Einführung in die Programmiersprache C++

Thomas Wiemann
Institut für Informatik
AG Wissensbasierte Systeme

Letzte Vorlesung

- Datentypen
- Array
- Operatoren

```
if(x >= -1.5 && x <= 1.5) return 0;
```

Pitfall
A Programm that compiles, links und runs but does something different than you expect.

~~`if (-1.5 <= x <= 1.5) return 0;`~~

Gliederung

Umwandlung von Datentypen

- Umwandlung von Daten unterschiedlichen Typs:

```
int i = 10;
float f = (float) i;
double d = (double) i;
```

- `(float)` etc. sind so genannte Type-Conversion-Operatoren
- Umwandlung wird als „Casting“ bezeichnet
- Der Compiler macht dies bei einigen Datentypen automatisch (Warning)
- Daher: Typumwandlungen immer von Hand vornehmen!!

```
int i, j;
double d;
i = 3;
j = 4;
d = i / j;                  /* d = ? */
                            /* 0 */
d = ((double) i) / j;       /* d = ? */
                            /* 0.75 */
```

Ausdrücke, Statements, Kommentare

- `i + 2 * j` ist ein Ausdruck (hat einen Wert)
- `i = j * k;` ist ein Statement
  - Endet mit einem Semikolon
  - Ist auch ein Ausdruck (Wert ist der Wert von `i`)
- `i = j = k = t = 0;` ist möglich
  - äquivalent zu `i = (j = (k = (t = 0)));`
  - und nicht zu `(((i = j) = k) = t) = 0;`
- `/* Dies ist ein Kommentar */`
-
```
/*
        Kommentare können
        mehrere Zeilen lang sein
*/
```
- `// Dies ist kein C-Kommentar!`

Kommentare (1)

- **Kommentare sind sehr wichtig!**
- Sinn:
  - Erklärungen, wie Funktionen benutzt werden
  - Erklärungen, wie Funktionen funktionieren
  - Erklärung von allem, was nicht offensichtlich ist
- Wer liest Kommentare?
  - Jeder, der den Code modifiziert
  - Der Programmierer in ein paar Wochen/Monaten/Jahren

Kommentare (2)

```
/*
 * area: finds area of circle
 * arguments: r: radius of circle
 * return value: the computed area
 */
double area(double r) {
    double  pi = 3.1415926;
    return (pi * r * r);
}
```

- Variablennamen
  - sprechende Bezeichner helfen:

