Kommentarzeilen

- wozu Kommentarzeilen?
- //
- /* ... */

Kommentarzeilen

- werden vom Interpreter/Compiler ausgelassen
- nur für den Leser des Programmcodes
- notwendig, um eigene Programme auch später noch zu begreifen
  - deshalb brauchbar für Übung?
- notwendig, damit andere den Code verstehen
  - soziale Komponente der Übung?
- extrem brauchbar zum debuggen
  - Teile des Source-Code "auskommentieren", sehen was passiert...
  - vor allem bei Fehlermeldungen des Parser
- Wichtige Regeln:
  - nie dt. Sonderzeichen verwenden
  - nicht zu viel und nicht zu wenig
  - zu Beginn des Source-Codes stets Autor & letzte Änderung kommentieren
    - vermeidet das Arbeiten an alten Versionen...

Kommentarzeilen in C

```
1   #include <stdio.h>
2
3   main() {
4     // printf("1 ");
5     printf("2 ");
6     /*
7       printf("3");
8       printf("4");
9     */
10    printf("5");
11    printf("\n");
12  }
```

- Gibt in C zwei Typen von Kommentaren:
  - einzeiliger Kommentar
    - eingeleitet durch //, geht bis Zeilenende
    - z.B. Zeile 4
    - stammt eigentlich aus C++
  - mehrzeiliger Kommentar
    - alles zwischen /* (Anfang) und */ (Ende)
    - z.B. Zeile 6–9
    - darf nicht geschachtelt werden!
      - d.h. /* ... /* ... */ ... */ ist Syntaxfehler
- Vorschlag
  - Verwende // für echte Kommentare
  - Verwende /* ... */ zum Debuggen
- Output:

```
2 5
```


Beispiel: Euklids Algorithmus

```
1   // author: Dirk Praetorius
2   // last modified: 19.03.2013
3
4   // Euklids Algorithmus zur Berechnung des ggT
5   // basiert auf ggT(a,b) = ggT(a-b,b) fuer a>b
6   // und ggT(a,b) = ggT(b,a)
7
8   int euklid(int a, int b) {
9     int tmp = 0;
10
11    // iteriert Uebergang ggT(a,b) = ggT(a-b,b),
12    // realisiert mittels Divisionsrest, bis
13    // b = 0. Dann war a==b, also ggT = a
14
15    while (b != 0) {
16      tmp = b;
17      b = a%b;
18      a = tmp;
19    }
20
21    return a;
22  }
```

Pointer

- Variable vs. Pointer
- Dereferenzieren
- Address-of Operator `&`
- Dereference Operator `*`
- Call by Reference

Variablen

- Variable = symbolischer Name für Speicherbereich
  - + Information, wie Speicherbereich interpretiert werden muss (Datentyp laut Deklaration)
- Compiler übersetzt Namen in Referenz auf Speicherbereich und merkt sich, wie dieser interpretiert werden muss

Pointer

- Pointer = Variable, die Adresse eines Speicherbereichs enthält
- Dereferenzieren = Zugriff auf den Inhalt eines Speicherbereichs mittels Pointer
  - Beim Dereferenzieren muss Compiler wissen, welcher Var.typ im gegebenen Speicherbereich liegt, d.h. wie Speicherbereich interpretiert werden muss

Pointer in C

- Pointer & Variablen sind in C eng verknüpft:
  - `var` Variable ⇒ `&var` zugehöriger Pointer
  - `ptr` Pointer ⇒ `*ptr` zugehörige Variable
  - insbesondere `*&var` = `var` sowie `&*ptr` = `ptr`
- Bei Deklaration muss Typ des Pointers angegeben werden, da `*ptr` eine Variable sein soll!
  - `int* ptr;` deklariert `ptr` als Pointer auf `int`
- Wie üblich gleichzeitige Initialisierung möglich
  - `int var;` deklariert Variable `var` vom Typ `int`
  - `int* ptr = &var;` deklariert `ptr` und weist Speicheradresse der Variable `var` zu
    - Bei solchen Zuweisungen muss der Typ von Pointer und Variable passen, sonst passiert Unglück!
      - I.a. gibt Compiler eine Warnung aus, z.B. `incompatible pointer type`
- Analog für andere Datentypen, z.B. `double`

Ein elementares Beispiel

```
 1  #include <stdio.h>
 2
 3  main() {
 4    int var = 1;
 5    int* ptr = &var;
 6
 7    printf("a) var = %d, *ptr = %d\n",var,*ptr);
 8
 9    var = 2;
10    printf("b) var = %d, *ptr = %d\n",var,*ptr);
11
12    *ptr = 3;
13    printf("c) var = %d, *ptr = %d\n",var,*ptr);
14
15    var = 47;
16    printf("d) *(&var) = %d,",*(&var));
17    printf("*&var = %d\n",*&var);
18
19    printf("e) &var = %p\n", &var);
20  }
```

