ModProg 15-16, Vorl. 8

Richard Grzibovski

Dec. 9, 2015

Übersicht

1. Zeiger: Zusammenfassung
2. void-Zeiger
3. Der Operator / die Funktion `sizeof`
4. Dynamische Felder
   - Speicherreservierung mit `malloc` und `calloc`
   - Wiederanforderung von Speicher mit `realloc`
   - Speicherfreigabe mit `free`
5. Das Makro `assert`
6. Implementierung von Matritzen
   - mittels Indextransformation
   - adressierte 1D-Implementierung

Zeiger: Zusammenfassung

```
1  float var=1.2, *zeiger;
2  zeiger=&var;
3  printf("10.0 + var= %le\n", 10.0 + (*zeiger));
```

- Zeigervariablen sind Variablen, deren Inhalt die Adresse eines Speicherbereiches (einer Variablen) ist.
- Die Adresse einer Variablen bekommt man mit Hilfe des unären Adressoperators &.
- Wenn die Addresse (Zeiger) bekannt ist, kriegt man die Variable mit Hilfe der unären Inhaltsoperator *.
- Der Zeigerwert `NULL` ist die Addresse von nicht existierenden Speicherbereich.

Zeiger: Felder

```
1  float feld[3]={1.1, 2.2, 3.3}, *zeiger;
2  zeiger=feld;
3  printf("10.0+var= %le\n", 10.0 + (*zeiger)); //-> 11.1
4  zeiger++;
5  printf("10.0+var= %le\n", 10.0 + (*zeiger)); //-> 12.2
```

- Feldvariablen zeigen auf den ersten Feldeintrag.
- Feldeinträge sind sequentiell gespeichert.

Zeiger: Zeigerarithmetik

```
1  float feld[3]={1.1, 2.2, 3.3};
2  float *z,*z1,*z2;
3   z = feld + 1;
4  z1 = z - 1;
5  z2 = z + 1;
6  printf("*z1= %le\n", (*z1)); //-> 1.1
7  printf("*z = %le\n", (*z )); //-> 2.2
8  printf("*z2= %le\n", (*z2)); //-> 3.3
9  printf("z2 - z1= %ld\n", z2 - z1); //-> 2
```

Ausdrück	Ergebniss	Bedeutung
Zeiger + Zahl	Zeiger	verschobene Addresse
Zeiger - Zahl	Zeiger	verschobene Addresse
Zeiger - Zeiger	Zahl	wie viele Variablen liegen zwischen
Zeiger + Zeiger	illigal	–

void*: Pointer, der auf beliebigen Datentyp zeigen kann

```
1  char  ch='a', *zch;
2  float fl=1.0, *zfl;
3  int     i=321, *zi ;
4  void *zgr;
5  zch=&ch; zfl=&fl;  zi=&i; // OK
6  zfl=&ch; // Fehler
7  zgr=&ch; zgr=&fl; zgr=&i; // OK!
8  zgr=zch; zgr=zfl; zgr=zi; // OK!
```

- `void*` $\neq$ "Zeiger auf nichts" (!!!)
  sondern
  `void*` == "Pointer, der auf beliebigen Datentyp zeigen kann"
- Vor der Anwendung von Inhaltsoperator auf void-Zeiger eine explizite Typumwandlung ist notwendig.
  `i = *((int*) zgr) + 2015;`
- Vorsicht: wie immer führt der Zuweisungsoperator eine implizite Typumwandlung durch:
  `zch=zgr; zi=igr; zfl=zgr; //OK`

void*: Pointer, der auf beliebigen Datentyp zeigen kann

```
 1  float fl=1.0,*zfl;
 2  int     i=321;
 3  void *zgr;
 4  zgr=&fl;
 5  printf("%f\n", *((float*) zgr) ); // expl. Typumw.
 6  zfl=zgr;                          // impl. Typumw.
 7  printf("%f\n", *zfl );
 8  zgr=&i;
 9  *((int*) zgr) = 1 + 2 + 3 + 4;    // expl. Typumw.
10  printf("%i\n", *((int*) zgr) );   // expl. Typumw.
11  printf("%i\n", i );
```

Der Operator / die Funktion `sizeof`

Um den Quelltext auf verschiedenen Architekturen (mit evtl. unterschiedlichen Größen von `int`, `long`, `double` etc.) nicht ändern zu müssen, existiert die Funktion `sizeof`.

```
size_t sizeof(Datentyp);
```

- `sizeof` liefert die Größe von *Datentyp* in Bytes zurück.
- Der Rückgabetyp `size_t` ist ganzzahlig, vorzeichenlos (`unsigned long int` oder `unsigned int` Architekturabhängig).
- Begriffe `size_t` und `sizeof` sind in `stdlib.h` deklariert.