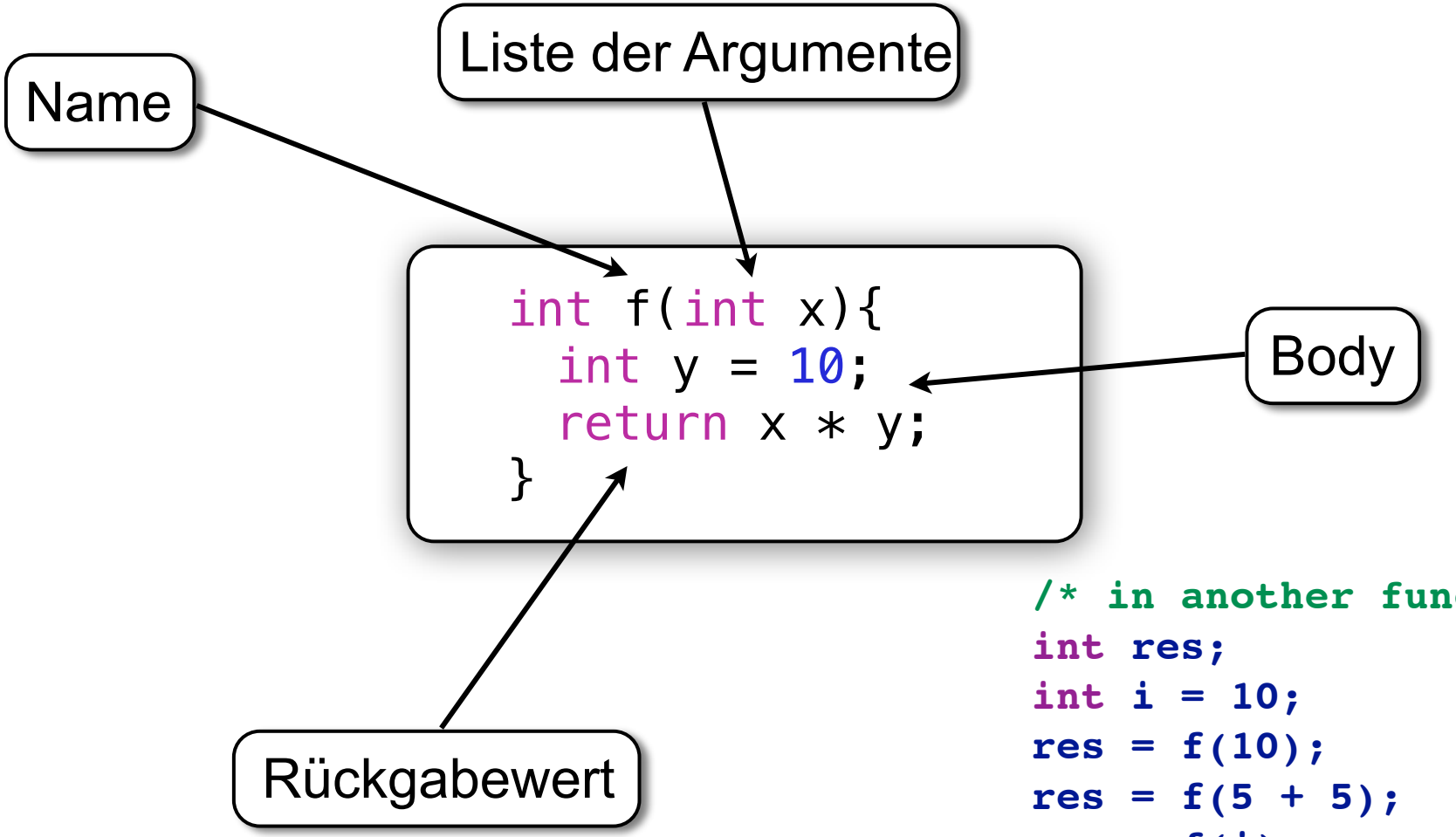
```
double x;          /* what does x mean? */
double distance;   /* better */
```

  - ... sind aber nicht immer notwendig:

```
int loop_index;    /* bad */
int i;             /* good */
```

Funktionen (1)

- Funktionen nehmen Argumente und geben einen wert zurück




```
/* in another function */
int res;
int i = 10;
res = f(10);
res = f(5 + 5);
res = f(i);
res = f(i*5 + i/2);
```

C-Funktionen (2)

- Nicht alle Funktionen geben einen Wert zurück

```
void print_number(int i) {
  printf("number is: %d\n", i);
}
```

- Rückgabewert ist `void` (nichts wird zurück gegeben)

```
void print_number(int i) {
  printf("number is: %d\n", i);
  return /* unnecessary */
}
```

- Aufruf:

```
/* in another function */
int i = 10;
print_number(20);
print_number(i);
```

C-Funktionen (3)

- Nicht alle Funktionen benötigen Argumente:

```
int five(void) {
  return 5;
}
```

- Aufruf von Funktionen ohne Argumente:

```
int value;
value = five();
```

- `value` hat den wert 5
- Beachte () beim Fuktionsaufruf!
- Optional:

```
int value;
value = five(void);
```

Funktionsprototypen

- Jede Funktion muss vor ihrer Verwendung definiert sein

```
int foo(int x) { ... }
int bar(int y)
{
    return 2 * foo(y);
}
```

Prototpen stehen in den Header files!

