Blöcke

- Blöcke {...}
- Deklaration von Variablen
- Lifetime & Scope
- Lokale & globale Variablen

Lifetime & Scope

- Lifetime einer Variable
  = Zeitraum, in dem Speicherplatz zugewiesen ist
  = Zeitraum, in dem Variable existiert
- Scope einer Variable
  = Zeitraum, in dem Variable sichtbar ist
  = Zeitraum, in dem Variable gelesen/verändert werden kann
- Scope $\subseteq$ Lifetime

Globale & Lokale Variablen

- globale Variablen
  = Variablen, die globale Lifetime haben (bis Programm terminiert)
  - eventuell lokaler Scope
  - werden am Anfang außerhalb von `main` deklariert
- lokale Variablen
  = Variablen, die nur lokale Lifetime haben

Blöcke

- Blöcke stehen innerhalb von { ... }
- Jeder Block startet mit Deklaration zusätzlich benötigter Variablen
  - Variablen *können/dürfen* nur am Anfang eines Blocks deklariert werden
- Die innerhalb des Blocks deklarierten Variablen werden nach Blockende vergessen (= gelöscht)
  - d.h. Lifetime endet
  - lokale Variablen
- Schachtelung { ... { ... } ... }
  - beliebige Schachtelung ist möglich
  - Variablen aus äußerem Block können im inneren Block gelesen und verändert werden, umgekehrt *nicht*. Änderungen bleiben wirksam.
    - d.h. Lifetime & Scope nur nach Innen vererbt
  - Wird im äußeren und im inneren Block Variable `var` definiert, so wird das „äußere“ `var` überdeckt und ist erst wieder ansprechbar (mit gleichem Wert wie vorher), wenn der innere Block beendet wird.
    - d.h. äußeres `var` ist nicht im inneren Scope
    - Das ist schlechter Programmierstil!

Bsp. zu lokalen & globalen Var.

```
1   #include <stdio.h>
2
3   int var0 = 5;
4
5   main() {
6     int var1 = 7;
7     int var2 = 9;
8
9     printf("a) %d, %d, %d\n", var0, var1, var2);
10    {
11      int var1 = 17;
12
13      printf("b) %d, %d, %d\n", var0, var1, var2);
14      var0 = 15;
15      var2 = 19;
16      printf("c) %d, %d, %d\n", var0, var1, var2);
17      {
18        int var0 = 25;
19        printf("d) %d, %d, %d\n", var0, var1, var2);
20      }
21    }
22    printf("e) %d, %d, %d\n", var0, var1, var2);
23  }
```

- Output:

```
a) 5, 7, 9
b) 5, 17, 9
c) 15, 17, 19
d) 25, 17, 19
e) 15, 7, 19
```

Funktionen

- Funktion
- Eingabe- / Ausgabeparameter
- Call by Value / Call by Reference

- `return`
- `void`

Funktionen

- Funktion = Zusammenfassung mehrerer Anweisungen zu einem aufrufbaren Ganzen
  - `output = function(input)`
    - Eingabeparameter `input`
    - Ausgabeparameter (Return Value) `output`
- Warum Funktionen?
  - Zerlegung eines großen Problems in überschaubare kleine Teilprobleme
  - Strukturierung von Programmen (Abstraktionsebenen)
  - Wiederverwertung von Programm-Code
- Funktion besteht aus Signatur und Rumpf (Body)
  - Signatur = Fkt.name & Eingabe-/Ausgabepar.
    - Anzahl & Reihenfolge ist wichtig!
  - Rumpf = Programmzeilen der Funktion

Funktionen in C

- In C können Funktionen
  - mehrere (oder keinen) Parameter übernehmen
  - einen einzigen oder keinen Rückgabewert liefern
  - Rückgabewert muss elementarer Datentyp sein
    - z.B. `double`, `int`
- Signatur hat folgenden Aufbau
  `<type of return value> <function name>(parameters)`
  - Funktion ohne Rückgabewert:
    - `<type of return value>` = `void`
  - Sonst: `<type of return value>` = Variablentyp
  - `parameters` = Liste der Übergabeparameter
    - getrennt durch Kommata
    - *vor* jedem Parameter Variablentyp angeben
    - *kein* Parameter ⇒ leere Klammer `()`
- Rumpf ist ein Block
  - Rücksprung ins Hauptprogramm mit `return` oder bei Erreichen des Funktionsblock-Endes, falls Funktionstyp = `void`
  - Rücksprung ins Hauptprogramm mit `return output`, falls die Variable `output` zurückgegeben werden soll
  - Häufiger Fehler: `return` vergessen
    - Dann Rückgabewert zufällig!
    - ⇒ Irgendwann Chaos (Laufzeitfehler!)

