Strukturen

- Warum Strukturen?
- Members
- Punktoperator .
- Pfeiloperator ->
- Shallow Copy vs. Deep Copy

- `struct`
- `typedef`

Deklaration von Strukturen

- **Funktionen**
  - Zusammenfassung von versch. Befehlen, um Abstraktionsebenen zu schaffen
- **Strukturen**
  - Zusammenfassung von Variablen versch. Typs zu einem neuen Datentyp
  - Abstraktionsebenen bei Daten
- **Beispiel:** Verwaltung der EPROG-Teilnehmer
  - pro Student jeweils denselben Datensatz

```
 1  // Declaration of structure
 2  struct _Student_ {
 3    char* firstname; // Vorname
 4    char* lastname;  // Nachname
 5    int studentID;   // Matrikelnummer
 6    int studiesID;   // Studienkennzahl
 7    int test1;       // Noten der Tests
 8    int test2;
 9    int uebung;      // Note der Uebung
10  };
11
12  // Declaration of corresponding data type
13  typedef struct _Student_ Student;
```

- Semikolon nach Struktur-Deklarations-Block
- erzeugt neuen Variablen-Typ `Student`

Strukturen & Members

- Datentypen einer Struktur heißen Members
- Zugriff auf Members mit Punkt-Operator
  - `var` Variable vom Typ `Student`
  - z.B. Member `var.firstname`

```
 1  // Declaration of structure
 2  struct _Student_ {
 3    char* firstname; // Vorname
 4    char* lastname;  // Nachname
 5    int studentID;   // Matrikelnummer
 6    int studiesID;   // Studienkennzahl
 7    int test1;       // Noten der Tests
 8    int test2;
 9    int uebung;      // Note der Uebung
10  };
11
12  // Declaration of corresponding data type
13  typedef struct _Student_ Student;
14
15  main() {
16    Student var;
17    var.firstname = "Dirk";
18    var.lastname = "Praetorius";
19    var.studentID = 0;
20    var.studiesID = 680;
21    var.test1 = 3;
22    var.test2 = 4;
23    var.uebung = 5;
24  }
```


Bemerkungen zu Strukturen

- laut erstem C-Standard verboten:
  - Struktur als Input-Parameter einer Funktion
  - Struktur als Output-Parameter einer Funktion
  - Zuweisungsoperator (=) für gesamte Struktur
- in der Zwischenzeit erlaubt, aber trotzdem:
  - idR. Strukturen dynamisch über Pointer
  - Zuweisung (= Kopieren) selbst schreiben
  - Zuweisung (=) macht sog. *shallow copy*
- Shallow copy:
  - nur die unterste Ebene wird kopiert
  - d.h. Werte bei elementaren Variablen
  - d.h. Adressen bei Pointern
  - also: Kopie hat (physisch!) dieselben dynamischen Daten
- Deep copy:
  - alle Ebenen der Struktur werden kopiert
  - d.h. alle Werte bei elementaren Variablen
  - plus Kopie der dynamischen Inhalte (d.h. durch Pointer adressierter Speicher)

Strukturen: Speicher allokieren

- Also Funktionen anlegen
  - `newStudent`: Allokieren und Initialisieren
  - `freeStudent`: Freigeben des Speichers
  - `cloneStudent`: Vollständige Kopie der Struktur inkl. dyn. Felder, z.B. Member `firstname` (sog. *deep copy*)
  - `copyStudent`: Kopie der obersten Ebene exkl. dynamischer Felder (sog. *shallow copy*)

```
1   Student* newStudent() {
2     Student* pointer = malloc(sizeof(Student));
3
4     (*pointer).firstname = NULL;
5     (*pointer).lastname = NULL;
6     (*pointer).studentID = 0;
7     (*pointer).studiesID = 0;
8     (*pointer).test1 = 0;
9     (*pointer).test2 = 0;
10    (*pointer).uebung = 0;
11
12    return pointer;
13  }
```