Es existiert zudem eine Operatorform von `sizeof`

```
sizeof ( Datenobjekt )
```

- In der Variante als unärer Operator kann das Datenobjekt z.B. eine konkrete Variable oder ein statisches Feld sein.
- Der Ausdruck besitzt als Wert die Größe des Operanden in Bytes.

sizeof: Beispiel

```
1  float  f,ff[]={1.0,2.0,3.0,4.0,5.0},*zf=ff;
2  double d,fd[]={1.0,2.0,3.0,4.0,5.0},*zd=fd;
3  printf("%i %i %i %i\n",
4     sizeof( float),sizeof(f),sizeof(zf),sizeof(ff));
5  printf("%i %i %i %i\n",
6     sizeof(double),sizeof(d),sizeof(zd),sizeof(fd));
```

Ergibt;

```
4 4 8 20
8 8 8 40
```

Speicheranforderung mit `malloc`

```
#include<stdlib.h>
 ...  ...  ...
void* malloc(size_t groesse);
```

fordert vom Betriebssystem einen Speicherbereich von `groesse` Bytes an.

- Ist diese Anfrage erfolgreich, so gibt die Funktion einen Zeiger vom Typ `void*` auf diesen Speicherbereich zurück.
- Schlägt die Anforderung fehl, so wird der vordefinierte Zeigerwert `NULL` zurückgeliefert. Man sollte bei Speicheranforderungen das zurückgelieferte Resultat immer auf den Wert `NULL` hin uberprüfen.
- Der reservierte Speicherbereich ist mit zufälligen Werten gefüllt.

Speicheranforderung mit `calloc`

```
#include<stdlib.h>
 ...   ...   ...
void* calloc(size_t anzahl, size_t groesse);
```

- verhält sich im Prinzip wie `malloc`, beim Aufruf wird aber nicht die Gesamtgröße des Speicherbereichs angegeben, sondern die Anzahl der Datenobjekte und die Größe eines einzelnen Datenobjekts getrennt.
- Im Gegensatz zu `malloc` wird der Speicherbereich mit Nullen initialisiert.

Speicherfreigabe mit free

```
#include<stdlib.h>
... ... ...
void free(void* zeiger);
```

- Die Variable zeiger muss dabei auf einen durch `malloc` oder `calloc` erhaltenen Speicherbereich zeigen. Ist zeiger der NULL-Zeiger, so kehrt `free` umgehend zum aufrufenden Programm zurück.
- Das System kann danach wieder uber den Speicher verfügen. Ausdrücke wie `zeiger[0]` oder `*zeiger` sind jetzt nicht mehr definiert und führen in der Regel zu einem Absturz des Programms.
- Zu jedem `malloc` bzw. `calloc` sollte eine `free`-Anweisung existieren, um ökonomisch mit dem Speicher umzugehen und so genannte Speicherleichen zu vermeiden.

Wiederanforderung von Speicher mit realloc

```
#include<stdlib.h>
 ...  ...  ...
void* realloc(void* zeiger, size_t groesse);
```

ändert die Größe des Speicherbereichs, auf den zeiger zeigt, auf groesse Bytes.

- Der zurückgelieferte Zeiger zeigt auf den neu angeforderten Speicherbereich.
- Wenn fehlgeschlagen, liefert den NULL-Zeiger zurück.
- Bis zum Minimum aus alter und neuer Speicherbereichsgröße bleibt der Speicherinhalt unverändert. Neu hinzukommender Speicher wird nicht initialisiert.
- Ist zeiger der NULL-Zeiger, so ist die realloc-Anweisung äquivalent zur malloc-Anweisung.
- Ansonsten muss zeiger durch eine vorangegangene malloc- bzw. calloc- Anweisung erzeugt worden sein.
- Ist groesse gleich 0, so ist die realloc-Anweisung äquivalent zur free-Anweisung.