Variablen

- Alle Variablen, die im Funktionsblock deklariert werden, sind lokale Variablen
- Alle elementaren Variablen, die in Signatur deklariert werden, sind lokale Variablen
- Funktion bekommt Input-Parameter als Werte, ggf. Type Casting!

Call by Value

- Dass bei Funktionsaufrufen Input-Parameter in lokale Variablen kopiert werden, bezeichnet man als Call by Value
  - Es wird neuer Speicher angelegt, der Wert der Eingabe-Parameter wird in diese abgelegt

Beispiel: Quadrieren

```
1   #include <stdio.h>
2
3   double square(double x) {
4     return x*x;
5   }
6
7   main() {
8     double x = 0;
9     printf("Input x = ");
10    scanf("%lf",&x);
11    printf("%f^2 = %f\n",x,square(x));
12  }
```

- Compiler muss Funktion vor Aufruf kennen
  - d.h. Funktion vor aufrufender Zeile definieren
- Ausführung startet immer bei `main()`
- Die Variable `x` in Funktion `square` und die Variable `x` in Funktion `main` sind verschieden!
- Eingabe von 5 ergibt als Output

```
Input x = 5
5^2 = 25.000000
```


Beispiel: Minimum zweier Zahlen

```
1   #include <stdio.h>
2
3   double min(double x, double y) {
4     if (x > y) {
5       return y;
6     }
7     else {
8       return x;
9     }
10  }
11
12  main() {
13    double x = 0;
14    double y = 0;
15
16    printf("Input x = ");
17    scanf("%lf",&x);
18    printf("Input y = ");
19    scanf("%lf",&y);
20    printf("min(x,y) = %f\n",min(x,y));
21  }
```

- Eingabe von 10 und 2 ergibt als Output

```
Input x = 10
Input y = 2
min(x,y) = 2.000000
```

- Programm erfüllt Aufgabenstellung der UE:
  - Funktion mit gewisser Funktionalität
  - aufrufendes Hauptprogramm mit
    - Daten einlesen
    - Funktion aufrufen
    - Ergebnis ausgeben

Deklaration von Funktionen

```
1   #include <stdio.h>
2
3   double min(double, double);
4
5   main() {
6     double x = 0;
7     double y = 0;
8
9     printf("Input x = ");
10    scanf("%lf",&x);
11    printf("Input y = ");
12    scanf("%lf",&y);
13    printf("min(x,y) = %f\n",min(x,y));
14  }
15
16  double min(double x, double y) {
17    if (x > y) {
18      return y;
19    }
20    else {
21      return x;
22    }
23  }
```

- Bei vielen Funktionen wird Code unübersichtlich
  - Alle Funktionen oben deklarieren, vgl. Zeile 3
    - Compiler weiß dann, wie Funktion agiert
  - vollständiger Fkt.code folgt, vgl. Zeile 16-23
- Alternative Deklaration = Fkt.code ohne Rumpf
  - `double min(double x, double y);`
    vgl. Zeile 3, 16
- in Literatur: *Forward Declaration* und *Prototyp*

Call by Value

```
1   #include <stdio.h>
2
3   void test(int x) {
4     printf("a) x=%d\n", x);
5     x = 43;
6     printf("b) x=%d\n", x);
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) x=12
b) x=43
d) x=12
```

Call by Reference

- Bei anderen Programmiersprachen, wird nicht der Wert eines Input-Parameters an eine Funktion übergeben, sondern dessen Speicheradresse (Call by Reference)
  - d.h. Änderungen an der Variable sind auch außerhalb der Funktion sichtbar

```
1   void test(int y) {
2     printf("a) y=%d\n", y);
3     y = 43;
4     printf("b) y=%d\n", y);
5   }
6
7
8   main() {
9     int x = 12;
10    printf("c) x=%d\n", x);
11    test(x);
12    printf("d) x=%d\n", x);
13  }
```

- Dieser Source-Code ist kein C-Code!
  - Ziel: nur was-wäre-wenn erklären!
- *Call by Reference* würde folgenden Output liefern:

```
c) x=12
a) y=12
b) y=43
d) x=43
```


Type Casting & Call by Value

```
1   #include <stdio.h>
2
3   double divide(double, double);
4
5   main() {
6     int int1 = 2;
7     int int2 = 3;
8
9     printf("a) %f\n", int1 / int2 );
10    printf("b) %f\n", divide(int1,int2));
11  }
12
13  double divide(double dbl1, double dbl2) {
14    return(dbl1 / dbl2);
15  }
16
```

- Type Casting von `int` auf `double` bei Übergabe
- Output:

```
a) 0.000000
b) 0.666667
```


Type Casting (Negativbeispiel!)

```
1   #include <stdio.h>
2
3   int isequal(int, int);
4
5   main() {
6     double x = 4.1;
7     double y = 4.9;
8
9     if (isequal(x,y)) {
10        printf("x == y\n");
11    }
12    else {
13        printf("x != y\n");
14    }
15  }
16
17  int isequal(int x, int y) {
18    if (x == y) {
19      return 1;
20    }
21    else {
22      return 0;
23    }
24  }
```

- Output:

```
x == y
```

- Aber eigentlich $x \neq y$!
  - Implizites Type Casting von `double` auf `int` durch Abschneiden, denn Input-Parameter sind `int`
- Achtung mit Type Casting bei Funktionen!

Rekursion

- Was ist eine rekursive Funktion?
- Beispiel: Berechnung der Faktorielle
- Beispiel: Bisektionsverfahren

Rekursive Funktion

- Funktion ist rekursiv, wenn sie sich selber aufruft
- natürliches Konzept in der Mathematik:
  - $n! = n \cdot (n-1)!$
- d.h. Rückführung eines Problems auf einfacheres Problem derselben Art
- Achtung:
  - Rekursion darf nicht endlos sein
  - d.h. Abbruchbedingung für Rekursion ist wichtig
  - z.B. $1! = 1$
- häufig Schleifen statt Rekursion möglich (später!)
  - idR. Rekursion eleganter
  - idR. Schleifen effizienter

Beispiel: Faktorielle

```
 1  #include <stdio.h>
 2
 3  int factorial(int n) {
 4    if (n <= -1) {
 5      return -1;
 6    }
 7    else {
 8      if (n > 1) {
 9        return n*factorial(n-1);
10      }
11      else {
12        return 1;
13      }
14    }
15  }
16
17  main() {
18    int n = 0;
19    int nfac = 0;
20    printf("n=");
21    scanf("%d",&n);
22    nfac = factorial(n);
23    if (nfac <= 0) {
24      printf("Fehleingabe!\n");
25    }
26    else {
27      printf("%d!=%d\n",n,nfac);
28    }
29  }
```


Bisektionsverfahren

- Gegeben: stetiges $f : [a,b] \to \mathbb{R}$ mit $f(a)f(b) \leq 0$
  - Toleranz $\tau > 0$
- Tatsache: Zwischenwertsatz $\Rightarrow$ mind. eine Nst
  - denn $f(a)$ und $f(b)$ haben versch. Vorzeichen
- Gesucht: $x_0 \in [a,b]$ mit folgender Eigenschaft
  - $\exists \widetilde{x}_0 \in [a,b] \quad f(\widetilde{x}_0) = 0$ und $|x_0 - \widetilde{x}_0| \leq \tau$
- Bisektionsverfahren = iterierte Intervallhalbierung
  - Solange Intervallbreite $|b-a| > 2\tau$
    - Berechne Intervallmittelpunkt m und $f(m)$
    - Falls $f(a)f(m) \leq 0$, betrachte Intervall $[a,m]$
    - sonst betrachte halbiertes Intervall $[m,b]$
  - $x_0 := m$ ist schließlich gesuchte Approximation
- Verfahren basiert nur auf Zwischenwertsatz
- terminiert nach endlich vielen Schritten, da jeweils Intervall halbiert wird
- Konvergenz gegen Nst. $\widetilde{x}_0$ für $\tau = 0$.

Beispiel: Bisektionsverfahren

```
 1  #include <stdio.h>
 2  #include <math.h>
 3
 4  double f(double x) {
 5    return x*x + exp(x) -2;
 6  }
 7
 8  double bisection(double a, double b, double tol){
 9    double m = 0.5*(a+b);
10    if ( b - a <= 2*tol ) {
11      return m;
12    }
13    else {
14      if ( f(a)*f(m) <= 0 ) {
15        return bisection(a,m,tol);
16      }
17      else {
18        return bisection(m,b,tol);
19      }
20    }
21  }
22
23  main() {
24    double a = 0;
25    double b = 10;
26    double tol = 1e-12;
27    double x = bisection(a,b,tol);
28
29    printf("Nullstelle x=%g\n",x);
30    printf("Funktionswert f(x)=%g\n",f(x));
31  }
```

Mathematische Funktionen

- Preprocessor, Compiler, Linker
- Object-Code
- Bibliotheken
- mathematische Funktionen

- `#define`
- `#include`

Preprocessor, Compiler & Linker

- Ein Compiler besteht aus mehreren Komponenten, die nacheinander abgearbeitet werden
- Preprocessor wird intern gestartet, *bevor* der Source-Code compiliert wird
  - Ersetzt Text im Code durch anderen Text
  - Preprocessor-Befehle beginnen *immer* mit `#` und enden *nie* mit Semikolon, z.B.
    - `#define text replacement`
      - in allen nachfolgenden Zeilen wird der Text `text` durch `replacement` ersetzt
      - geeignet, um Konstanten zu definieren
    - `#include file`
      - einfügen der Datei `file`
- Compiler übersetzt (Source-)Code in Object-Code
  - Object-Code = Maschinencode, bei dem symbolische Namen (z.B. Funktionsnamen) noch vorhanden sind
- Weiterer Object-Code wird zusätzlich eingebunden
  - z.B. Bibliotheken (= Sammlungen von Fktn)
- Linker ersetzt symbolische Namen im Object-Code durch Adressen und erstellt dadurch ein ausführbares Programm, sog. Executable

Bibliotheken & Header-Files

- (Funktions-) Bibliothek (z.B. math. Funktionen) besteht immer aus 2 Dateien
  - Object-Code
  - zugehöriges Header-File
- Im Header-File steht die Deklaration aller Fktn, die in der Bibliothek vorhanden sind
- Will man Bibliothek verwenden, muss man zugehöriges Header-File einbinden
  - `#include <header>` bindet Header-File `header` aus Standardverzeichnis `/usr/include/` ein,
    - z.B. `math.h` (Header-File zur math. Bib.)
  - `#include "datei"` bindet Datei aus *aktuellem* Verzeichnis ein (z.B. Downloads vom Internet)
  - idR. führt C-Compiler `#include <stdio.h>` von allein aus (in zugehöriger Bib. liegt z.B. `printf`)
- Ferner muss man den Object-Code der Bibliothek hinzulinken
  - Wo Object-Code der Bibliothek liegt, muss `gcc` mittels Option `-ℓ` (und `-L`) mitgeteilt werden
  - z.B. `gcc file.c -ℓm` linkt math. Bibliothek
  - Standardbibliotheken automatisch gelinkt, z.B. `stdio` (also keine zusätzliche Option nötig)

Mathematische Funktionen

- Deklaration der math. Funktionen in `math.h`
  - Input & Output der Fktn sind vom Typ `double`
- Wenn diese Funktionen benötigt werden
  - im Source-Code: `#include <math.h>`
  - Compilieren des Source-Code mit *zusätzlicher* Linker-Option `-ℓm`, d.h.

```
gcc file.c -o output -ℓm
```

  erzeugt Executable `output`
- Diese Bibliothek stellt u.a. zur Verfügung
  - Trigonometrische Funktionen
    - `cos`, `sin`, `tan`, `acos`, `asin`, `atan`, `cosh`, `sinh`, `tanh`
  - Exponentialfunktion und Logarithmus
    - `exp`, `log`, `log10`
  - Potenz- und Wurzelfunktion
    - `pow`, `sqrt` (wobei $x^y = \mathtt{pow(x,y)}$)
  - Absolutbetrag `fabs`
  - Rundung auf ganze Zahlen: `floor`, `ceil`

