

- Um einen raschen Start in den praktischen Teil zu ermöglichen, wird C zunächst etwas oberflächlich mit einigen Beispielen vorgestellt.
- Später werden dann die Feinheiten vertieft vorgestellt.
- Im Vergleich zu Java gibt es in C keine Klassen. Stattdessen sind alle Konstrukte recht nah an den gängigen Prozessorarchitekturen, die das ebenfalls nicht kennen.
- Statt Klassen gibt es in C Funktionen, die Parameter erhalten und einen Wert zurückliefern. Da sie sich nicht implizit auf ein Objekt beziehen, sind sie am ehesten vergleichbar mit den statischen Methoden in Java.
- Jedes C-Programm benötigt ähnlich wie in Java eine *main*-Funktion.

hallo.c

```
main() {  
    /* puts: Ausgabe einer Zeichenkette nach stdout */  
    puts("Hallo zusammen!");  
}
```

- Dieses Programm gibt den gezeigten Text aus, gefolgt von einem Zeilentrenner – analog zu *System.out.println*.
- Im Unterschied zu Java muss wirklich eine Zeichenkette angegeben werden. Andere Datentypen werden hier nicht implizit über eine *toString*-Methode in Zeichenketten zum Ausdrucken verwandelt.

```
clonmel$ gcc -fmessage-length=70 -Wall hallo.c
hallo.c:1:1: warning: return type
  defaults to 'int' [-Wimplicit-int]
main() {
^
hallo.c: In function 'main':
hallo.c:3:4: warning: implicit
  declaration of function 'puts'
  [-Wimplicit-function-declaration]
  puts("Hallo zusammen!");
  ^
clonmel$ a.out
Hallo zusammen!
clonmel$
```

- Der *gcc* ist der *GNU-C-Compiler*, mit dem wir unsere Programme übersetzen.
- Ist kein Name für das zu generierende ausführbare Programm angegeben, so wird dieses *a.out* genannt.
- Die Option *-Wall* bedeutet, dass alle Warnungen ausgegeben werden sollen.

- Die Programmiersprache C ist standardisiert (ISO 9899). Den Standard gibt es in den Versionen von 1990, 1999, 2011 und 2018.
- Voreinstellungsgemäß geht *gcc* ab Version 5.1 von *C11* aus, bei älteren Versionen (etwa 4.9) ist noch *C90* die Voreinstellung. Es ist auch möglich, mit der Option „*-std=c11*“ den aktuellen Standard von 2011 explizit auszuwählen.
- Statt „*-std=c11*“ ist auch „*-std=gnu11*“ möglich – dann stehen auch verschiedene Erweiterungen zur Verfügung, die nicht über C99 oder C11 vorgegeben sind.
- Für die Übungen empfiehlt sich grundsätzlich die Wahl von *gnu11*, wobei letzteres erst ab GCC 4.7.x unterstützt wird. Auf unseren Maschinen haben wir folgende Versionen: 7.3.0 (Solaris/Intel, Theon), 5.2.0 (Solaris/Intel, Thales), 4.9.2 (Debian, Pool in E.44) oder 4.8.0 (Solaris/SPARC, Theseus).
- Mit dem Aufruf von *gcc --version* lässt sich die Version des *gcc*-Übersetzers feststellen.

hallo1.c

```
#include <stdio.h> /* Standard-I/O-Bibliothek einbinden */

int main() {
    /* puts: Ausgabe eines Strings nach stdout */
    puts("Hallo zusammen!");
}
```

- Da die Ausgabefunktion *puts()* nicht bekannt war, hat der Übersetzer geraten. Nun ist diese Funktion durch das Einbinden der Deklarationen der Standard-I/O-Bibliothek (siehe **#include** <stdio.h>) bekannt.
- Der *Typ des Rückgabewertes* der *main()*-Funktion ist nun als **int** (Integer) angegeben (der Übersetzer hat vorher auch **int** geraten.)

```
doolin$ gcc -Wall -o hallo1 hallo1.c
doolin$ hallo1
Hallo zusammen!
doolin$
```