- Output:

```
a) var = 1, *ptr = 1
b) var = 2, *ptr = 2
c) var = 3, *ptr = 3
d) *(&var) = 47,*&var = 47
e) &var = 0x7fff518baba8
```

Call by Reference in C

- Elementare Datentypen werden in C mit *Call by Value* an Funktionen übergeben
  - z.B. `int`, `double`, Pointer
- *Call by Reference* ist über Pointer realisierbar:

```
 1  #include <stdio.h>
 2
 3  void test(int* y) {
 4    printf("a) *y=%d\n", *y);
 5    *y = 43;
 6    printf("b) *y=%d\n", *y);
 7  }
 8
 9
10  main() {
11    int x = 12;
12    printf("c) x=%d\n", x);
13    test(&x);
14    printf("d) x=%d\n", x);
15  }
```

- Output:

  c) x=12

  a) *y=12

  b) *y=43

  d) x=43

Begrifflichkeiten

- Call by Value
  - Funktionen erhalten Werte der Input-Parameter und speichern diese in lokalen Variablen
  - Änderungen an den Input-Parameter wirken sich nicht außerhalb der Funktion aus
- Call by Reference
  - Funktionen erhalten Variablen als Input ggf. unter lokal neuem Namen
  - Änderungen an den Input-Parametern wirken sich außerhalb der Funktion aus

Wiederholung

- Standard in C ist Call by Value
- Kann Call by Reference mittels Pointern realisieren
- Vektoren werden mit Call by Reference übergeben

Warum Call by Reference?

- Funktionen haben in C maximal 1 Rückgabewert
- Falls Fkt mehrere Rückgabewerte haben soll ...

Ein Beispiel

```
 1  #include <stdio.h>
 2  #define DIM 5
 3
 4  void scanvector(double input[], int dim) {
 5    int j = 0;
 6    for (j=0; j<dim; ++j) {
 7      input[j] = 0;
 8      printf("%d: ",j);
 9      scanf("%lf",&input[j]);
10    }
11  }
12
13  void minmax(double vector[],int dim,
14              double* min, double* max) {
15    int j = 0;
16    *max = vector[0];
17    *min = vector[0];
18
19    for (j=1; j<dim; ++j) {
20      if (vector[j] < *min) {
21        *min = vector[j];
22      }
23      else if (vector[j] > *max) {
24        *max = vector[j];
25      }
26    }
27  }
28
29  main() {
30    double x[DIM];
31    double max = 0;
32    double min = 0;
33    scanvector(x,DIM);
34    minmax(x,DIM, &min, &max);
35    printf("min(x) = %f\n",min);
36    printf("max(x) = %f\n",max);
37  }
```

- Funktion `minmax` liefert mittels Call by Reference
  - Minimum und Maximum eines Vektors

Anmerkungen zu Pointern

- Standard-Notation zur Deklaration ist anders als meine Sichtweise:
  - `int *pointer` deklariert Pointer auf `int`
- Von den C-Erfindern wurden Pointer *nicht* als Variablen verstanden
- Für das Verständnis scheint mir aber "variable" Sichtweise einfacher
- Leerzeichen wird vom Compiler ignoriert:
  - `int* pointer`, `int *pointer`, `int*pointer`
- `*` wird nur auf den folgenden Namen bezogen
- ACHTUNG bei Deklaration von Listen:
  - `int* pointer, var;` deklariert Pointer auf `int` und Variable vom Typ `int`
  - `int *pointer1, *pointer2;` deklariert zwei Pointer auf `int`
- ALSO Listen von Pointern vermeiden!
  - auch Zwecks Lesbarkeit!

Elementare Datentypen

- Arrays & Pointer
- sizeof

Elementare Datentypen

C kennt folgende elementare Datentypen:

- Datentyp für Zeichen (z.B. Buchstaben)
  - `char`
- Datentypen für Ganzzahlen:
  - `short`
  - `int`
  - `long`
- Datentypen für Gleitkommazahlen:
  - `float`
  - `double`
  - `long double`
- Alle Pointer gelten als elementare Datentypen

Bemerkungen:

- Deklaration und Gebrauch wie bisher
- Man kann Arrays & Pointer bilden
- Für UE nur `char`, `int`, `double` & Pointer
- Genaueres zu den Typen später!

Der Befehl `sizeof`

```
 1  #include <stdio.h>
 2
 3  void printf_sizeof(double vector[]) {
 4    printf("sizeof(vector) = %d\n",sizeof(vector));
 5  }
 6
 7  main() {
 8    int var = 43;
 9    double array[11];
10    double* ptr = array;
11
12    printf("sizeof(var) = %d\n",sizeof(var));
13    printf("sizeof(double) = %d\n",sizeof(double));
14    printf("sizeof(array) = %d\n",sizeof(array));
15    printf("sizeof(ptr) = %d\n",sizeof(ptr));
16    printf_sizeof(array);
17  }
```

- Ist `var` eine Variable eines elementaren Datentyps, gibt `sizeof(var)` die Größe der Var. in Bytes zurück
- Ist `type` ein Datentyp, so gibt `sizeof(type)` die Größe einer Variable dieses Typs in Bytes zurück
- Ist `array` ein *lokales statisches Array*, so gibt `sizeof(array)` die Größe des Arrays in Bytes zurück
- Output:

```
sizeof(var) = 4
sizeof(double) = 8
sizeof(array) = 88
sizeof(ptr) = 8
sizeof(vector) = 8
```


Funktionen

- Elementare Datentypen werden an Funktionen mit Call by Value übergeben
- Return Value einer Funktion darf nur `void` oder ein elementarer Datentyp sein

Arrays

- Streng genommen, gibt es in C keine Arrays!
  - Deklaration `int array[N];`
    - legt Pointer `array` vom Typ `int*` an
    - organisiert ab der Adresse `array` Speicher, um `N`-mal einen `int` zu speichern
    - d.h. `array` enthält Adresse von `array[0]`
  - Da Pointer als elementare Datentypen mittels Call by Value übergeben werden, werden Arrays augenscheinlich mit Call by Reference übergeben

Laufzeitfehler!

```
 1  #include <stdio.h>
 2
 3  double* scanfvector(int length) {
 4    double vector[length];
 5    int j = 0;
 6    for (j=0; j<length; ++j) {
 7      vector[j] = 0;
 8      printf("vector[%d] = ",j);
 9      scanf("%lf",&vector[j]);
10    }
11    return vector;
12  }
13
14  main() {
15    double* x;
16    int j = 0;
17    int dim = 0;
18
19    printf("dim = ");
20    scanf("%d",&dim);
21
22    x = scanfvector(dim);
23
24    for (j=0; j<dim; ++j) {
25      printf("x[%d] = %f\n",j,x[j]);
26    }
27  }
```

- Syntax des Programms ist OK
- Problem: Speicher zu `x` mit Blockende 12 aufgelöst
  - d.h. Pointer aus 11 zeigt auf Irgendwas
- Abhilfe: Call by Reference (vorher!) oder händische Speicherverwaltung (gleich!)

Dynamische Vektoren

- statische & dynamische Vektoren
- Vektoren & Pointer
- dynamische Speicherverwaltung

- `stdlib.h`
- `NULL`
- `malloc`, `realloc`, `free`

- `#ifndef ... #endif`

Statische Vektoren

- `double array[N];` deklariert statischen Vektor `array` der Länge `N` mit `double`-Komponenten
  - Indizierung `array[j]` mit $0 \leq j \leq N-1$
  - `array` ist intern vom Typ `double*`
    - enthält Adr. von `array[0]`, sog. *Base Pointer*
  - Länge `N` kann während Programmablauf nicht verändert werden

- Funktionen können Länge `N` nicht herausfinden
  - Länge `N` als Input-Parameter übergeben

Speicher allokieren

- Nun händische Speicherverwaltung von Arrays
  - dadurch Vektoren dynamischer Länge möglich

- Einbinden der Standard-Bibl: `#include <stdlib.h>`
  - wichtige Befehle `malloc`, `free`, `realloc`

- `pointer = malloc(N*sizeof(type));`
  - allokiert Speicher für Vektor der Länge `N` mit Komponenten vom Typ `type`
    - `malloc` kriegt Angabe in Bytes → `sizeof`
  - `pointer` muss vom Typ `type*` sein
    - Base Pointer `pointer` bekommt Adresse der ersten Komponente `pointer[0]`
  - `pointer` und `N` muss sich Prg merken!

- **Häufiger Laufzeitfehler:** `sizeof` vergessen!
- **Achtung:** Allokierter Speicher ist uninitialisiert!

Speicher freigeben

- `free(pointer)`
  - gibt Speicher eines dyn. Vektors frei
  - `pointer` muss Output von `malloc` sein
- **Achtung:** Speicher wird freigegeben, aber `pointer` existiert weiter
  - Erneuter Zugriff führt (irgendwann) auf Laufzeitfehler
- **Achtung:** Speicher freigeben, nicht vergessen!
- **Konvention:** Pointer ohne Speicher bekommen den Wert `NULL` zugewiesen
  - führt sofort auf Speicherzugriffsfehler bei Zugriff

Beispiel

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  double* scanfvector(int length) {
 5    int j = 0;
 6    double* vector = malloc(length*sizeof(double));
 7    for (j=0; j<length; ++j) {
 8      vector[j] = 0;
 9      printf("vector[%d] = ",j);
10      scanf("%lf",&vector[j]);
11    }
12    return vector;
13  }
14
15  void printfvector(double* vector, int length) {
16    int j = 0;
17    for (j=0; j<length; ++j) {
18      printf("vector[%d] = %f\n",j,vector[j]);
19    }
20  }
21
22  main() {
23    double* x = NULL;
24    int dim = 0;
25
26    printf("dim = ");
27    scanf("%d",&dim);
28
29    x = scanfvector(dim);
30    printfvector(x,dim);
31
32    free(x);
33    x = NULL;
34  }
```


Dynamische Vektoren

- `pointer = realloc(pointer,Nnew*sizeof(type))`
  - verändert Speicherallokation
    - zusätzliche Allokation für `Nnew` $>$ `N`
    - Speicherbereich kürzen für `Nnew` $<$ `N`
  - Alter Inhalt bleibt (soweit möglich) erhalten

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  main() {
 5    int N = 5;
 6    int Nnew = 10;
 7    int j = 0;
 8
 9    int* array = malloc(N*sizeof(int));
10
11    for (j=0; j<N; ++j)
12      array[j] = j;
13
14    array = realloc(array,Nnew*sizeof(int));
15
16    for (j=N; j<Nnew; ++j)
17      array[j] = 10*j;
18
19    for (j=0; j<Nnew; ++j)
20      printf("%d ",array[j]);
21    printf("\n");
22
23    free(array);
24    array = NULL;
25  }
```

- Output:

```
0 1 2 3 4 50 60 70 80 90
```


Bemerkungen

- Base Pointer (= Output von `malloc` bzw. `realloc`) merken & nicht verändern
  - notwendig für fehlerfreies `free` und `realloc`
- bei `malloc` und `realloc` nicht `sizeof` vergessen
  - Typ des Base Pointers muss zum `sizeof` passen!
- allokierter Speicherbereich ist stets uninitialisiert
  - nach Allokation stets initialisieren
- Länge des dynamischen Arrays merken
  - kann Programm nicht herausfinden!
- Nicht mehr benötigten Speicher freigeben
  - insb. vor Blockende }, da dann Base Pointer weg
- Pointer auf `NULL` setzen, wenn ohne Speicher
  - Fehlermeldung, falls Programm "aus Versehen" auf Komponente `array[j]` zugreift
- Nie `realloc`, `free` auf statisches Array anwenden
  - Führt auf Laufzeitfehler, da Compiler `free` selbständig hinzugefügt hat!
- Ansonsten gleicher Zugriff auf Komponenten wie bei statischen Arrays
  - Indizierung `array[j]` für $0 \leq j \leq N-1$

Eine erste Bibliothek

- Header-File `dynamicvectors.h` zur Bibliothek
  - enthält alle Funktionssignaturen
  - enthält Kommentare zu den Funktionen
- Header-File beginnt mit

  ```
  #ifndef NAME
  #define NAME
  ```

- Header-File ended mit

  ```
  #endif
  ```

- erlaubt mehrfaches Einbinden
  - vermeidet doppelte Deklaration

```
 1  #ifndef DYNAMICVECTORS
 2  #define DYNAMICVECTORS
 3
 4  #include <stdio.h>
 5  #include <stdlib.h>
 6
 7  // allocate and initialize dynamic double vector of length n
 8  double* mallocvector(int n);
 9
10  // free a dynamic vector and set the pointer to NULL
11  double* freevector(double* vector);
12
13  // extend dynamic double vector and initialize new entries
14  double* reallocvector(double* vector, int n, int nnew);
15
16  // allocate dynamic double vector of length n and read
17  // entries from keyboard
18  double* scanfvector(int n);
19
20  // print dynamic double vector of length n to shell
21  double* printfvector(double* vector, int n);
22
23  #endif
```


Source-Code (Ausschnitt)

```
 1  #include "dynamicvectors.h"
 2
 3  double* mallocvector(int n) {
 4    int j = 0;
 5    double* vector = malloc(n*sizeof(double));
 6    for (j=0; j<n; ++j)
 7      vector[j] = 0;
 8    return vector;
 9  }
10
11  double* freevector(double* vector) {
12    free(vector);
13    return NULL;
14  }
15
16  double* reallocvector(double* vector,
17                        int n, int nnew) {
18    int j = 0;
19    vector = realloc(vector,nnew*sizeof(double));
20    for (j=n; j<nnew; ++j)
21      vector[j] = 0;
22    return vector;
23  }
24
25  double* scanfvector(int n) {
26    int j = 0;
27    double* vector = mallocvector(n);
28    for (j=0; j<n; ++j) {
29      vector[j] = 0;
30      printf("vector[%d] = ",j);
31      scanf("%lf",&vector[j]);
32    }
33    return vector;
34  }
```