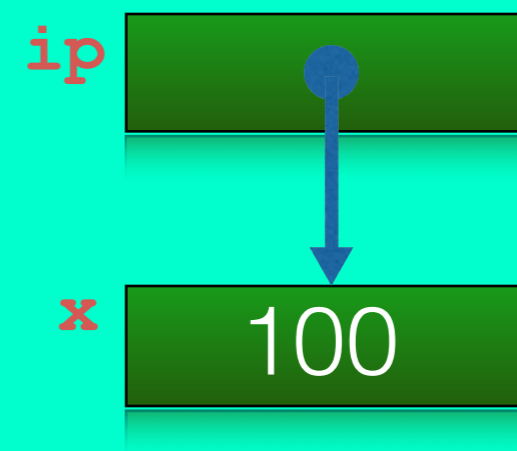


- **Wichtig:**
 - wird ein Zeiger deklariert, zeigt er zunächst irgendwo hin (nur globale Zeiger sind mit Null initialisiert)
 - Zeiger sollte also immer initialisiert werden (ggf. mit **NULL** [==0])
dann kann dieser Zustand erkannt werden: `if (p)...` // nicht: `if (p!=NULL)...`



- somit produziert
 - `int *ip; *ip = 100;` 
 - undefiniertes Verhalten (z.B. Abbruch des Programms **oder** unbemerktes Überschreiben des Wertes an einer (zufälligen) Adresse **oder** Formatieren der Festplatte ...).

- (mögliche) Korrekturen:
 - `int *ip; int x;`
 - `ip = &x; *ip = 100;`



- kurze Typnamen für Typkonstrukte kann man per

```
typedef Typkonstukt Typname;
```

definieren

- Benutzung dieser neuen Typen: wie vordefinierte Typen

Beispiel:

```
/* Typdefinition */  
typedef float real;  
typedef char letter;  
typedef int* intZeiger;  
/* Variablendefinition */  
real sum = 0.0;  
letter nextletter;
```

weil die Größen von Objekten im Speicher nicht normativ festgelegt sind, braucht man einen programmatischen Zugang zu dieser Information:

Der Operator `sizeof`

- liefert die Speicherplatzgröße (Byteanzahl) als Wert vom Typ `size_t` (zumeist unsigned long)
- wird immer zur Compile-Zeit berechnet

- kann in zwei syntaktischen Formen auftreten

(1) `sizeof (<ausdruck>)` beliebiger Ausdruck, Klammern dürfen fehlen

```
int a[10], n;
```

```
n= sizeof (a);    /* n= sizeof a; */ Empfehlung: immer Klammern!
```

(2) `sizeof (<typspezifikation>)` Klammern dürfen **NICHT** fehlen

Definition globaler Variablen

- globale Variablen werden vor dem `main()`-Programm wie folgt deklariert/definiert:

```
short number, sum;  
int bignumber, bigsum;  
char letter;  
int main() { ... }
```

- nur globale Daten erhalten eine implizite Initialisierung auf Null
- möglich: Initialisierung globaler Variablen mittels Zuweisungsoperator = dann immer eine Definition

- Beispiel:


```
float sum = 0.0;  
int bigsum = 0;  
char letter = 'A';  
int main() { ... }
```

Speicherverwaltung für globale Variablen

- Speicherplatzreservierung für globale Variablen erfolgt zum Zeitpunkt des Ladens des (übersetzten und verbundenen) Programms, Compiler hat bereits die Größe eines zusammenhängenden Speicherbereiches für sämtliche globale Variablen berechnet
- der bereitgestellte Speicherbereich ist mit 0 vorinitialisiert, wenn keine explizite Initialisierung erfolgt
- falls nutzerdefinierte Initialisierungen vorgesehen sind, erfolgen diese vor Ausführung der `main()`-Funktion
- Speicherplatzfreigabe erfolgt mit Beendigung des Programms