- Mit der Option „-o“ kann der Name des Endprodukts beim Aufruf des `gcc` spezifiziert werden.
- Anders als bei Java ist das Endprodukt selbständig ausführbar, da es in Maschinensprache übersetzt wurde.
- Das bedeutet jedoch auch, dass das Endprodukt nicht portabel ist, d.h. bei anderen Prozessorarchitekturen oder Betriebssystemen muss das Programm erneut übersetzt werden.
- Das gilt auch auf unseren Maschinen, die in drei Gruppen fallen: Solaris/Intel, Solaris/SPARC und Debian/Intel.

```
.file "hallo1.c"
.section ".rodata"
.align 8
.LLC0:
.asciz "Hallo zusammen!"
.section ".text"
.align 4
.global main
.type main, #function
.proc 04
main:
save %sp, -96, %sp
sethi %hi(.LLC0), %g1
or %g1, %lo(.LLC0), %o0
call puts, 0
nop
mov 0, %g1
mov %g1, %i0
return %i7+8
nop
.size main, .-main
.ident "GCC: (GNU) 4.8.0"
```

- Resultat von „`gcc -S hallo.c`“ auf einer SPARC-Plattform.

```
.file "hallo1.c"
.section      .rodata
.LC0:
.string "Hallo zusammen!"
.text
.globl main
.type main, @function
main:
    pushl    %ebp
    movl    %esp, %ebp
    andl    $-16, %esp
    subl    $16, %esp
    movl    $.LC0, (%esp)
    call    puts
    movl    $0, %eax
    leave
    ret
    .size   main, .-main
    .ident  "GCC: (GNU) 4.7.1"
```

- Resultat von „`gcc -S hallo.c`“ auf einer Intel/x86-Plattform.



quadrates.c

```
#include <stdio.h>

const int MAX = 20;    /* globale Integer-Konstante */

int main() {
    puts("Zahl | Quadratzahl");
    puts("-----+-----");
    for (int n = 1; n <= MAX; n++) {
        printf("%4d | %7d\n", n, n * n); /* formatierte Ausgabe */
    }
}
```

- Dieses Programm gibt die ersten 20 natürlichen Zahlen und ihre zugehörigen Quadratzahlen aus.
- Variablendeklarationen können außerhalb von Funktionen stattfinden. Dann gibt es die Variablen genau einmal und ihre Lebensdauer erstreckt sich über die gesamte Programmlaufzeit.

```
quadrante.c
```

```
printf("%4d | %7d\n", n, n * n); /* formatierte Ausgabe */
```

- Formatierte Ausgaben erfolgen in C mit Hilfe von *printf*.
- Die erste Zeichenkette kann mehrere Platzhalter enthalten, die jeweils mit „%“ beginnen und die Formatierung eines auszugebenden Werts und den Typ spezifizieren.
- „%4d“ bedeutet hier, dass ein Wert des Typs **int** auf einer Breite von (mindestens) vier Zeichen dezimal auszugeben ist.

quadrate.c

```
for (int n = 1; n <= MAX; n++) {  
    printf("%4d | %7d\n", n, n * n); /* formatierte Ausgabe */  
}
```

- Wie in Java kann eine Schleifenvariable im Initialisierungsteil einer **for**-Schleife deklariert und initialisiert werden.
- Dies ist im Normalfall vorzuziehen.
- Gelegentlich finden sich noch Deklarationen von Schleifenvariablen außerhalb der **for**-Schleife, weil dies frühere C-Versionen nicht anders vorsahen.

```
#include <stdio.h>

int main() {
    printf("Geben Sie zwei positive ganze Zahlen ein: ");
    /* das Resultat von scanf ist die
       Anzahl der eingelesenen Zahlen
       */
    int x, y;
    if (scanf("%d %d", &x, &y) != 2) { /* &-Operator konstruiert Zeiger */
        return 1; /* Exit-Status ungleich 0 => Fehler */
    }

    int x0 = x;
    int y0 = y;

    while (x != y) {
        if (x > y) {
            x = x - y;
        } else {
            y = y - x;
        }
    }

    printf("ggT(%d, %d) = %d\n", x0, y0, x);
}
```

euklid.c

```
if (scanf("%d %d", &x, &y) != 2) {  
    /* Fehlerbehandlung */  
}
```