Strukturen & Pfeiloperator

- Im Programm ist `pointer` vom Typ `Student*`
- Zugriff auf Members, z.B. `(*pointer).firstname`
  - Bessere Schreibweise dafür `pointer->firstname`
- Strukturen nie statisch, sondern stets dynamisch
  - Verwende gleich `student` für Typ `Student*`
- Funktion `newStudent` lautet besser wie folgt

```
1   Student* newStudent() {
2     Student* student = malloc(sizeof(Student));
3
4     student->firstname = NULL;
5     student->lastname = NULL;
6     student->studentID = 0;
7     student->studiesID = 0;
8     student->test1 = 0;
9     student->test2 = 0;
10    student->uebung = 0;
11
12    return student;
13  }
```


Strukturen: Speicher freigeben

- Freigeben einer dynamisch erzeugten Struktur-Variable vom Typ `Student`
- Achtung: Zugewiesenen dynamischen Speicher vor Freigabe des Strukturpointers freigeben

```
1   Student* delStudent(Student* student) {
2     if (student != NULL) {
3       if (student->firstname != NULL) {
4         free(student->firstname);
5       }
6
7       if (student->lastname != NULL) {
8         free(student->lastname);
9       }
10
11      free(student);
12    }
13    return NULL;
14  }
```


Shallow Copy

- Kopieren einer dynamisch erzeugten Struktur-Variable vom Typ `Student`
  - Kopieren der obersten Ebene einer Struktur exklusive dynamischen Speicher (Members!)

```
1   Student* copyStudent(Student* student) {
2     Student* copy = newStudent();
3
4     // ACHTUNG: Pointer!
5     copy->firstname = student->firstname;
6     copy->lastname = student->lastname;
7
8     // Kopieren der harmlosen Daten
9     copy->studentID = student->studentID;
10    copy->studiesID = student->studiesID;
11    copy->test1 = student->test1;
12    copy->test2 = student->test2;
13    copy->uebung = student->uebung
14
15    return copy;
16  }
```

Deep Copy

- Kopieren einer dynamisch erzeugten Struktur-Variable vom Typ `Student`
- Vollständige Kopie, inkl. dynamischem Speicher
- Achtung: Zugewiesenen dynamischen Speicher mitkopieren

```
 1  Student* cloneStudent(Student* student) {
 2    Student* copy = newStudent();
 3    int length = 0;
 4
 5    if (student->firstname != NULL) {
 6      length = strlen(student->firstname)+1;
 7      copy->firstname = malloc(length*sizeof(char));
 8      strcpy(copy->firstname, student->firstname);
 9    }
10
11
12    if (student->lastname != NULL) {
13      length = strlen(student->lastname)+1;
14      copy->lastname = malloc(length*sizeof(char));
15      strcpy(copy->lastname, student->lastname);
16    }
17
18    copy->studentID = student->studentID;
19    copy->studiesID = student->studiesID;
20    copy->test1 = student->test1;
21    copy->test2 = student->test2;
22    copy->uebung = student->uebung;
23
24    return copy;
25  }
```


Arrays von Strukturen

- Ziel: Array mit Teilnehmern von EPROG erstellen
- keine statischen Arrays verwenden, sondern dynamische Arrays
  - Studenten-Daten sind vom Typ `Student`
  - also intern verwaltet mittels Typ `Student*`
  - also Array vom Typ `Student**`

```
1  // Declare array
2  Student** participant=malloc(N*sizeof(Student*));
3
4  // Allocate memory for participants
5  for (j=0; j<N; ++j)
6    participant[j] = newStudent();
```

- Zugriff auf Members wie vorher
  - `participant[j]` ist vom Typ `Student*`
  - also z.B. `participant[j]->firstname`

Schachtelung von Strukturen

```
 1  struct _Address_ {
 2    char* street;
 3    char* number;
 4    char* city;
 5    char* zip;
 6  };
 7  typedef struct _Address_ Address;
 8
 9
10  struct _Employee_ {
11    char* firstname;
12    char* lastname;
13    char* title;
14    Address* home;
15    Address* office;
16  };
17  typedef struct _Employee_ Employee;
```