- lokale Variablen werden innerhalb einer Funktion oder eines lokalen Blockes definiert



```
void foo() {  
    short number, sum; /* nicht initialisiert ! */  
    {  
        int sum = 0; /* verdeckt sum aus übergeordnetem Block */  
    }  
}
```



- unbedingte Empfehlung: Initialisierung lokaler Variablen, wenn erster Zugriff lesend ist

```
void foo() {  
    short number = 1, sum = 0; /* initialisiert ! */  
    {  
        int sum = 0; /* verdeckt sum aus übergeordnetem Block */  
    }  
}
```

Speicherverwaltung für lokale Funktionsvariablen

- Speicherplatzreservierung für lokale Variablen erfolgt zum Zeitpunkt des Funktionsaufrufs im Speicher auf dem Programm-Stack (gehört zum Aktivierungsbereich der Funktion) **auto** (überflüssig)
- eine Initialisierung ist vom Nutzer vorzusehen
- am Ende der Funktion wird Aktivierungsbereich freigegeben (Werte sind verloren)
- **Achtung:** eine nicht-initialisierte Variable erhält vorheriges (zufälliges) Bitmuster, d.h. der Wert ist undefiniert 
- **Achtung:** eine Adresse einer lokalen Variablen niemals außerhalb des Bezugsrahmens verwenden 

Variablen können außerdem zusätzlich als `static` ausgezeichnet sein, für globale und lokale mit völlig unterschiedlicher Semantik



- global
 - `static` - nur in diesem File sichtbar (nicht Quelltext-übergreifend):
 - ohne `static` - (Quelltext-übergreifend) in nur einem File definiert und in anderen [ggf. implizit] als `extern` deklariert
- lokal
 - ohne `static` - wird bei jeder Ausführung der Funktion/des Blockes neu (auf dem Stack) angelegt, enthält bei fehlender Initialisierung einen undefinierten Wert
 - `static` - wird bei jeder Ausführung der Funktion/des Blockes wieder sichtbar, überlebt aber das Ende der Funktion/des Blockes, wird wie globale Daten einmalig auf 0 initialisiert und behält aber den zuletzt hinterlassenen Wert

- ANSI-C erlaubt die Angabe von Konstanten

```
int const a = 1;  
const int a = 2;
```

- Konstantendefinition kann vor oder nach der Typdefinition erfolgen
- alternativ (aber nicht besser): Definition von Konstanten durch den C-Präprozessor (mehr dazu später)

- `const` wird bei Zeigern (nicht konsequent) berücksichtigt:

```
const int c = 42;  
int* p = &c;  Initializing 'int *' with an expression of type 'const int *' discards qualifiers  
const int* pc = &c;  
// (*pc)++;  Read-only variable is not assignable
```

Ausgabe von Variablen

- C erlaubt formatierte Ausgaben mittels `printf()`-Funktion aus der C-Standard-Bibliothek `#include <stdio.h>`
- Formatanweisungen werden im ersten Parameter (ein String) kodiert, danach folgen Variablen, die ausgegeben oder eingelesen werden sollen
- `printf()` benutzt das spezielle Zeichen `%` zur Formatierung
 - `%c` : characters
 - `%d` : integers
 - `%f` : doubles (floats werden implizit nach double umgewandelt)
 - `%s` : strings, á la "Hallo"
 - `%p` : beliebige Zeiger, Adresse hexadezimal
- Beispiel:

```
printf(" %c %d %f \n", ch, i, x);
```

- Wichtig:
 - Der Programmierer ist dafür verantwortlich, dass Formatangaben und Typen der Variablen übereinstimmen,
sonst undefiniertes Verhalten (z. B. core dump) !!!



