - **void** delete (**struct** element *l);
- Item zurückgeben.
  - **double** delete (**struct** element *l);
- Pointer auf **struct** element zurückgeben.
  - **struct** element *delete (**struct** element *l);

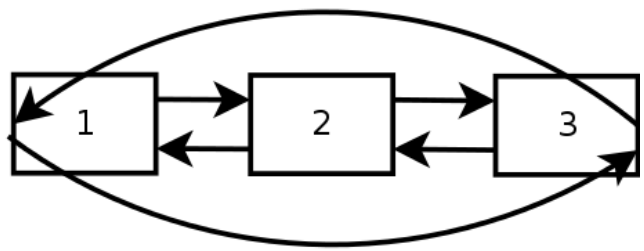
Nach definiertem Vergleich löschen
$\Rightarrow$ suchen, d.h. Übergabe von Item und Listendurchlauf nötig.
Problem: Ggf. löschen aller Vorkommen!

Weitere Listentypen

- doppeltverkettete Liste (doubly-linked list)

```
struct node
{
int value;
node *forward;
node *backward;
};
```

- Doppeltverkettete zyklische Liste



Weitere Listentypen (2)

- Einfachverkettete Kammliste

```
struct node
{
node2 *value;
node *forward;
};
```

- Einfachverkettete Kammliste