- `bar` kann nicht vor `foo` vorkommen
- Compiler sind doof
- Funktionsprototypen enthalten die Signatur der Funktion

```
int foo(int x);          /* no body yet. */
int bar(int y);          /* no body yet. */
[…]
int bar(int y)
{
   return 2 * foo(y);    /* OK */
}
/* Define 'foo' later. */
```

C-Deklarationen (1)

- Typdeklarationen befinden sich am Anfang einer Funktion
- In C benötigt man am Beginn jeder Funktion eine Deklaration der lokalen Variablen
- Richtig:

```
int foo(int x) {
  int y;                /* type declaration */
  y = x * 2;
  return y;
}
```

- Falsch:

```
int foo(int x) {
  int y;                /* type declaration */
  y = x * 2;            /* code */
  int z = y * y;        /* type decl. after code */
  return z;
}
```

C-Deklarationen (2)

- Variablendeklarationen können lokal und global sein
  - Lokal: innerhalb einer Funktion
  - Global: außerhalb von Funktionen
    - Von jeder Funktion aus zugreifbar

```
int x;                  /* global variable          */
int y = 10;             /* initialized global var.*/

int foo(int z) {
  int w;                /* local variable           */
  x = 42;               /* assign to global var.    */
  w = 10;               /* assign to local var.     */
  return (x + y + z + w);
}
```

- Globale Variablen sollten vermieden werden, da sie von jeder Funktion geändert werden können
- Debugging wird schwierig
- Oft sind globale Variablen nicht nötig, aber praktisch

Übergabe von Feldern an Funktionen (1)

- Was passiert bei folgendem Code-Fetzen:

```
void foo(int i) {
    i = 42;
}

/* later... */
int i = 10;
foo(i);  /* What is i now? */
```

- Der Wert von `i` wird in die Funktion *kopiert*
- Übergabe einer Variablen an eine Funktion ändert nicht den Wert außerhalb der Funktion - „**Call by Value**“
- Bei Arrays ist dies nicht der Fall
  - eigentlich nicht wirklich, aber man benötigt das Konzept der Pointer um alles zu erklären
  - kommt später

Übergabe von Feldern an Funktionen (2)

- Übergebene Arrays können modifiziert werden:

```
void foo(int arr[]) {
    arr[0] = 42; /* modifies array */
}

/* later... */
int my_array[5] = { 1, 2, 3, 4, 5 };
foo(my_array);
printf("%d\n", my_array[0]);
```

- Die erste Array-Dimension braucht nicht angegeben zu werden:

```
void foo2(int arr[5])    /* same as arr[] */
{
    arr[0] = 42;
}
```

C-Strings

- In C sind Strings Arrays vom Typ `char`
- Woher wissen String-Funktionen, wann ein String zu Ende ist?

```
char string1[] = "Dies ist ein sehr langer String";
char string2[] = "kurzer String";
printf("string1: %s\n", string1);
printf("string2: %s\n", string2);

/* kopieren von Strings */
strcpy(string1, string2);
printf("string1: %s\n", string1);
```

- String enden immer auf „0“ (Null-Terminierung)
- String-Helfer-Funktionen befinden sich in `<string.h>`

Bedingungen (2)

- Achtung!

```
int a = 0;
if (a = 10)                          /* always true! */
{
    printf("a equals 10\n");
}
else
{
    printf("a doesn't equal 10\n");
}
```

- Hier sollte wohl folgendes stehen:

```
int a = 0;
if (a == 10)                         /* not always true! */
{
    printf("a equals 10\n");
}
else [snip]
```

besser noch:
`if (10 == a)...`

Bedingungen (3)

- `else`-Klausel ist optional
- `else if` für mehrere Fälle:

```
int a = 0;
if (a == 10) {
    printf("a equals 10\n");
} else if (a < 10) {
    printf("a is less than 10\n");
} else {
    printf("a is greater than 10\n");
}
```

Bedingungen (4)

- Bei Vergleichen auf `int`s besser `switch` verwenden:

```
void do_stuff(int i) {
  switch (i) {
    case 0:
      printf("zero\n");
      break;
    case 1:
      printf("one\n");
      break;
    default:
      printf("…\n");
      break;
  }
}
```

Bedingungen (5)

- Häufiger Fehler:

```
switch (i) {
    case 0:  /* Start here if i == 0 */
        printf("zero\n");
        /* oops, forgot the break */
    case 1:  /* "fall through" from case 0 */
        printf("one\n");
}
```

- Kann Sinn machen, ist aber in der Regel nur selten erwünscht

Bedingungen (6)

- C besitzt einen trinären Operator (drei Argumente)
  - `?` `:` (question mark) Operator
- Wie ein `if`-Statement, das einen Wert zurück gibt

```
int i = 10;
int j;
j = (i == 10) ? 20 : 5; /* note 3 args */
/* "(i == 10) ? 20 : 5" means:
 * "If i equals 10 then 20 else 5." */
```

- Wird nicht häufig benutzt
- Kann verwirren

Schleifen (1)

- Zählschleifen

```
for (<initialization>;
     <test>;
     <increment>)
{ <body> }
```

- Beispiel:

```
int i;
for (i = 0; i < 10; i++)
{
   printf("cowabunga!!!\n");
}
```

Schleifen (2)

- `while`-Schleifen

```
int a = 10;
while (a > 0)
{
    printf("a = %d\n", a);
    a--;
}
```

- Hier wird das Abbruchkriterium direkt am Schleifenkopf geprüft, d.h. unter Umständen wird die Schleife nicht durchlaufen
- `do-while`-Schleifen

```
int a = 10;
do{
  printf("a = %d\n", a);
  a--;
} while(a > 0);
```

`break` und Co.

- **`break`** beendet die umschließende Schleife
- um aus geschachtelten Schleifen auszusteigen, kann man **`goto`** benutzen

```
for (i = 0; i < m; i++) {
  for (j = 0; j < n; j++) {
      /* code ... */
      goto out;          /* something went wrong */
  }
}
out:  /* a label */
/* continue here */
```

nicht verwenden!

- **`continue`** beendet die aktuelle Iteration der Schleife

```
int i;
for (i = 0; i < 100; i++) {
    if (i % 2 == 0) continue;
    else printf("i = %d\n", i)
}
```

Zeige ungerade Zahlen

Bemerkungen zur Syntax

- Die Bodies von **`for`**, **`while`**, **`do`**/**`while`**, **`if`**, **`if`**/**`else`** können einfache Anweisungen oder Blöcke von Programmcode sein
- Besser ist es, immer einen Block aufzumachen, um späteres Einfügen von Code zu erleichtern:

```
int i;
for (i = 0; i < 100; i++) {
    if (i % 2 == 0) continue;
    else printf("i = %d\n", i)
}
```

```
int i;
for (i = 0; i < 100; i++) {
    if (i % 2 == 0) {
      continue;
    } else {
      printf("i = %d\n", i)
    }
}
```

- Rechts ist einfacher zu verwalten und zu lesen

Ein- und Ausgabe mit `scanf` und Co. (1)

- C stellt Ein- und Ausgabe „Files“ zur Verfügung
  - `stdin` zur Eingabe von der Konsole
  - `stdout` zur Ausgabe auf die Konsole
  - `stderr` zur Ausgabe von Fehlern
    - meistens auch auf die Konsole
- All dies ist im Header `<stdio.h>` definiert
- `printf()` - Ausgabe auf stdout
- `scanf()` - einlesen von stdin
- Allgemeiner:
  - `fprintf()` - Ausgabe in eine beliebige Datei
  - `fscanf()` - Eingabe aus einer beliebigen Datei
- `fprintf()` mit `stderr` wird benutzt um Fehler auszugeben:

```
fprintf(stderr, “something went wrong!\n");
```

Ein- und Ausgabe mit `scanf` und Co. (2)

- `scanf()` liest analog zu `printf()`von der Konsole
- „komische“ Syntax

```
char s[100];
scanf(“%99s“, s);
```

- D.h. lese in String s nicht mehr als 99 Zeichen
- `scanf()` ändert den Wert der Variablen in der Liste der Argumente
- Die Funktion `scanf()` hat einen Integer-Rückgabewert
  - Wenn `scanf()` erfolgreich war, dann wird die Anzahl der Items zurückgegeben, die gelesen wurden
  - Wenn die Eingabe nicht zur Verfügung steht, wird `EOF` zurückgegeben
  - `EOF` ist auch im `<stdio.h>`-Header definiert

Ein- und Ausgabe mit `scanf` und Co. (3)

- Beispiel:

```
int val;
int result;
result = scanf("%d", &val);
if (result == EOF)
{
    /* print an error message */
}
```

- Beachte das `&val` in `scanf()`:

```
int val, result;
result = scanf("%d", &val);
```

- Eine genauere Erklärung folgt später, wenn wir über Pointer reden
- Regel: `&` ist notwendig, wenn `ints`, `doubles` etc. gelesen werden, aber nicht bei Strings

Kommandozeilenparameter

- Programmname und Argumente:

```
% myprog -param1 -param2 ...
```

- Erinnerung:
  - Strings sind Arrays von Charakters (`char[]`)
  - Auch geschrieben als `char*` (Pointer-Einführung sehr bald)
- Kommandozeilenparameter werden vom System an die main-Funktion übergeben

```
main(int argc, char* argv[])
```

- `argc` enthält die Anzahl der übergebenen Parameter
- `argv` enthält ein Array von Strings mit den Übergebenen Parametern
- `char* argv[]` ist zu lesen als `(char*) argv[]`
- `argv[][]` ist leider nicht erlaubt

Kommandozeilenparameter (2)

- Beispiel:

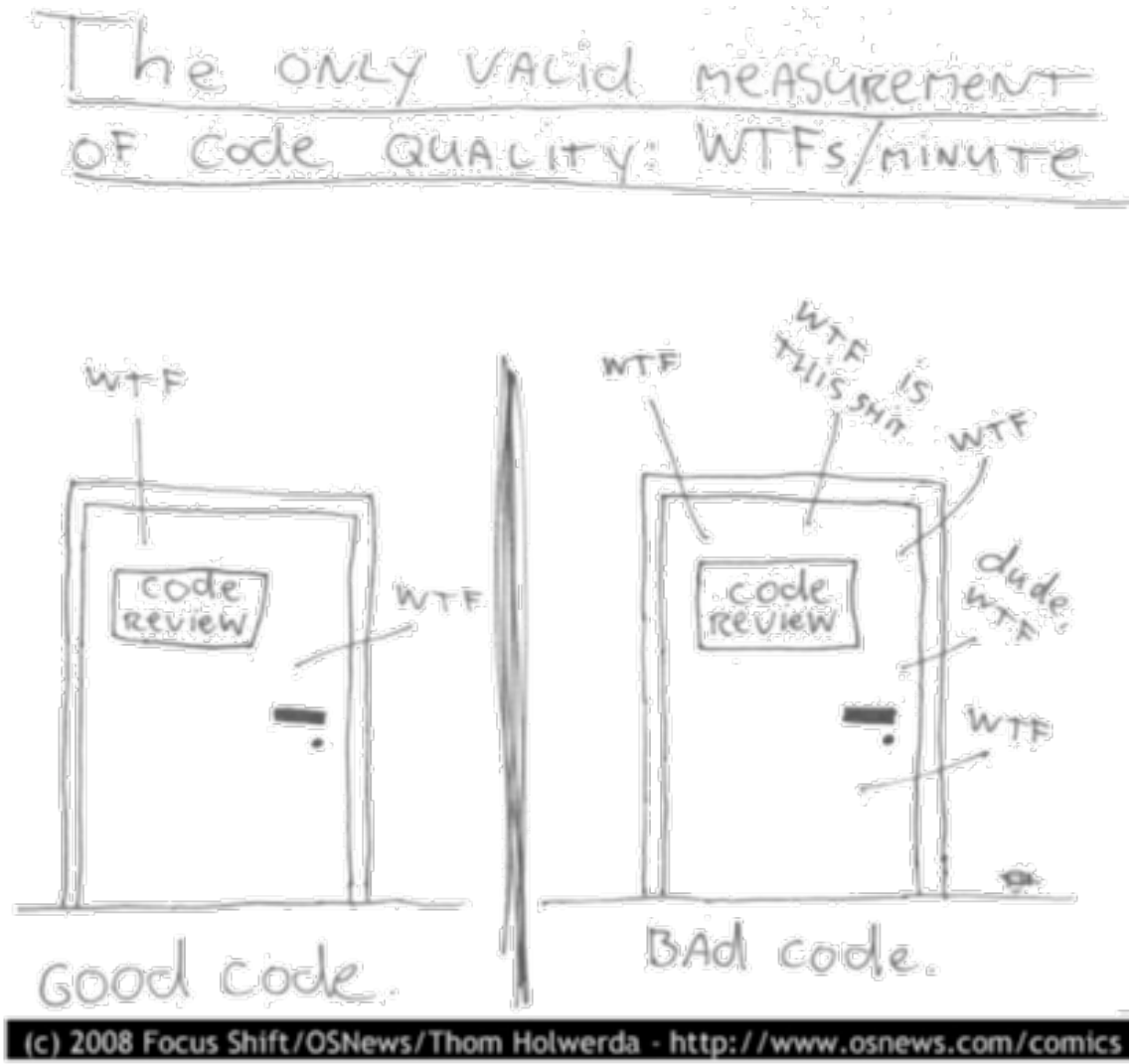
```
#include <string.h>
int main(int argc, char *argv[]) {
    int i;
    /* process command-line arguments */
    for (i = 0, i < argc; i++) {
        if (strcmp(argv[i], "-b") == 0) {
        /* whatever... */
        }
    }
    /* ... rest of program ... */
}
```

- Kommandozeilenparameter sind `argv[0]`, `argv[1]`, ...
- `argv[0]` ist der Programmname
- `strcmp()`-Funktion vergleicht Strings (falls gleich ist der Rückgabewert 0)

Kommandozeilenparameter (3)

- Nützliche Funktionen zum Einlesen der Kommandozeile
  - `atoi()` wandelt Strings in integer um (in `<stdlib.h>`)
  - `atof()` String nach float
  - `strcmp()` vergleicht Strings (in `<strings.h>`)

Ordentlich Programmieren



Assertions (1)

- Manchmal erwartet man, dass sich Programmcode in einer bestimmten Art-und-Weise verhält
- Beispiel: `sort()` sollte die Eingabe sortieren
- Programme sollten sich selbst überprüfen
- Annahme: wir haben eine Funktion `sorted()` geschrieben, die 1 zurückgibt, falls das Feld sortiert ist, ansonsten 0
- Benutze nun `assert()` zusammen mit `sorted()`, um das Sortieren des Feldes zu überprüfen.

```
#include <assert.h>
void sort(int arr[]) {
    /* sort the array somehow */
    assert(sorted(arr));
    /* "sorted" defined somewhere else;
     * returns 1 if array is sorted;
     * otherwise returns 0. */
}
```

Assert (2)

- Wenn die assertion fehlschlägt, wird das Programm beendet und Name der C-Files und Zeilennummer ausgegeben.
- Programm wird aber dadurch langsamer.
- Nach dem Debuggen, kann man die assertion ausschalten

```
% gcc –DNDEBUG program.c –o program
```

- -D bedeutet define (analog #define im C-Code)

Rekursion

- Funktionen können sich selbst aufrufen:

```
int factorial(int n) {
    assert(n >= 0);
    if (n == 0) {
        return 1;  /* Base case. */
    } else {
        /* Recursive step: */
        return n * factorial(n - 1);
    }
}
```

```
factorial(5)
--> 5 * factorial(4)
--> 5 * 4 * factorial(3)
--> 5 * 4 * 3 * factorial(2)
--> 5 * 4 * 3 * 2 * factorial(1)
--> 5 * 4 * 3 * 2 * 1 * factorial(0)
--> 5 * 4 * 3 * 2 * 1 * 1
--> 120
```