Elementares Beispiel

```
1  #include <stdio.h>
2  #include <math.h>
3
4  main() {
5    double x = 2.;
6    double y = sqrt(x);
7    printf("sqrt(%f)=%f\n",x,y);
8  }
```

- Precompiler-Befehle in 1, 2 ohne Semikolon
- Compilieren mit `gcc sqrt.c -ℓm`
- Vergisst man `-ℓm` ⇒ Fehlermeldung des Linkers

```
In function ‘main’
sqrt.c:(.text+0x24): undefined reference to ‘sqrt’
collect2: ld returned 1 exit status
```

- Output:

```
sqrt(2.000000)=1.414214
```


Arrays (=Felder)

- Vektoren, Matrizen
- Operator `[...]`
- Matrix-Vektor-Multiplikation
- Lineare Gleichungssysteme

Vektoren

- Deklaration eines Vektors $x = (x_0, \ldots, x_{N-1}) \in \mathbb{R}^N$:
  - `double x[N];` $\mapsto$ `x` ist `double`-Vektor
- Zugriff auf Komponenten:
  - `x[j]` entspricht x_j
  - Jedes `x[j]` ist vom Typ `double`
- Analoge Deklaration für andere Datentypen
  - `int y[N];` $\mapsto$ `y` ist `int`-Vektor
- ACHTUNG mit der Indizierung der Komponenten
  - Indizes $0, \ldots, N-1$ in C
  - idR. Indizes $1, \ldots, N$ in Mathematik
- Initialisierung bei Deklaration möglich:
  - `double x[3] = {1,2,3};` dekl. $x = (1,2,3) \in \mathbb{R}^3$
- Vektor-Initialisierung nur bei Deklaration erlaubt
  - Später zwingend komponentenweises Schreiben!
    - d.h. `x[0] = 1; x[1] = 2; x[2] = 3;` ist OK!
    - `x = {1,2,3}` ist verboten!

Beispiel: Einlesen eines Vektors

```
1  #include <stdio.h>
2
3  main() {
4    double x[3] = {0,0,0};
5
6    printf("Einlesen eines Vektors x in R^3:\n");
7    printf("x_0 = ");
8    scanf("%lf",&x[0]);
9    printf("x_1 = ");
10   scanf("%lf",&x[1]);
11   printf("x_2 = ");
12   scanf("%lf",&x[2]);
13
14   printf("x = (%f, %f, %f)\n",x[0],x[1],x[2]);
15 }
```

- Ausgabe `double` über `printf` mit Platzhalter `%f`
- Einlesen `double` über `scanf` mit Platzhalter `%ℓf`

Achtung: Statische Arrays

- Die Länge von Arrays ist statisch
  - nicht veränderbar während Programmablauf
  - $x \in \mathbb{R}^3$ kann nicht zu $x \in \mathbb{R}^5$ erweitert werden
- Programm kann nicht selbständig herausfinden, wie groß ein Array ist
  - d.h. Programm weiß bei Ablauf nicht, dass Vektor $x \in \mathbb{R}^3$ Länge 3 hat
  - Aufgabe des Programmierers!
- Achtung mit Indizierung!
  - Indizes laufen $0, \ldots, N-1$ in C
  - Prg kann nicht wissen, ob `x[j]` definiert ist
    - `x` muss mindestens Länge $j+1$ haben!
    - falsche Indizierung ist kein Syntaxfehler!
    - sondern bestenfalls Laufzeitfehler!
- Arrays dürfen nicht Output einer Funktion sein!
- Arrays werden mit Call by Reference übergeben!
- Dasselbe gilt für Matrizen bzw. allgemeine Arrays

Arrays & Call by Reference

```
1   #include <stdio.h>
2
3   void callByReference(double y[3]) {
4     printf("a) y = (%f, %f, %f)\n",y[0],y[1],y[2]);
5     y[0] = 1;
6     y[1] = 2;
7     y[2] = 3;
8     printf("b) y = (%f, %f, %f)\n",y[0],y[1],y[2]);
9   }
10
11
12  main() {
13    double x[3] = {0,0,0};
14
15    printf("c) x = (%f, %f, %f)\n",x[0],x[1],x[2]);
16    callByReference(x);
17    printf("d) x = (%f, %f, %f)\n",x[0],x[1],x[2]);
18  }
```

- Output:

```
c) x = (0.000000, 0.000000, 0.000000)
a) y = (0.000000, 0.000000, 0.000000)
b) y = (1.000000, 2.000000, 3.000000)
d) x = (1.000000, 2.000000, 3.000000)
```

- Call by Reference bei Vektoren!
- Erklärung folgt später ($\rightarrow$ Pointer!)

Falsche Indizierung von Vektoren

```
1   #include <stdio.h>
2   #define WRONG 1000
3
4   main() {
5     int x[3] = {0,1,2};
6
7     x[WRONG] = 43;
8
9     printf("x = (%d, %d, %d), x[%d] = %d\n",
10            x[0],x[1],x[2],WRONG,x[WRONG]);
11  }
```

- Zeile 2 definiert Konstante `WRONG`
  - Namen von Konstanten in Großbuchstaben
- Zeile 7, 9-10: Falscher Zugriff auf Vektor `x`
  - Trotzdem keine Fehlermeldung/Warnung vom Compiler!
  - Für korrekte Indizes sorgt der Programmierer!
- Output:

```
x = (0, 1, 2), x[1000] = 43
```

- Für `WRONG` klein $\Rightarrow$ i.a. keine Fehlermeldung
- Für `WRONG` groß genug $\Rightarrow$ Laufzeitfehler

Matrizen

- Matrix $A \in \mathbb{R}^{M \times N}$ ist rechteckiges Schema

$$A = \begin{pmatrix} A_{00} & A_{01} & A_{02} & \ldots & A_{0,N-1} \\ A_{10} & A_{11} & A_{12} & \ldots & A_{1,N-1} \\ A_{20} & A_{21} & A_{22} & \ldots & A_{2,N-1} \\ \vdots & \vdots & \vdots & & \vdots \\ A_{M-1,0} & A_{M-1,1} & A_{M-1,2} & \ldots & A_{M-1,N-1} \end{pmatrix}$$

  mit Koeffizienten $A_{jk} \in \mathbb{R}$
- zentrale math. Objekte der Linearen Algebra
- Deklaration einer Matrix $A \in \mathbb{R}^{M \times N}$:
  - `double A[M][N];` $\mapsto$ `A` ist `double`-Matrix
- Zugriff auf Komponenten:
  - `A[j][k]` entspricht A_{jk}
  - Jedes `A[j][k]` ist vom Typ `double`
- zeilenweise Initialisierung bei Deklaration möglich:
  - `double A[2][3] = {{1,2,3},{4,5,6}};`

    deklariert + initialisiert $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$
  - Nur bei gleichzeitiger Deklaration erlaubt, vgl. Vektoren

Allgemeine Arrays

- Vektor ist ein 1-dim. Array
- Matrix ist ein 2-dim. Array
- Ist `type` Datentyp, so deklariert
  - `type x[N];` einen Vektor der Länge N
  - Koeffizienten `x[j]` sind Variablen vom Typ `type`
- Ist `type` Datentyp, so deklariert
  - `type x[M][N];` eine $M \times N$ Matrix
  - `x[j]` ist Vektor vom Typ `type` (der Länge N)
  - Koeff. `x[j][k]` sind Variablen vom Typ `type`
- Auch mehr Indizes möglich
  - `type x[M][N][P];` deklariert 3-dim. Array
  - `x[j]` ist $N \times P$ Matrix vom Typ `type`
  - `x[j][k]` ist Vektor vom Typ `type` (der Länge P)
  - Koeff. `x[j][k][p]` sind Variablen vom Typ `type`
- etc.