- C erlaubt formatierte Eingaben mittels `scanf()`-Funktion von einfachen Werten und Datenstrukturen
- Formatierung ähnlich zu `printf`
`scanf("%c %d %lf", &ch, &i, &x);`
- Argumente werden immer *per call by value* übergeben, wie auch Ergebnisse von Funktionen *return by value*
- um Effekte in Argumenten auszulösen braucht man also eine Indirektion (per Zeiger)

Hello World

```
/* sample program */  
#include <stdio.h>  
int main() {  
    printf("Hello, World\n"); return 0;  
}
```

- **return** ist ein Statement, das zum Beenden der Funktion führt (in C notwendig! Sonst Warnung und undefiniertes Verhalten!)
 - muss gefolgt werden von einem Ausdruck passend zum Rückgabotyp
- C erfordert ein ";" am Ende eines Ausdrucks, um ihn zur Anweisung zu machen (Funktionsruf ist Ausdruck, **NICHT** Anweisung) !
- "\n" erzeugt ein Zeilenende in der Ausgabe
- Achtung: bei Ausgabe in Dateien ist die ASCII-Kodierung des Zeilenendes abhängig vom Betriebssystem

winDOS: **CR**(015/0xD,\r')+**LF**(012/0xA/'\n')

Unix: **LF**(012/0xA/'\n')

Funktionen (vorab)

- eine Funktionsdefinition hat folgende Form:

```
type function_name (parameters)
{ local variables  C-Statements  }
< C99
```

- jede Funktion sollte vor ihrem Aufruf per Prototyp deklariert werden !

```
type function_name (parameters);
```

Ansonsten geht der Compiler davon aus, dass die Funktion ein **int**-Resultat liefert und beliebige Parameter verarbeitet (was ernsthafte Fehler verursachen kann, wenn dem nicht so ist) !

live DEMO: Wie wichtig Prototypen von Funktionen sind.

- Bisher Operationen für Bildschirmein-/Ausgabe
- Datei- Ein- und Ausgabe

`fscanf` und `fprintf`

`f` wie "File"- Operation

- Datei öffnen und schliessen (im aktuellen Verzeichnis der Programmausführung)

`fopen (name, modus)` und `fclose (fileptr)`

modus kann sein: "r" (lesend), "w" (schreibend), "a" (anhängend)

```
FILE *fopen(), *fp;
```

```
fp = fopen (name, "r");
```

```
fscanf(fp, "%d", &r);
```

```
printf ("%d", r);
```

```
fclose (fp)
```

```
...
```

Arithmetische Operationen

- arithmetische Standardoperatoren: + - * / %
- und es gibt noch mehr....
 - ++ und -- mit Variablen in Präfix- und Postfixmodus, also ++x, x++
 - Semantik: erhöhen/reduzieren um den Wert 1

Beispiel

```
int x,y,w;  
int main() { x=((++y) - (w--)) % 100; return 0; }
```

ist (im Prinzip!) äquivalent zu

```
int x,y,w;  
int main() {  
    ++y; x=(y-w) % 100; w--;  
    return 0;  
}
```

ACHTUNG: Die Reihenfolge der Berechnung von Operanden ist in C **UNDEFINIERT!** (von links nach rechts, umgekehrt, parallel, beliebige Reihenfolge, ...)



- Modulo-Operator “%” ist nur für `int`-Typ definiert
- Division “/” ist für `int` und `double/float` definiert
- Achtung:
 - Ergebnis von `x = 3 / 2` ist 1, selbst wenn `x` als `float` definiert wurde!!
 - sind beide Argumente von “/” als `int` definiert, wird die Operation als integer -Division durchgeführt
- korrekte Spezifikation:
 - `x = 3.0 / 2` oder `x = 3 / 2.0`
 - oder (besser) `x = 3.0 / 2.0`

- **ACHTUNG:** Die Reihenfolge der Berechnung von Operanden ist in C **UNDEFINIERT!**
- Sofern dabei Seiteneffekte möglich sind, hat das Programm **undefined behaviour!**



Sie müssen lernen zu unterscheiden zwischen:

- Was macht mein C-Programm (beobachtbar) auf einer bestimmten Plattform (Compiler + Betriebssystem-Umgebung) ?
- Was legt der Sprachstandard fest ?

Berechnung von Operanden

Damit ist das Argument:

„Aber auf der Plattform XYZ hat mein Programm funktioniert!“

wertlos !

Neue Qualität von Problemen bei der Programmentwicklung
(die Sie von Java nicht gewohnt sind)

Hilfsmittel:

- profunde Sprachkenntnis (insb. der Teile, die **NICHT** explizit definiert sind)
- Maximales Warnungslevel im Compiler einstellen und Warnungen ernst nehmen
- Cross-Checks: verschiedene Compiler auf verschiedenen Plattformen nutzen
- Meta-Tools (flex, bison) generieren korrekten Code

Berechnung von Operanden

Ein Beispiel zur Abschreckung:

```
#include <stdio.h>

int foo(int x, int y)
{ return x + y; }

int main() {
    int i=1;

    printf("%d\n", foo(++i, --i));
    return 0;
}
```

Berechnung von Operanden

```
class Foo {  
    static int foo(int x, int y)  
    { return x + y; }  
  
    public static void main(String[] s) {  
        int i=1;  
        System.out.println(foo(++i, --i));  
    }  
}
```

Ausgabe ?

3

Berechnung von Operanden

gcc 3.3.3

```
$ gcc -Wall -o foo foo.c
foo.c: In Funktion »main«:
foo.c:11: Warnung: operation on `i' may be undefined
$ foo
1
$
```

Reduziere i ($1 \rightarrow 0$)
Wert von i (0) auf den Stack
Erhöhe i ($0 \rightarrow 1$)
Wert von i (1) auf den Stack
Rufe `foo`: `return 1 + 0`

Berechnung von Operanden

vc++ 7.1

```
H:>cl /W4 foo.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86
Copyright (C) Microsoft Corporation 1984-2002. All rights reserved.
```

```
Microsoft (R) Incremental Linker Version 7.10.3077
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
/out:foo.exe
foo.obj
```

```
H:>foo
2
```

```
H:>
```

Erhöhe i (1 -> 2)

Reduziere i (2 -> 1)

Wert von i (1) auf den Stack

Wert von i (1) auf den Stack

Rufe foo: return 1 + 1

Beide Compiler arbeiten korrekt !

Kurzform von Operatoren

- C stellt "elegante" Abkürzungen für Operatoren zur Verfügung
 - Beispiel: $i = i + 3$ oder $x = x * (y + 2)$
 - Umschreibung in C (generell) in "Kurzform":
 $expression_1 \text{ op } = expression_2$
 - Dies ist äquivalent zu (und u. U. effizienter als):
 $expression_1 = expression_1 \text{ op } expression_2$
 - Beispiel umgeformt:
 $i = i + 3$ als $i += 3$
 $x = x * (y + 2)$ als $x *= y + 2$
- $x *= y + 2$ bedeutet $x = x * (y + 2)$ und **nicht** $x = x * y + 2$

- Test auf Gleichheit: "=="
Achtung: Bitte "=" nicht mit "==" verwechseln !!!
- zulässig ist auch: `if (i = j) ...`
 - legales C-Statement (aus syntaktischer Sicht):
Zuweisung des Wertes von "j" nach "i",
gleichzeitig Wert des Ausdrucks, der als TRUE interpretiert wird,
falls j ungleich 0 ist
 - manche Compiler (nicht alle) warnen

Vergleichsoperatoren

- ungleich ist: "!="
- andere Operatoren
 - < (kleiner als)
 - > (größer als)
 - <= (kleiner oder gleich),
 - >= (größer oder gleich)

Die logischen Grundoperatoren sind:

- `&&` für logisches AND (short circuit evaluation!)
- `||` für logisches OR (short circuit evaluation!)
- `!` Für logisches NOT

Achtung: `&` und `|` existieren auch als zweistellige Operatoren, haben aber eine andere Semantik:

- Bit-orientiertes AND
- Bit-orientiertes OR (später)

Achtung: `&` ist auch ein einstelliger Operator (Adresse von)

- Verwendung in logischen Ausdrücken (als `int` bewertet)

Präzedenzen von Operatoren

- Bedeutung von $a + b * c$
 - Gemeint könnte sein
 - $(a + b) * c$
 - $a + (b * c)$
 - alle Operatoren besitzen einen "Präzedenzwert" (Priorität)
 - Operatoren mit hoher Priorität werden vor Operatoren mit geringerer Priorität evaluiert
 - Operatoren mit gleicher Priorität werden von links nach rechts evaluiert, wenn sie rechts-assoziativ sind:
 - $a - b - c$ wird als $(a - b) - c$ evaluiert
- im Zweifelsfall besser ein Klammerpaar zu viel, als eines zu wenig

Präzedenzordnung

- Operatoren in C von hoher bis niedriger Priorität (Präzedenz):
(sind noch nicht alle eingeführt):

```
( ) [ ] -> .  
! - * & sizeof cast ++ -- (diese werden von rechts nach links ausgewertet)  
/ %  
+ -  
< <= >= > == !=  
&  
|  
&&  
||  
?: (rechts nach links)  
= += -= .... (rechts nach links)  
, (comma)
```

- Beispiel:** `a < 10 && 2 * b < c` wird als
`(a < 10) && ((2 * b) < c)` interpretiert
- `i = foo() , bar() , 42; // rufe foo() , rufe bar() , i = 42;`

Konflikt mit Komma in anderen Kontexten mit Klammern lösbar:

```
int i = (foo() , bar() , 42); baz((foo() , bar() , 42));
```

Die Programmiersprache C

3. Anweisungen, Funktionen, Felder, Strukturen

Prof. Dr. sc. Joachim Fischer (Dr. Klaus Ahrens)
Institut für Informatik, Humboldt-Universität zu Berlin
SS 2014

Algorithmik analog zu Java in Syntax & Semantik

- Zuweisung (auch +=, -=, ...)
- **if, switch**
- **while, do-while, for**
- **break, continue**
- Funktionsaufruf (Java: Methodenaufruf)
- **return**

```
loop: x=y;  
...  
goto loop;
```

```
goto skip;  
...  
skip:
```

nicht vorhanden

throw, try, catch, synchronized

Zuweisungsoperator

- Zuweisung durch "="
- Zuweisung ist **KEINE** Anweisung, sondern ein Ausdruck (wie in Java auch) !
- C erlaubt Mehrfachzuweisungen (wie Java)
- Beispiel:
`a=b=c=d=3;`
- ...dies ist äquivalent zu (aber nicht notwendig effizienter als):
`d=3; c=3; b=3; a=3;`

if - Anweisung

- Grundform:

```
if (expression) statement
```

```
if (expression) statement1 else statement2
```

- Schachtelung möglich:

```
if (expression) statement1
```

```
else if (expression) statement2
```

```
else statement3
```

- Beispiel:

```
int main() {      int x, y, w, z;  
    if (x>0) { ... z=w; ... }  
    else      { ... z=y; ... }  
}
```

```
if (exp1)  
    if (exp2) stmt1  
    else stmt2
```

dangling else

- Bindung an das innerste if
- auch durch Formatierung unterstreichen

```
if (exp1)  
    {if (exp2) stmt1}  
else stmt2
```


Operator ? :

- Der »? :«-Operator (»ternary condition«) ist die effizientere Form, um einfache **if**-Anweisungen auszudrücken
- syntaktische Form:
`expression1 ? expression2 : expression3` `expression2` und `expression3` müssen kompatible Typen haben
- Semantik:
if `expression1` then Wert = `expression2` else Wert = `expression3`
- Beispiel: Zuweisung des Maximums von a und b auf z
`z = (a > b) ? a : b;`
äquivalent zu:
`if (a > b) z = a; else z = b;`

- Die `switch`-Anweisung erlaubt mehrfache Alternativen einer Selektion auf einer »Ebene«

```
switch (expression) {  
    case item1: statement1  
    case item2: statement2  
    case itemn: statementn  
    default : statement  
}
```

- In jeder Alternative muss der Wert von `itemi` eine Konstante sein, Variablen sind nicht erlaubt
- leere Anweisung durch ein »;« möglich

switch - Anweisung

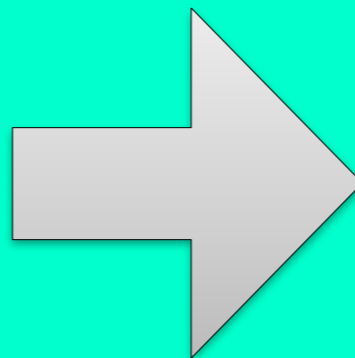
- Beispiel:

```
switch (letter) {  
    case 'A': howmanyAs++; /* fall through */  
    case 'E':  
    case 'I':  
    case 'O':  
        case 'U': numberofvowels++; break;  
        case ' ': numberofspaces++; break;  
        default: numberofothers++; break;  
}
```

switch - Anweisung

- Die Syntax der switch-Anweisung ist in Wahrheit:
switch (expression) statement
- damit kann man (anders als in Java) überraschende Effekte ausdrücken,
Beispiel: *Duff's Device* (loop unrolling)

```
send(to, from, count)
register short *to, *from;
register count;
{
    do
        *to = *from++;
    while (--count>0);
}
```



```
send(to, from, count)
register short *to, *from;
register count;
{
    register n = (count+7)/8;
    switch (count%8) {
        case 0: do { *to = *from++;
        case 7:     *to = *from++;
        case 6:     *to = *from++;
        case 5:     *to = *from++;
        case 4:     *to = *from++;
        case 3:     *to = *from++;
        case 2:     *to = *from++;
        case 1:     *to = *from++;
    } while (--n>0);
}
```

http://de.wikipedia.org/wiki/Duff's_Device

switch - Anweisung

- Die Syntax der switch-Anweisung ist in Wahrheit:

switch (expression) statement

- damit kann man (anders als bei if) mehrere Alternativen angeben
Beispiel: *Duffs Device*

```
send(to, from, count)
register short *to, *from;
register count;
{
    do
        *to = *from++;
    while (--count>0);
}
```

http://de.wikipedia.org/wiki/Duffs_Device

im (Ur-) C (auch K&R C)
wurde die Typinformation
von Parametern nach der
Parameterliste spezifiziert

switch - Anweisung

- Die Syntax der switch-Anweisung ist in Wahrheit:
switch (expression) statement
- damit kann man (anders als in Java) überraschende Effekte ausdrücken,
Beispiel: *Duffs Device* (loop unrolling)

register ist (immer noch) ein Keyword von C (auch aus K&R-Zeiten) um anzuzeigen, dass Parameter/lokale Variablen in Registern (statt im Hauptspeicher) angelegt werden sollen - heute *depricated*: der Compiler macht das in eigener Regie

```
send(to, from, count)
register short *to, *from;
register count;
{
    register n = (count+7)/8;
    switch (count%8) {
        case 0: do { *to = *from++;
        case 7:      *to = *from++;
        case 6:      *to = *from++;
        case 5:      *to = *from++;
        case 4:      *to = *from++;
        case 3:      *to = *from++;
        case 2:      *to = *from++;
        case 1:      *to = *from++;
    } while (--n>0);
}
```

rice

for - Anweisung

- Die for-Anweisung hat die folgende Form:
`for` (for-init-statement expression₁; expression₂) statement

Erklärung:

- for-init-statement initialisiert die Iteration
- expression₁ ist der Test zur Beendigung der Iteration
- expression₂ modifiziert eine Schleifenvariable (mehr als nur das Erhöhen eine Schleifenvariablen um 1)
- C99 kennt auch sog. for-Scope: `for (int i=1;...;...) ...`
C benutzt `for`-Anweisung oft anstelle von `while`-Schleifen
- Beispiel:

```
int main() { int x; for (x=3;x>0;x--) { printf("x=%d\n",x); } }
```

... erzeugt als Ausgabe:

```
x=3  
x=2  
x=1
```

while - Anweisung

- Die while-Anweisung hat die folgende Form:

```
while (expression) statement
```

Beispiel:

```
int main() {int x=3;  
    while (x>0) { printf("x=%d\n",x); x--; }  
}
```

...erzeugt als Ausgabe:

```
x=3  
x=2  
x=1
```

- legale while-Anweisungen:

```
while (x--) ...
```

```
while (x=x+1) ...
```

```
while (x+=5) ...
```

```
while (1) ... /* forever, auch for(;;) ... */
```


while - Anweisung

üblich in C: vollständige Ausführung von Operationen im **while**-Ausdruck:

```
while (i++ < 10);  
while ( (ch = getchar()) != 'q' )  
    putchar(ch);  
while (*dest++ = *src++); // ?????  
/* klassisches C-Idiom (strcpy) */
```

do-while - Anweisung

do-while-Anweisung hat die Form:

`do statement while (expression);` <- hier Semikolon explizit !

Beispiel:

```
int main() {  
    int x=3;  
    do { printf("x=%d\n", x--); }  
    while (x>0);  
}
```

... erzeugt als Ausgabe:

```
x=3  
x=2  
x=1
```

break und continue

C enthält zwei Möglichkeiten zur Schleifensteuerung:

- **break**: Verlassen der (innersten) Schleife oder `switch`-Anweisung.
- **continue**: Überspringen einer Schleifeniteration

Beispiel:

```
while (scanf("%d", &value ) == 1 && value != 0) {
    if (value < 0) {
        printf("Illegal value\n"); break; /* Abandon the loop */
    }
    if (value > 100) {
        printf("Invalid value\n"); continue; /* Skip to start loop again */
    }
    /* Process the value read, guaranteed to be between 1 and 100 */
    ...;
} /* end while */
```

- Form

```
returntype fn_name (paramdef1, paramdef2, ...)  
{ localvariables  
  functioncode }
```

bei C99 (und C++) nicht zwingend nur am Anfang

- Beispiel Durchschnitt zweier `float`-Werte

```
/* besser, weil kompakter: */  
float findaverage (float a, float b) {  
    return (a+b)/2;  
}
```

- Aufruf der Funktion

```
void foo() { float a=5, b=15, result;  
             result=findaverage(a,b);  
             printf("average=%f\n", result); }
```

Basissyntax (Definition) wie bei Java-Methoden

```
void readvectors (vector v1, vector v2) {  
    int i;        /* zu Beginn */  
    ...          /* dann Anweisungen */  
                /* ab C99 Typ- u. Variablendeklarationen später möglich */  
}
```

Besondere Sichtbarkeitsregeln:

```
static void readvectors (v1, v2);  
// nur in Übersetzungseinheit (File) sichtbar
```

```
extern void readvectors (v1, v2);  
// auch nach außen sichtbar, extern ist Standardannahme
```

Verschachtelung von Funktionen

- wie in Java **nicht** erlaubt, in C sind alle Funktionen global, obwohl Blockkonzept (Gültigkeitsbereiche für Bezeichner) seit Algol-60 bekannt
- **Gründe:**
 - Leichter und effektiver durch Compiler zu verarbeiten (Compilezeit)
 - Verwaltungsaufwand für Funktionsrufe geringer (Laufzeit)
- **Kritik:** methodischer Nachteil
Programmstruktur entspricht u.U. nicht der Problemstruktur