- Die Programmiersprache C kennt nur die *Werteparameter-Übergabe* (*call by value*).
- Daher stehen auch bei *scanf()* nicht direkt die Variablen *x* und *y* als Argumente, weil dann *scanf()* nur die Kopien der beiden Variablen zur Verfügung stehen würden.
- Mit dem Operator *&* wird hier jeweils ein *Zeiger* auf die folgende Variable erzeugt. Der Wert eines Zeigers ist die *virtuelle Adresse* der Variablen, auf die er zeigt.
- Daher wird in diesem Zusammenhang der unäre Operator *&* auch als *Adressoperator* bezeichnet.

euklid.c

```
if (scanf("%d %d", &x, &y) != 2) {  
    /* Fehlerbehandlung */  
}
```

- Die Programmiersprache C kennt weder eine Überladung von Operatoren oder Funktionen.
- Entsprechend gibt es nur eine einzige Instanz von *scanf()*, die in geeigneter Weise „erraten“ muss, welche Datentypen sich hinter den Zeigern verbergen.
- Das erfolgt (analog zu *printf*) über Platzhalter. Dabei steht „%d“ für das Einlesen einer ganzen Zahl in Dezimaldarstellung in eine Variable des Typs **int**.
- Variablen des Typs **float** (einfache Genauigkeit) können mit „%f“ eingelesen werden, **double** (doppelte Genauigkeit) mit „%lf“.

euklid.c

```
if (scanf("%d %d", &x, &y) != 2) {  
    /* Fehlerbehandlung */  
}
```

- Der Rückgabewert von *scanf* ist die Zahl der erfolgreich eingelesenen Werte.
- Deswegen wird hier das Resultat mit der 2 verglichen.
- Es sollte immer überprüft werden, ob Einlesefehler vorliegen. Normalerweise empfiehlt sich dann eine Fehlermeldung und ein Ausstieg mit *exit(1)* bzw. innerhalb von *main* mit **return 1**.
- Ausnahmenbehandlungen (*exception handling*) gibt es in C nicht. Stattdessen geben alle Ein- und Ausgabefunktionen (in sehr unterschiedlicher Form) den Erfolgsstatus zurück.

|                        |   |   |
|------------------------|---|---|
|                        | → | ⟨external-declaration⟩                    |
|                        | → | ⟨translation-unit⟩ ⟨external-declaration⟩ |
| ⟨external-declaration⟩ | → | ⟨declaration⟩                             |
|                        | → | ⟨function-definition⟩                     |

- Eine Übersetzungseinheit (*translation unit*) in C ist eine Folge von globalen (d.h. normalerweise extern sichtbaren) *Vereinbarungen*, zu denen Funktionsdefinitionen, Funktionsdeklarationen, Typ-Vereinbarungen und Variablenvereinbarungen gehören.
- Deklarationen und Definitionen sind Vereinbarungen. Eine Definition führt zum Anlegen einer Variablen oder Funktion. Eine Deklaration verbindet einen Variablen- oder Funktionsnamen nur mit einem Typ bzw. Signatur, ohne diese anzulegen.



```
int i = 7; // Variablendefinition
extern int j; // Variablendeklaration
double sin(double x); // Funktionsdeklaration
double f(double x) { // Funktionsdefinition
    return 2 * x;
}
```

- Deklarationen und Definitionen sind Vereinbarungen. Eine Definition führt zum Anlegen einer Variablen oder Funktion. Eine Deklaration verbindet einen Variablen- oder Funktionsnamen nur mit einem Typ bzw. Signatur, ohne diese anzulegen.
- Header-Dateien bestehen typischerweise bei C nur aus Deklarationen von Objekten, die in den entsprechenden Bibliotheken definiert sind.
- Global deklarierte Objekte mit uneingeschränkter Sichtbarkeit füllen mit ihrem Namen den globalen Namensraum.
- Die Mehrfachdefinition eines Namens führt normalerweise zu Fehlern beim Zusammenbau eines Programms. Genauso eine Deklaration ohne zugehörige Definition.

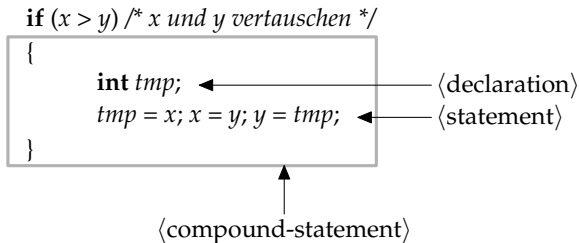
|   |                   |   |
|---|-------------------|---|
| $\langle \text{declaration} \rangle$            | $\longrightarrow$ | $\langle \text{declaration-specifiers} \rangle$<br>$\langle \text{init-declarator-list} \rangle$ „;“  |
|   | $\longrightarrow$ | $\langle \text{static-assert-declaration} \rangle$  |
| $\langle \text{declaration-specifiers} \rangle$ | $\longrightarrow$ | $\langle \text{declaration-specifier} \rangle$<br>[ $\langle \text{declaration-specifiers} \rangle$ ] |
| $\langle \text{declaration-specifier} \rangle$  | $\longrightarrow$ | $\langle \text{storage-class-specifier} \rangle$  |
|   | $\longrightarrow$ | $\langle \text{type-specifier} \rangle$   |
|   | $\longrightarrow$ | $\langle \text{type-qualifier} \rangle$   |
|   | $\longrightarrow$ | $\langle \text{function-specifier} \rangle$   |
|   | $\longrightarrow$ | $\langle \text{alignment-specifier} \rangle$  |

- Bei einer einfachen Variablendeklaration `int i = 7;` ist „**int**“ der  $\langle \text{type-specifier} \rangle$ , während „`i = 7`“ zur  $\langle \text{init-declarator-list} \rangle$  gehört.

|                       |   |  |
|-----------------------|---|--|
| ⟨function-definition⟩ | → | ⟨declaration-specifiers⟩ ⟨declarator⟩<br>[ <i>declaration – list</i> ]<br>⟨compound-statement⟩ |
| ⟨compound-statement⟩  | → | „{“ [ ⟨block-item-list⟩ ] „}“  |
| ⟨block-item-list⟩     | → | ⟨block-item⟩   |
|                       | → | ⟨block-item-list⟩ ⟨block-item⟩   |
| ⟨block-item⟩          | → | ⟨declaration⟩  |
|                       | → | ⟨statement⟩  |

- Bei „**double** *square*(**double** *x*){ **return** *x\*x*; }“ wird „**double**“ unter die ⟨declaration-specifiers⟩ subsummiert, „*square*(**double** *x*)“ bildet den ⟨declarator⟩ und „{ **return** *x\*x*; }“ das ⟨compound-statement⟩.

⟨statement⟩ → ⟨expression-statement⟩  
→ ⟨labeled-statement⟩  
→ ⟨compound-statement⟩  
→ ⟨conditional-statement⟩  
→ ⟨iterative-statement⟩  
→ ⟨switch-statement⟩  
→ ⟨break-statement⟩  
→ ⟨continue-statement⟩  
→ ⟨return-statement⟩  
→ ⟨goto-statement⟩  
→ ⟨null-statement⟩



- Mit **int tmp;** wird eine lokale Variable mit dem Datentyp **int** deklariert.
- Die Sichtbarkeit von *tmp* erstreckt sich auf den umrandeten Anweisungsblock (⟨compound-statement⟩).
- Lokale Variablen werden dann erzeugt, wenn der sie umgebende Block ausgeführt wird. Ihre Existenz endet mit dem Erreichen des Blockendes. