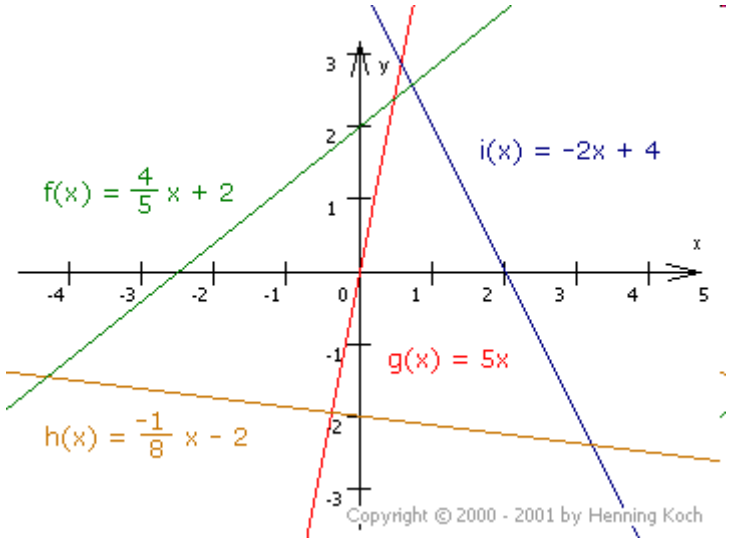
INF1
Funktionen: Einführung

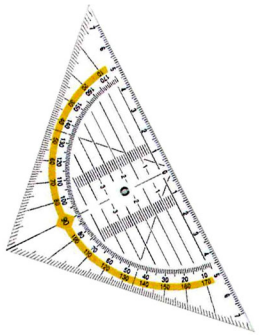


- Definition von Funktionen
- Funktionsaufruf
- Deklaration und Definition von Funktionen
- Variablen in Funktionen
- Rekursive Funktionen

Definition von Funktionen

Funktionen

- Zentrale Komponenten von C-Programmen
  - Erlauben die Strukturierung von Programmen
  - Programmteile, die mehrmals in einem Programm vorkommen, müssen nur einmal implementiert werden
  - Alle Funktionen in einem C-Programm müssen verschiedene Namen haben!
- *Libraries* (Programmbibliotheken) stellen Funktionen zur Verfügung
  - Die "Werkzeugkasten" für die Programmierung



```
...
sin
cos
tan
sqrt
abs
fabs
...
```

Funktionen

- Eine *Funktion* ist eine Anweisungsfolge, die unter einem Namen abgelegt ist und über ihren Namen aufrufbar ist
- Beim *Aufruf* (Verwendung) einer Funktion wird die aktuelle Anweisungsfolge verlassen und in diejenige der Funktion gesprungen

Funktion main

- Funktion, die beim Programmstart aufgerufen wird: *main*

```
int main (void) {
    printf("Hallo Welt!\n");
    return 0;
}
```

- Ein C-Programm besteht aus lauter Funktionen – eine davon ist die *main()*-Funktion

Funktionen in C

School of Engineering

- Bereits benutzt (neben main): *printf*, *scanf, fabs, ...*
- Bibliotheken (Libraries) stellen Funktionen zur Verfügung
- *printf* und *scanf* sind Bestandteil der Bibliothek *stdio.h*
  - werden mittels #include <stdio.h> innerhalb des eigenen Programms verfügbar
  - ".h" Dateien sind Sammlungen von Funktionen (eigentlich nur die Funktionsköpfe -> später)
- Man kann auch selber Funktionen definieren
  - in der eigenen Programmquelle
  - selber sog. Libraries erstellen (später)

Teile einer Funktionsdefinition

School of Engineering

- Beschreibung, Kommentar
- Typ der Funktion, d.h. der Typ des Rückgabewertes
- Name der Funktion
- Liste der Parameter, d.h. der Übergabewerte
- Anweisungsteil

Einführendes Beispiel

- Funktionen werden oft eingesetzt um grössere Programme zu strukturieren und somit übersichtlicher zu gestalten

Funktionsdefinition

```
double dreiecksFlaeche(double a, double b, double c) {
    double s = (a+b+c)/2;
    return sqrt(s*(s-a)*(s-b)*(s-c));
}


int main(void) {
      // Seiten a,b,c, einlesen
      double flaeche = dreiecksFlaeche(a,b,c);
      ...
}
```

Funktionsaufruf

Beschreibung

/** */ Kommentar für Doku

```
/**
 *  Berechne Fläche des Dreiecks
 *
 *  @param a    1. Seite des Dreiecks
 *  @param b    2. Seite des Dreiecks
 *  @param c    3. Seite des Dreiecks
 *
 *  @return Fläche des Dreiecks
 */

double dreiecksFlaeche(double a, double b, double c) {
    ...
}
```

- Beschreibung, Kommentar
- Typ der Funktion, d.h. der Typ des Rückgabewertes
- Name der Funktion
- Liste der Parameter, d.h. der Übergabewerte
- Anweisungsteil

Meist projektspezifische Konventionen für die Dokumentation

Wichtig: Konsistenz

Für automatische Doku-Generierung müssen bestimmte Formate eingehalten werden

Bsp: Doxygen

Programmierstil

- Zu jeder Funktion soll ein Beschreibungsteil in Form eines Kommentars existieren
- Beschrieben werden muss zumindest
  - Was die Funktion macht
  - Der Rückgabewert
  - Die Parameter (Wertebereiche)
- Oft sind weitere Angaben (Fehlerbehandlung, Autor, letzte Änderung, History, Seiteneffekte, usw.) notwendig

Typ der Rückgabewerts (= Typ der Funktion)

```
/**
*/
double dreiecksFlaeche(double a, double b, double c) {
    ...
    return flaeche;
}
```

Rückgabetyp

Rückgabe des Wertes

- Beschreibung, Kommentar
- Typ der Funktion, d.h. der Typ des Rückgabewertes
- Name der Funktion
- Liste der Parameter, d.h. der Übergabewerte
- Anweisungsteil

Typ des Rückgabewerts

Name der Funktion

```
/**
*/

double dreiecksFlaeche(double a, double b, double c) {
    ...
    return flaeche;
}
```

- Beschreibung, Kommentar
- Typ der Funktion, d.h. der Typ des Rückgabewertes
- Name der Funktion
- Liste der Parameter, d.h. der Übergabewerte
- Anweisungsteil

Konvention: Mit kleinem Anfangsbuchstaben

Parameterliste

```
/**
*/

double dreiecksFlaeche(double a, double b, double c) {
    ...
    return flaeche;
}
```

- Beschreibung, Kommentar
- Typ der Funktion, d.h. der Typ des Rückgabewertes
- Name der Funktion
- Liste der Parameter, d.h. der Übergabewerte
- Anweisungsteil

Datenübergabe mit Hilfe von Parametern

In der Funktion: *Formale Parameter*

Liste von Variablen-definitionen

Entsprechen lokalen Variablen in der Funktion (beim Aufruf der Funktion initialisiert)

Parameter und Rückgabewert

- Beliebig viele Parameter – nur ein Rückgabewert
- Eine Funktion kann auch keine Parameter haben
  - Hat in C eine Funktion keine Parameter ***kann*** sie mit ***void*** (=nichts) gekennzeichnet sein
    - *Bsp:* `main(void)`
  - Eine leere Parameterliste bedeutet in C: Keine Aussage über Anzahl und Typ der Parameter
- Eine Funktion kann auch keinen Rückgabewert haben
  - Eine Funktion ohne Rückgabewert ***muss*** den Rückgabetyp void haben
    - *Bsp:* `void foo(void)`

Anweisungsteil

```
  /**
  */

double dreiecksFlaeche(double a, double b, double c) {
    double s = (a+b+c)/2;
    return sqrt(s*(s-a)*(s-b)*(s-c));
}
```

- Beschreibung, Kommentar
- Typ der Funktion, d.h. der Typ des Rückgabewertes
- Name der Funktion
- Liste der Parameter, d.h. der Übergabewerte
- Anweisungsteil

Auch als *Implementierung* der Funktion bezeichnet

In einem Block

Wird beim Aufruf der Funktion ausgeführt

Ausführung der Funktion endet mit der return-Anweisung

Zusammengefasst

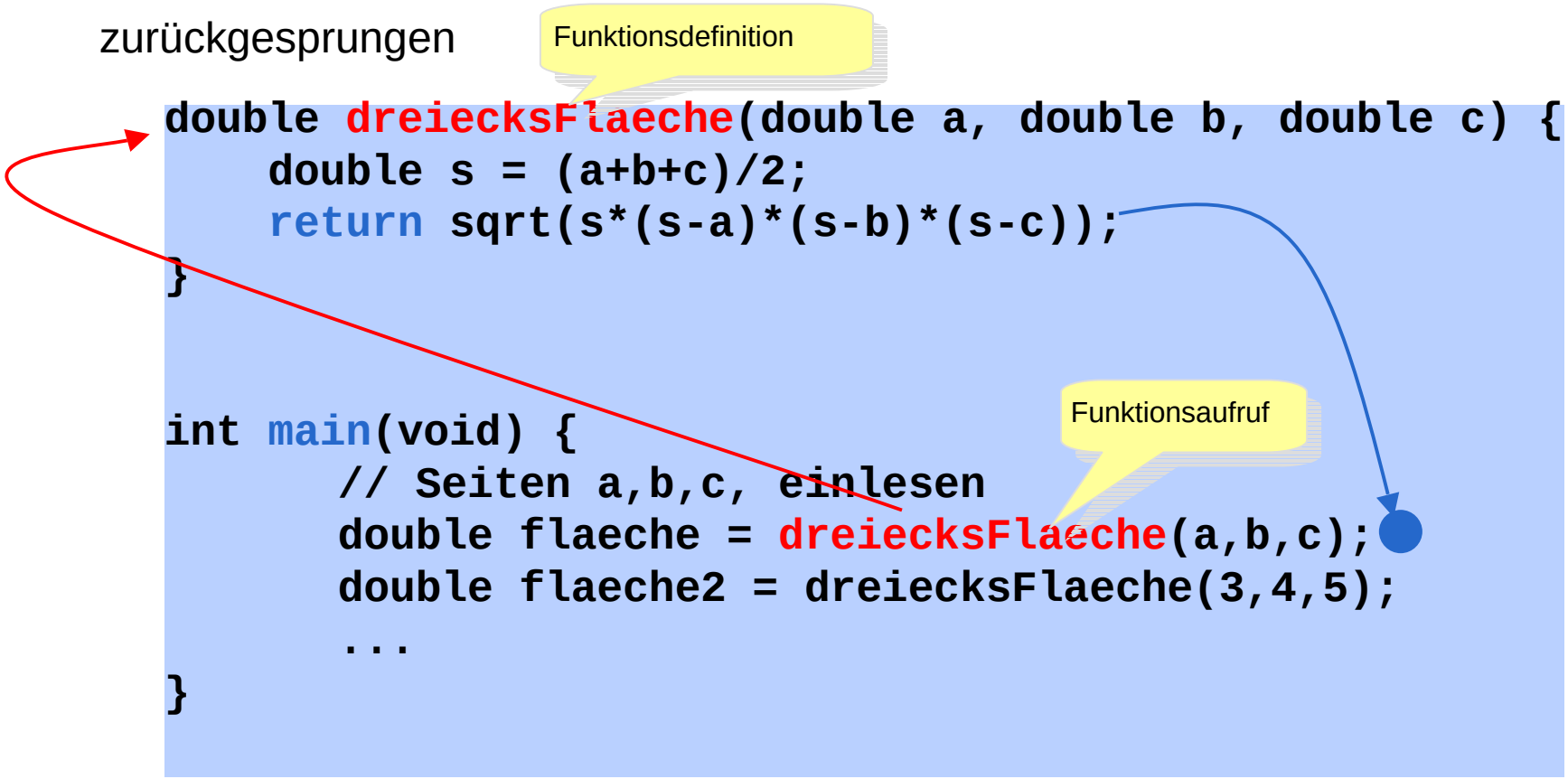
```
RückgabeTyp Funktionsname (Parameterliste) {    // Funktionskopf
                                                // Funktionsrumpf
    Lokale Variablen, die nur in dieser
    Funktion gültig sind

    Anweisungen                                  // Implementierung
}
```

- Der Funktionskopf gibt Auskunft darüber, was gemacht wird, die Implementierung wie es gemacht wird
  - Es ist möglich, eine Funktion zu verwenden, ohne deren Implementierung zu kennen
- Im Funktionsrumpf folgen Variablendeklarationen und die Implementierung

Funktionsaufruf

Funktionsaufruf

- Beim Aufruf wird zum **Anfang der Funktion** "gesprungen"
- Nach dem Beenden der Funktion, wird an die Stelle ***nach dem Aufruf*** zurückgesprungen




```
double dreiecksFlaeche(double a, double b, double c) {
    double s = (a+b+c)/2;
    return sqrt(s*(s-a)*(s-b)*(s-c));
}


int main(void) {
      // Seiten a,b,c, einlesen
      double flaeche = dreiecksFlaeche(a,b,c);
      double flaeche2 = dreiecksFlaeche(3,4,5);
      ...
}
```

Funktionsaufruf

- Erfolgt in irgendeiner Code-Sequenz ein Funktionsaufruf, so wird diese Code-Sequenz verlassen und in die Funktion gesprungen
- Nachdem die Funktion ihre Arbeit erledigt hat, wird der Return-Wert in den Ausdruck eingesetzt, der den Funktionsaufruf enthält
- Anschliessend erfolgt der Rücksprung auf die erste Instruktion nach dem Funktionsaufruf

Funktionsaufruf

- Name der Funktion, gefolgt von der Liste der *aktuellen Parameter (Argumente)*
- Die *formalen Parameter* der Funktion werden mit den aktuellen Parametern initialisiert
- Dazu müssen die Datentypen passen
  - übereinstimmen
  - Oder entsprechend (automatisch) konvertiert werden

Beispiel: Aufruf

```
// Definition
int isZero (double value) {
    if (fabs(value) < 0.001)
        return 1;
    else
        return 0;
}

// Aufruf
....
if (isZero(y)) { ... }
....
```

Formale Parameter

Aktuelle Parameter = Argumente

- Hier: Ein Parameter
  - Variable y
  - In der Funktion: value
  - Übergabe:
    double value = y;
- Ein Rückgabewert
  - Aufruf direkt in einer if-Anweisung verwendet

Weitere Beispiele

- Funktionsaufrufe können in Ausdrücken eingesetzt werden

```
i = fakultaet(5);
j = fakultaet(3+2*6);
k = fakultaet(fakultaet(3));
m = 2 * fakultaet(7);
```

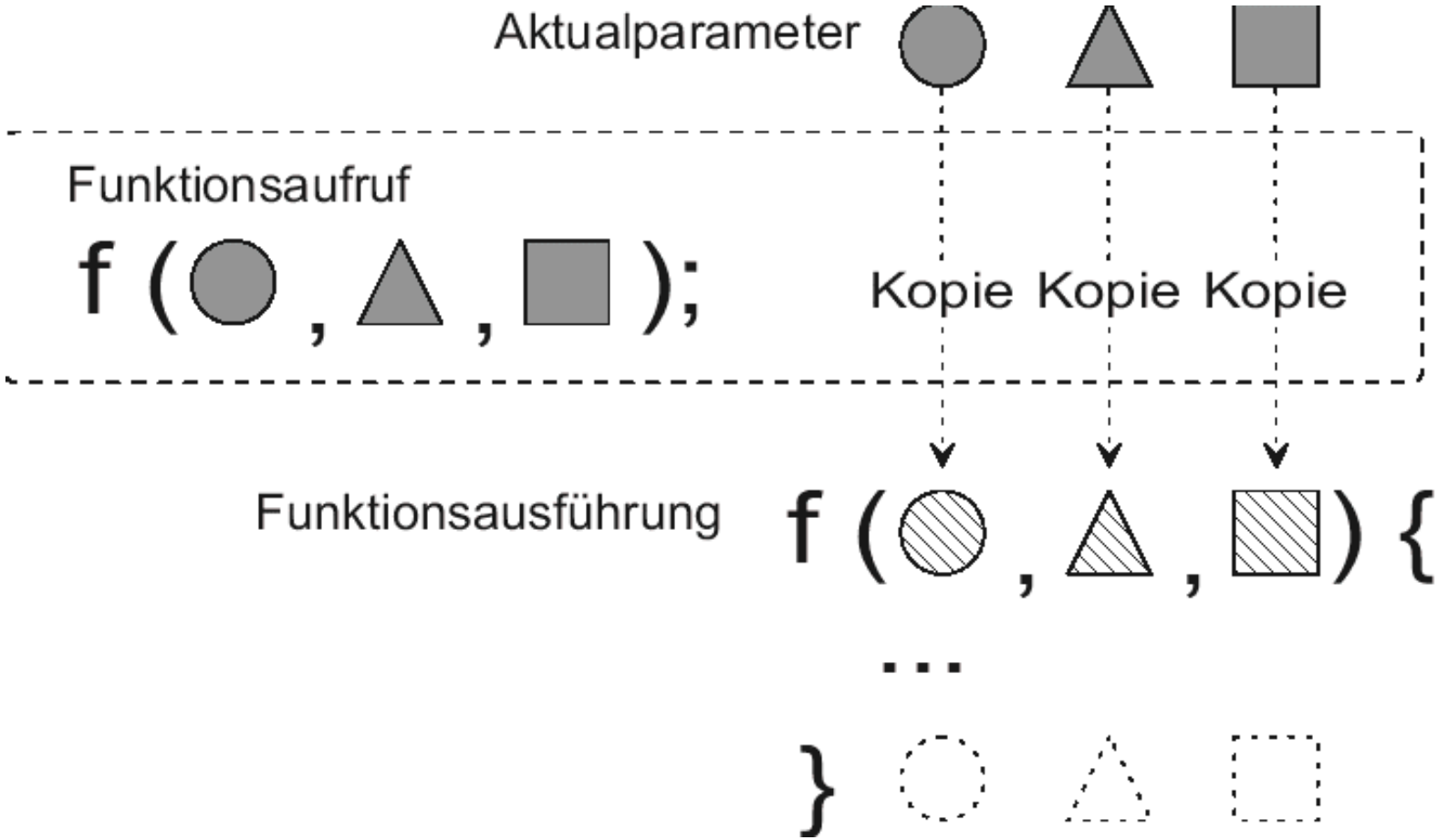
- Sie können auch aufgerufen werden ohne dass man den Funktionswert einer Variablen zuweist (wobei der Funktionswert verloren geht)

```
fakultaet(5+3*2);
```