Zusicherung mit assert

- assert.h definiert Präprozessor-Makro

```
#include<assert.h>
... ... ... ...
assert( Zusicherung );
```

- Falls Ausdruck *Zusicherung* falsch ist, Abbruch mit Meldung:
  `Assertion failed:` *Zusicherung*`, file` *Datei*`, line` *nnn*
- Falls bei Übersetzung das Makro `NDEBUG` definiert ist, verschwinden die `assert`-Tests aus dem Programm:
  Compilierung während der Testphase mit

```
gcc -g program.c
```

  und für Produktionsversion mit

```
gcc -DNDEBUG program.c
```

```c
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>
int main(void)
{
    int anz, k;
    double *vec;
    printf("Dimension: ");
    scanf("%i", &anz);
    assert(anz>=0);
    vec = malloc(anz * sizeof(double));
    if(vec==NULL) {
        printf("Speicheranforderung gescheitert!\n");
        return -1;
    }
    for(k=0;k<anz;k++) vec[k]=0.0; // initialisierung
    ... ...
    free(vec);
  return 0;
}
```

```c
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>
int main(void)
{
    int anz, k;
    double *vec;
    printf("Dimension: ");
    scanf("%i", &anz);
    assert(anz>=0);
    vec = calloc(anz, sizeof(double));
    assert(vec!=NULL);
    ... ...
    free(vec);
  return 0;
}
```

```c
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>
int main(void) {
    int anz, k;
    double *vec,a;
    anz=0;
    vec=NULL;
    printf("Geben Sie ein Vektor ein\n");
    while(scanf("%lf", &a)==1) {
        anz++;
        vec = realloc(vec,anz*sizeof(double));
        assert(vec!=NULL);
        vec[anz-1] = a;
    }
    printf("Dimension: %d\n",anz);
    for(k=0;k<anz;k++)
        printf("vec[%8d]= % le\n",k,vec[k]);
    free(vec);
   return 0;
}
```

Bemerkungen zum `malloc`, `realloc`

- Der Rückgabewert von `malloc`, `calloc`, `realloc` wird implizit umwandelt beim Zuweisung. Mansche (alte) Compiler von C fordern explizite Typumwandlungen wie

  ```
  vec=(double*) malloc(200*sizeof(double));
  ```

- Speicheranforderungen und Wiederanforderungen sind "teuer". Man müss die Anzahl von entsprechende Aufrufe minimieren.

Matrizen und ihre Implementierung

Mit $\mathbb{R}^{m\times n}$ bezeichnen wir den Raum der $(m \times n)$-Matrizen mit reellen Einträgen. Seine Elemente sind von der Form

$$A = (a_{ij})_{i=1,j=1}^{m,n} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1\,n-1} & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2\,n-1} & a_{2n} \\ a_{31} & & \dots & & a_{3n} \\ \vdots & & \dots & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{m\,n-1} & a_{mn} \end{pmatrix}$$

Die folgende Operationen mit Matrizen sind wichtig:

- Multiplikation mit Skalar $\alpha \in \mathbb{R}$, $A \in \mathbb{R}^{m\times n}$: $F = \alpha A \in \mathbb{R}^{m\times n}$
- Addition $A, B \in \mathbb{R}^{m\times n}$: $F = B + A = A + B \in \mathbb{R}^{m\times n}$
- Multiplikation $A \in \mathbb{R}^{m\times k}$, $B \in \mathbb{R}^{k\times n}$: $F = AB \in \mathbb{R}^{m\times n}$
  Der Eintrag f_{ij} in der Ergebnismatrix ist das euklidische Skalarprodukt aus der i-ten Zeile von A und der j-ten Spalte von B.

Implementierung von Matrizen mittels Indextransformation

Die Grundidee, ist, die Matrix $A \in \mathbb{R}^{m \times n}$ Zeile für Zeile hintereinander abzulegen.
Beginnt man die Indizierung wie in C ublich mit 0, so legt man

$$a_{0,0} \ldots a_{0,n-1},\ a_{1,0}, \ldots a_{1,n-1}, \ldots\ldots a_{m-1,0}, \ldots a_{m-1,n-1}$$

im Speicher nacheinander ab.
Der Index des eindimensionalen Feldes ist

$$\mathrm{idx} = \Phi(i, j),$$

wobei

$$\Phi : \{0, \ldots, m-1\} \times \{0, ..., n-1\} \mapsto \{0, ..., mn-1\},$$
$$\Phi(i, j) = in + j.$$

Es handelt sich um eine Bijektion, die durch ganzzahlige Division mit Rest,

$$\Phi^{-1}(idx) = (idx \div n, idx \bmod n),$$

invertiert werden kann. Dabei steht $\div$ für die ganzzahlige Division.