- Mitarbeiterdaten strukturieren
  - Name, Wohnadresse, Büroadresse
- Für `employee` vom Typ `Employee*`
  - `employee->home` Pointer auf `Address`
  - also z.B. `employee->home->city`
- Achtung beim Allokieren, Freigeben, Kopieren

Strukturen & Math

- Strukturen für mathematische Objekte:
  - Punkte im $\mathbb{R}^3$
  - allgemeine Vektoren
  - Matrizen

Strukturen für Punkte im $\mathbb{R}^3$

- Struktur zur Speicherung von $v = (x, y, z) \in \mathbb{R}^3$

```
1  // Declaration of structure
2  struct _Vector3_ {
3    double x;
4    double y;
5    double z;
6  };
7
8  // Declaration of corresponding data type
9  typedef struct _Vector3_ Vector3;
```

- kann Struktur-Deklaration und Datentyp-Definition verbinden

```
1  typedef struct _Vector3_ {
2    double x;
3    double y;
4    double z;
5  } Vector3;
```


Abstand zweier Punkte im $\mathbb{R}^3$

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <math.h>
4
5  typedef struct _Vector3_ {
6    double x;
7    double y;
8    double z;
9  } Vector3;
10
11 Vector3* newVector3(double x, double y, double z) {
12   Vector3* v = malloc(sizeof(Vector3));
13   v->x = x;
14   v->y = y;
15   v->z = z;
16   return v;
17 }
18
19 Vector3* delVector3(Vector3* v) {
20   free(v);
21   return NULL;
22 }
23
24 double dist(Vector3* v, Vector3* w) {
25   return sqrt( (v->x - w->x)*(v->x - w->x)
26              + (v->y - w->y)*(v->y - w->y)
27              + (v->z - w->z)*(v->z - w->z) );
28 }
29
30 main() {
31   Vector3* v = newVector3(1,1,1);
32   Vector3* w = newVector3(1,2,3);
33   printf("dist(x,y) = %f\n", dist(v,w));
34   v = delVector3(v);
35   w = delVector3(w);
36 }
```


Strukturen und Vektoren

- Datentyp zur Speicherung von $x \in \mathbb{R}^n$
  - Dimension n vom Typ `int`
  - Datenfeld x_j zur Speicherung von `double`

```
1  // Declaration of structure
2  struct _Vector_ {
3    int n;          // Dimension
4    double* entry;  // Vector coefficients
5  };
6
7  // Declaration of corresponding data type
8  typedef struct _Vector_ Vector;
```

- kann Struktur-Deklaration und Datentyp-Definition verbinden

```
1  typedef struct _Vector_ {
2    int n;          // Dimension
3    double* entry;  // Vector coefficients
4  } Vector;
```


Allokieren eines Vektors

- Funktion bekommt Länge $n \in \mathbb{N}$ des Vektors
- allokiert Struktur, weist Dimension n zu
- allokiert und initialisiert Datenfeld

```
1  Vector* newVector(int n) {
2    Vector* X = malloc(sizeof(Vector));
3    int i = 0;
4
5    X->n = n;
6    X->entry = malloc(n*sizeof(double));
7
8    for (i=0; i<n; ++i) {
9      X->entry[i] = 0;
10   }
11
12   return X;
13 }
```

Freigeben eines Vektors

- Datenfeld freigeben
- Struktur freigeben
- `NULL` zurückgeben

```
1  Vector* delVector(Vector* X) {
2    free(X->entry);
3    free(X);
4
5    return NULL;
6  }
```

Zugriff auf Strukturen

- Es ist guter (aber seltener) Programmierstil, auf Members einer Struktur nicht direkt zuzugreifen
- Stattdessen lieber
  - für jeden Member `set` und `get` schreiben

```
1   int getVectorN(Vector* X) {
2     return X->n;
3   }
4
5
6   double getVectorEntry(Vector* X, int i) {
7     return X->entry[i];
8   }
9
10
11  void setVectorEntry(Vector* X, int i, double Xi){
12    X->entry[i] = Xi;
13  }
```

- Wenn kein `set`, dann Schreiben nicht erlaubt!
- Wenn kein `get`, dann Lesen nicht erlaubt!
- Dieses Vorgehen erlaubt leichte Umstellung der Datenstruktur bei späteren Modifikationen

Beispiel: Vektor einlesen

```
1   Vector* inputVector() {
2
3     Vector* X = NULL;
4     int i = 0;
5     int n = 0;
6     double input = 0;
7
8     printf("Dimension des Vektors n=");
9     scanf("%d",&n);
10
11    X = newVector(n);
12
13    for (i=0; i<n; ++i) {
14      input = 0;
15      printf("x[%d]=",i);
16      scanf("%lf",&input);
17      setVectorEntry(X,i,input);
18    }
19
20    return X;
21  }
```

- Einlesen von $n \in \mathbb{N}$ und eines Vektors $x \in \mathbb{R}^n$

Beispiel: Euklidische Norm

```
1   double normVector(Vector* X) {
2
3     int n = getVectorN(X);
4     int i = 0;
5     double Xi = 0;
6     double norm = 0;
7
8     for (i=0; i<n; ++i) {
9       Xi = getVectorEntry(X,i);
10      norm = norm + Xi*Xi;
11    }
12    norm = sqrt(norm);
13
14    return norm;
15  }
```

- Berechne $\|x\| := \left(\sum_{j=1}^{n} x_j^2\right)^{1/2}$ für $x \in \mathbb{R}^n$

Beispiel: Skalarprodukt

```
1   double productVector(Vector* X, Vector* Y) {
2
3     int n = getVectorN(X);
4     double Xi = 0;
5     double Yi = 0;
6     double product = 0;
7     int i = 0;
8
9     for (i=0; i<n; ++i) {
10      Xi = getVectorEntry(X,i);
11      Yi = getVectorEntry(Y,i);
12      product = product + Xi*Yi;
13    }
14
15    return product;
16  }
```

- Berechne $x \cdot y := \sum_{j=1}^{n} x_j y_j$ für $x, y \in \mathbb{R}^n$

Strukturen und Matrizen

- Datentyp zur Speicherung von $A \in \mathbb{R}^{m \times n}$
  - Dimensionen m, n vom Typ `int`
  - Datenfeld A_{ij} zur Speicherung von `double`

```
1   // Declaration of structure
2   struct _Matrix_ {
3     int m;           // Dimension
4     int n;
5     double** entry; // Matrix entries
6   };
7
8   // Declaration of corresponding data type
9   typedef struct _Matrix_ Matrix;
```

- kann Struktur-Deklaration und Datentyp-Definition verbinden

```
1   typedef struct _Matrix_ {
2     int m;           // Dimension
3     int n;
4     double** entry; // Matrix entries
5   } Matrix;
```


Allokieren einer Matrix

- Wir speichern die Einträge der Matrix als `double**`
  - Allokation der Einträge wie oben besprochen

```
1   Matrix* newMatrix(int m, int n) {
2     int i = 0;
3     int j = 0;
4
5     Matrix* A = malloc(sizeof(Matrix));
6
7     A->m = m;
8     A->n = n;
9     A->entry = malloc(m*sizeof(double*));
10
11    for (i=0; i<m; ++i) {
12      A->entry[i] = malloc(n*sizeof(double));
13      for (j=0; j<n; ++j) {
14        A->entry[i][j] = 0;
15      }
16    }
17
18    return A;
19  }
```


Freigeben einer Matrix

- Erst Datenfeld `A->entry` freigeben
  - erst Zeilenvektoren freigeben
  - dann Spaltenvektor freigeben
- Dann Struktur freigeben

```
1   Matrix* delMatrix(Matrix* A) {
2     int i = 0;
3
4     for (i=0; i<A->m; ++i) {
5       free(A->entry[i]);
6     }
7
8     free(A->entry);
9
10    free(A);
11
12    return NULL;
13  }
```


Zugriffsfunktionen

```
1   int getMatrixM(Matrix* A) {
2     return A->m;
3   }
4
5
6   int getMatrixN(Matrix* A) {
7     return A->n;
8   }
9
10
11  double getMatrixEntry(Matrix* A, int i, int j) {
12    return A->entry[i][j];
13  }
14
15
16  void setMatrixEntry(Matrix* A, int i, int j,
17                      double Aij) {
18    A->entry[i][j] = Aij;
19  }
```

Beispiel: Matrix-Vektor-Produkt

```
 1  Vector* matrixvector(Matrix* A, Vector* X) {
 2
 3    int m = getMatrixM(A);
 4    int n = getMatrixN(A);
 5    Vector* B = newVector(m);
 6    double Aij = 0;
 7    double Xj = 0;
 8    double Bi = 0;
 9    int i = 0;
10    int j = 0;
11
12    for (i=0; i<m; ++i) {
13      Bi = 0;
14      for (j=0; j<n; ++j) {
15        Aij = getMatrixEntry(A,i,j);
16        Xj = getVectorEntry(X,j);
17        Bi = Bi + Aij*Xj;
18      }
19      setVectorEntry(B,i,Bi);
20    }
21
22    return B;
23  }
```

- Gegeben $A \in \mathbb{R}^{m \times n}$ und $x \in \mathbb{R}^n$
- Berechne $b \in \mathbb{R}^m$ mit $b_i = \sum_{j=1}^{n} A_{ij} x_j$

Beispiel: Zeilensummennorm

```
 1  double normMatrix(Matrix* A) {
 2
 3    int m = getMatrixM(A);
 4    int n = getMatrixN(A);
 5    double Aij = 0;
 6    double max = 0;
 7    double sum = 0;
 8    int i = 0;
 9    int j = 0;
10
11    for (i=0; i<m; ++i) {
12      sum = 0;
13      for (j=0; j<n; ++j) {
14        Aij = getMatrixAij(A,i,j);
15        sum = sum + fabs(Aij);
16      }
17      if (sum > max) {
18        max = sum;
19      }
20    }
21
22    return max;
23  }
```

- Gegeben $A \in \mathbb{R}^{m \times n}$
- Berechne $\|A\|_Z := \max_{i=1,\ldots,m} \sum_{j=1}^{n} |A_{ij}|$

Strukturen und Matrizen, v2

- Manchmal Fortran-Bib nötig, z.B. LAPACK
  - will auf `A->entry` Fortran-Routinen anwenden!
- Fortran speichert $A \in \mathbb{R}^{m \times n}$ spaltenweise in Vektor der Länge mn
  - A_{ij} entspricht `A[i+j*m]`, wenn $A \in \mathbb{R}^{m \times n}$

```
 1  typedef struct _Matrix_ {
 2    int m;
 3    int n;
 4    double* entry;
 5  } Matrix;
```

- Allokieren der neuen Matrix-Struktur

```
 6  Matrix* newMatrix(int m, int n) {
 7    int i = 0;
 8
 9    Matrix* A = malloc(sizeof(Matrix));
10
11    A->m = m;
12    A->n = n;
13    A->entry = malloc(m*n*sizeof(double));
14
15    for (i=0; i<m*n; ++i) {
16      A->entry[i] = 0;
17    }
18
19    return A;
20  }
```


Noch einmal free, set, get

- Freigeben der neuen Matrix-Struktur

```
 1  Matrix* delMatrix(Matrix* A) {
 2    free(A->entry);
 3    free(A);
 4    return NULL;
 5  }
```

- `set` und `get` für Matrix-Einträge

```
 7  void setMatrixEntry(Matrix* A, int i, int j,
 8                      double Aij) {
 9    A->entry[i+j*A->m] = Aij;
10  }

11  double getMatrixEntry(Matrix* A, int i, int j) {
12    return A->entry[i+j*A->m];
13  }
```

- Plötzlich werden `set` und `get` eine gute Idee
  - verhindert Fehler!
  - macht Programm im Nachhinein flexibel, z.B. bei Änderungen an Datenstruktur