Parameterübergabe (als Wert)

- Die formalen Parameter der Funktion werden mit den aktuellen Parametern initialisiert
- Die formalen Parameter entsprechen lokalen Variablen in der Funktion
- Wenn diese in der Funktion geändert werden, hat dies keine Auswirkungen „nach aussen“
- Man arbeitet in der Funktion quasi *mit Kopien* der übergebenen Daten
  - falls Ausdrücke übergeben werden, werden diese zuerst ausgewertet
- Man nennt dies: *Call by value*

Call by value



Call by value (Beispiel 1)

```c
#include <stdio.h>

void f (int j) {
    j+=1;
    printf("j in der Funktion: %d \n", j);
}

int main (void) {
    int i=2;
    printf("i vorher: %d \n", i);
    f(i);
    printf("i nachher: %d \n", i);
    return 0;
}
```

Call by value (Beispiel 2)

```
int max (int a, int b) {          // int a = 35;
   if (a > b)                     // int b = 37;
       return a;                  // lokale Kopien
   else
       return b;
}

int main (void) {
   int temp = max(35, 37);
   printf("%d\n", temp);
    printf("%d\n", max(35, 37));
   return 0;
}
```

Rückgabe des Funktionswerts

```
return <Ausdruck>;
```

- Ausdruck nach *return*-Anweisung muss Rückgabetyp entsprechen (oder automatisch umgewandelt werden können)

```
double dreiecksFlaeche(double a, double b, double c) {
    double s = (a+b+c)/2;
    return sqrt(s*(s-a)*(s-b)*(s-c));
}
```

Rückgabe des Funktionswerts

- Die *return* Anweisung muss nicht am Ende der Funktion stehen
- Durch die *return* Anweisung wird die Funktion sofort beendet

```
int even (int n) {
    if (n % 2 == 0) return 1;
    else return 0 ;
}
```

Mehrere return-Anweisungen in einer Funktion erlaubt aber können zu schlecht lesbarem Code führen....

```
int even (int n) {
   return (n % 2 == 0);
}
```

Rückgabe des Funktionswerts

- Oft – besonders in der *main*-Funktion – wird der Rückgabewert auch dazu benutzt, einen Fehlerwert-/Code zurückzugeben
- Konvention: Rückgabewert von 0 steht für das erfolgreiche Ablaufen der Funktion


```
int main(void){
   return EXIT_SUCCESS;
}
```

- Andere Werte stehen für bestimmte Fehler, die bei der Ausführung aufgetreten sind

```
int main(void){
   return EXIT_FAILURE;
}
```

Funktionen ohne Rückgabewert

- In diesem Fall wird als Typ *void* angegeben
- Sie werden als Anweisungen aufgerufen
- Sie sind nicht in Ausdrücken verwendbar

Funktionen ohne Rückgabewert

```
void printInfo (int i) {
  if (i%2==0)
    printf("%d ist gerade.\n" , i);
  else
    printf("%d ist ungerade.\n" , i);
}

int main (void) {
  int zahl=42;
  printInfo(zahl);        /* "42 ist gerade."    */
  printInfo(zahl/2);      /* "21 ist ungerade."  */
  printInfo(3*4+7-9/3);   /* "16 ist gerade."    */
  return 0;
}
```

Deklaration und Definition von Funktionen

Reihenfolge: Definition, Aufruf

Problem?

- Funktion wird erst nach dem Aufruf definiert
- Beim Übersetzen des Funktionsaufrufs kennt der Compiler die Funktion noch nicht

```
#include <stdio.h>

int main (void) {
  int x, y;
  printf("1. Zahl: ");
  scanf("%d", &x);
  printf("2. Zahl: ");
  scanf("%d", &y);
  printf("Maximum: %d\n", max(x, y));
}

int max (int a, int b) {
  if (a >= b) {
    return a;
  }
  return b;
}
```

(Voraus-) Deklaration von Funktionen

- Eine Funktion kann erst aufgerufen werden, wenn sie zuvor *deklariert* wurde
- Die *Definition* der Funktion kann auch noch später erfolgen

```
#include <stdio.h>

int max (int a, int b);

int main (void) {
  int x, y;
  printf("1. Zahl: ");
  scanf("%d", &x);
  printf("2. Zahl: ");
  scanf("%d", &y);
  printf("Maximum: %d\n", max(x, y));
}

int max (int a, int b) {
  if (a >= b) {
    return a;
  }
  return b;
}
```

(Voraus-) Deklaration von Funktionen

- Auch eine (*Voraus-) Deklaration* ist möglich
  - Diese enthält keinen Anweisungsteil
  - Wird auch als *Prototyp* der Funktion oder *explizite Deklaration* bezeichnet

- Anwendung:
  - Zwei Funktionen, die sich gegenseitig aufrufen
  - Erstellen von Bibliotheken:
    - *".h" Files sind nichts anderes als Sammlungen von Deklarationen, z.B. "stdio.h"*

(Voraus-) Deklaration und Definition

outlook

- Man kann (eigene) Vorausdeklarationen in ".h" Files zusammenfassen
- und dann in (andern) Programmen verwenden
  - der Compiler kennt nun alle benötigten Funktionen

Definition in einer eigenen Datei. Eine Header-Datei, z.B. „funk1.h“ enthält die Deklaration von funk1, und in der Quelldatei wird dann funk1definiert.

```
// Datei funk1.h:
int funk1(int i); // Deklaration
```

```
// Datei funk1.cpp:
#include "funk1.h"
int funk1(int i)  // Definition
{ ... }
```

In der Datei, in der funk1 benutzt wird, wird funk1.h eingebunden, d.h. die Deklaration von funk1:

```
// Datei main.cpp:
#include “funk1.h“
...
int main() {
  ... int rueck=funk1(17); ...
}
```

main.cpp und funk1.cpp müssen nun beide übersetzt werden und dann mit dem *Linker* verbunden werden.

(Voraus-) Deklaration von Funktionen

outlook

```
int funktA (int abstand, int anzahl);
int funktA (int, int);
```

- Parameternamen können bei der Vorausdeklaration weggelassen werden
  - oft erleichtern sie aber das Verstehen der Funktion
- Für den Compiler reicht die Deklaration, um Aufrufe in anderen Funktionen zu verarbeiten
- Die Definition der Funktion kann dann in der gleichen oder aber auch in einer anderen Datei erfolgen

Variablen in Funktionen

Lokale Variablen

School of Engineering

- Innerhalb eines Blocks oder in einer Funktion können *lokale Variablen* eingeführt werden
- Diese gelten nur innerhalb der Funktion oder des Blocks
- Sollten am Anfang eines Blocks oder einer Funktion stehen (in älteren C Versionen nur dort erlaubt)

```
double dreiecksFlaeche(double a, double b, double c) {
    double s = (a+b+c)/2;
    return sqrt(s*(s-a)*(s-b)*(s-c));
}
```

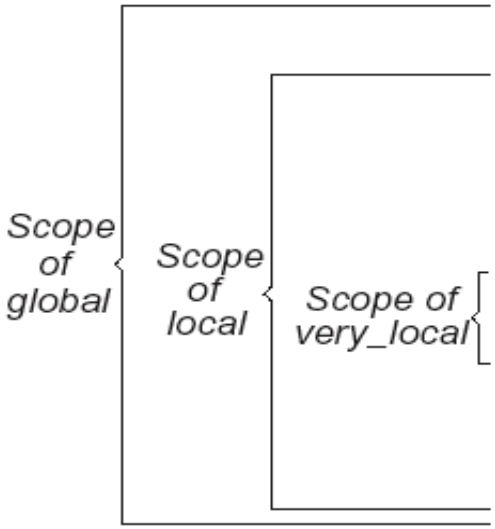
Globale Variable

- Variablen, die ausserhalb einer Funktion deklariert werden sind *global*
- Ihr Gültigkeitsbereich ist das ganze Programm

```
double s;
double dreiecksFlaeche(double a, double b, double c) {
    s = (a+b+c)/2;
    return sqrt(s*(s-a)*(s-b)*(s-c));
}
```