Implementierung von Matrizen mittels Indextransformation

Die Vorteile dieser Methode sind:

- Einfache Implementierung durch eindimensionale Felder, d.h. ein einfacher `malloc`- oder `calloc`-Aufruf genügt,
- einfache zeilenweise Ein- und Ausgabe,
- Matrix-Vektor-Produkt sehr einfach zu realisieren.
- Speicherfreigabe einfach durch Anwendung von `free`.

Andere Operation wie Transposition von Matrizen (Vertauschen von Zeilen und Spalten) und das Matrizenprodukt erfordern jedoch etwas Konzentration bei der Implementierung (siehe Übungen).

```
1  double *matr;
2  int m=4, n=15;
3  matr=calloc(m*n,sizeof(double));
4  ...
5  matr[2*n+4] = -10.0; // 5-te Eintrag in 3-te Zeile
6  ...
7  free(matr);
```

Spaltenweise Indextransformation

Dies ist sozusagen die "transponierte Variante" der Indextransformation Φ. Wir legen die *Matrixspalten* nacheinander in einem eindimensionalen Feld ab, d.h. erst

`A[0][0], A[1][0], ..., A[m-1][0],`

dann

`A[0][1], A[1][1], ..., A[m-1][1],`

usw. bis

`A[0][n-1], A[1][n-1], ..., A[m-1][n-1].`

Die Indextransformation für diese Speichervariante lautet

$$\mathrm{idx} = \Psi(i,j) = jm + i,\ i = 0,\ldots,m-1,\ j = 0,\ldots,n-1.$$

Auch sie ist durch ganzzahlige Division mit Rest leicht zu invertieren, also eine Bijektion.

Adressierte 1D-Implementierung

```
1  #include<stdio.h>
2  int main(void)
3  {
4        double *matrix[2];
5        double matrixvals[] = {2, 4, 8, 7, 5, 6};
6        int i, j ;
7        matrix[0] = matrixvals ;
8        matrix[1] = matrixvals + 3;
9        for(i=0;i<2;i++) {
10             for(j=0;j<3;j++)
11                   printf (" % le ", matrix[i][j]);
12             printf("\n");
13       }
14       return 0;
15 }
```

Ergibt:

```
2 4 8
7 5 6
```

Adressierte 1D-Implementierung

1. Wir reservieren den Speicher für $m \times n$ Matrixeinträge mit `malloc` oder `calloc`: `matr=malloc(m*n*sizeof(double));`
2. Um auf die Matrixinträge a_{ij} als `a[i][j]` zugreifen zu können, brauchen wir ein Feld `a` mit m Zeigern. Deswegen deklarieren wir einen Doppelzeiger `double **a;` und reservieren den Speicherplatz für m Zeiger:
   `a = malloc(m*sizeof(double*));`
3. Da `matr` eine Zeilenfolge darstellt, muss der Zeiger `a[i]` auf den Anfang der `i`-ten Zeile zeigen
   `for(i=0;i<m;i++) a[i] = matr + n*i;`
   oder äquivalent
   `for(i=0;i<m;i++) a[i] = &(matr[n*i]);`

```c
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>
int main(void)
{
      double **a, *matr;
      int i, j ,m=4, n=3;
      matr=calloc(m*n, sizeof(double));
      a = malloc(m*sizeof(double*));
      for(i=0;i<m;i++)
            a[i] = matr + n*i;
      // jetzt a[i][j] == matr[i*n+j]
      ...
      free(a);
      free(matr);
      return 0;
}
```

```c
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>
int main(void)
{
        double **a, matr[]={2, 4, 8, 7, 5, 6};
        int i, j ,m=2, n=3;
        a = malloc(m*sizeof(double*));
        for(i=0;i<m;i++)
                a[i] = matr + n*i;
        for(i=0;i<2;i++) {
                for(j=0;j<3;j++)
                        printf (" % le ", a[i][j]);
                printf("\n");
        }
        free(a);
        return 0;
}
```

Ergibt:

```
2 4 8
7 5 6
```