- Nach Möglichkeit globale Variablen vermeiden

Gültigkeitsbereich




```
int global;                          // a global variable
main()
{
     int  local;                     // a local variable

     global = 1;                     // global can be used here
     local = 2;                      // so can local

     {                               // beginning a new block
         int  very_local             // this is local to the block

         very_local = global+local;
     }

        // We just closed the block
        // very_local can not be used
}
```

Gültigkeitsbereich (Scope)

- Dateiebene
  - Funktionsebene
    - *Blockanweisung*

- Es wird immer die Variable bevorzugt genutzt, die im selben Gültigkeitsbereich wie der aktuelle Programmteil liegt
- Falls im aktuellen Programmteil keine Variablen dieses Namens definiert ist, wird im nächst höheren gesucht und so fort

Gültigkeitsbereich

Lokale Variable

count überdeckt

die globale Variable

count in diesem

Bereich

```
int total, count;
int main (void) {
    total = 0;
    count = 0; /*set global counter*/
    {
        int count; /*a local counter*/
        count=0;
        if (count % 2 == 0) {
            total += count;
            ++count;
        }
    }
    ++count;
    return 0;
}
```

Gültigkeitsbereich

- Wird innerhalb eines Blocks eine Variable mit dem gleichen Namen definiert wie eine globale Variable, so *verdeckt* die neue Variable die globale Variable

```
{ int k;
  k = 4;
  {double k;
      k = 4.2;
  }
}
```

Ebenso können lokale Variablen eines übergeordneten Blocks verdeckt werden

- Dies führt schnell zu unleserlichem Code

Gültigkeitsbereich (Scope)

```
int i; /* Global Scope */

int sum (int num1, int num2) {
  i = 1;
  int result = num1 + num2; /* Function Scope */
  return result;
}

int main (void) {
  int s = sum(2,4);
  s += i;
}
```

Lokale Variablen – Beispiel 1

```
int i=4;

void funktA (void) {
  printf("%d\n", i);
}

void funktB (void) {
  int j=7;
  printf("%d\n", j);
}
```

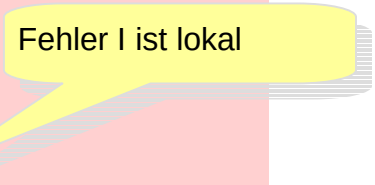
Globale Variablen – Beispiel

```
#include <stdio.h>

int max = 0;        /* Definition globale Variable */
void checkMax(int a);

int main (void) {
  int a;
  while (1) {
    printf("a = ");
    scanf("%d", &a);
    checkMax(a);
    printf("Maximum bis jetzt: %d\n", max);
  }
}

void checkMax (int a) {
  if (a > max) {
    max = a;
  }
}
```

Globale Variablen vermeiden

Lokale Variablen – Beispiel 2

```
void funktA (void);

void main (void) {
  int i=12; /* lokale Variable in
 main */
  funktA();
}

void funktA (void) {
   printf("%i\n", i);
}
```



Speicherklasse

- Lokale Variablen sie sind nur temporär innerhalb der Funktion gültig (werden implizit der Speicherklasse *auto* zugeordnet,)
- Durch Angabe der *Speicherklasse static* erhält eine lokale Variable eine längere Lebensdauer ("permanent")
- i.e. eine statische Variable ist eine lokale Variable, die nur einmal Speicherplatz zugewiesen bekommt und diesen behält.

Statische lokale Variablen

```
  /* Speicherplatz von n bleibt über mehrere
     Funktionsaufrufe erhalten, die
     Initialisierung wird nur einmal ausgeführt */

void zaehler (void) {
  static int n = 1;
  printf("Dies ist Aufruf Nummer %d", n++);
}
```

- Das könnte man auch mit einer globalen Variablen erreichen (was aber unschön ist, da die Variable dann auch von anderen Funktionen verwendet werden kann)

Statische lokale Variablen

```
#include <stdio.h>
int checkMax(int a);

int main (void) {
  int a;
  while (1) {
    printf("a = ");
    scanf("%d", &a);
    printf("Maximum bis jetzt: %d\n",
           checkMax(a));
  }
}

int checkMax (int a) {
  static int max = 0;  /* Deklaration     */
  if (a > max) {       /* statische       */
    max = a;           /* lokale Variable */
  }
  return max
}
```

Scope and Storage Class

Declared	Scope	Storage Class	Initialized
Outside all blocks	**Global**	**Permanent**	**Once**
Static outside all blocks	Global	Permanent	Once
Inside a block	**Local**	**Temporary**	**Each time block is entered**
Static inside a block	Local	Permanent	Once

Rekursive Funktionen

Rekursion

- Funktionen in C können sich selbst aufrufen
- Dies ist sinnvoll, wenn
  - Das Problem durch den rekursiven Aufruf einfacher wird
  - Ein Abbruchkriterium existiert: Das Problem ist so einfach, dass kein rekursiver Aufruf mehr erforderlich ist

Rekursion: Beispiel

```
int fakultaet (int n) {
    if (n==0) return 1;
    else return n * fakultaet(n-1);
}
```

- Die Fakultät von 0 ist 1
- Die Fakultät von n kann auf ein einfacheres Problem zurückgeführt werden: Fakultät von n-1
- Was geschieht, wenn die Funktion für eine negative Zahl aufgerufen wird?

Unit Tests

outlook

Unit Tests

outlook

- Programm wird auf Stufe Einheit (Unit) getestet
- Guter Stil:
  - Programme immer zu testen
  - Bei Änderungen Tests nochmals durchlaufen lassen
- Tests müssen automatisiert und standardisiert werden
- Testframeworks
  - Java -> JUnit
  - C# -> NUnit
- C -> CUnit
  - leider zu den obigen beiden sehr unterschiedlich und relativ komplex
  - deshalb eigenes kleines Framework in assertions.c

Assertions - Basis für Abgabeserver

outlook

assertEquals(char* test, int expected, int value)

- überprüft ob value dem expected Wert entspricht
- Zeigt Error (und test-Meldung) an, andernfalls
- Daneben eine Reihe von anderen Assertions, z.B.
  - assertTrue(char* msg, bool expected)
  - assertEquals(char* msg, int expected, int value)
  - assertDoubleEquals(char* msg, double expected, double value)
  - assertDoubleEqualsEps(char* msg, double expected, double value, double eps) {

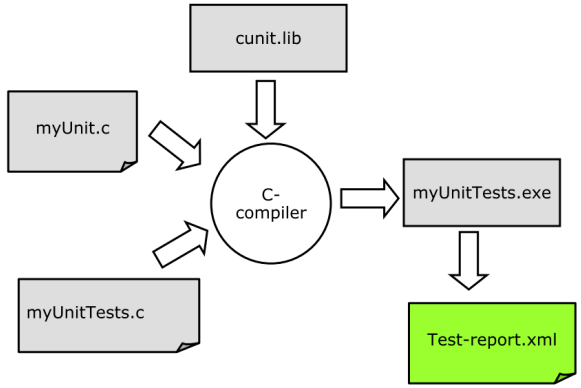
```
int foo() …
double bar() …
void testFoo() {
   assertEquals("testFoo",42,foo());
}
void testBar() {
   assertDoubleEquals("testBar",3.14,bar());
}
```

pro Funktion eine analoge testFunktion

CUnit: Standard Framework for CUnit Tests

outlook

- Struktur



- Beispiele von Assertions

```
CU_ASSERT_EQUAL(rectangularTriangle,
classifyTriangle(13,12,5) );
int actual_val;
CU_ASSERT(stringToInt("+0",&actual_val));
CPPUNIT_ASSERT_EQUAL(0, actual_val );
char* argv4[4]=  {programName,"1","1","2"};
CU_ASSERT_EQUAL(string( "Isosceles Triangle"),
                string(checkTriangle(4,argv4)));
```

- Beispiel von Test Driver File

```
int RunAllTests(void)
{
  CU_pSuite pSuite = NULL;
  pSuite=getTriangleSuite();

   CU_set_output_filename("TriangleTest");
   CU_list_tests_to_file();
   CU_automated_run_tests();
}
```

http://cunit.sourceforge.net/doxdocs/group__Framework.html

http://people.cs.aau.dk/~bnielsen/TOV07/lektioner/cunit-intro-07.pdf

https://netbeans.apache.org/kb/docs/cnd/c-unit-test.html

Anderer Entry Point für ein Program

outlook

- Übersetzte Program mit folgenden Optionen

```
gcc -Wl,-e_testAll t.c -o t.exe
```

"_" je nach Compiler Version/ Option

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

void testAll() {
        printf("testAll\n");
        testFoo();
        testBar();
        exit(0);
}
void main() {
        printf("main\n");
}
```

Programm muss mit exit beendet werden

Noch Fragen?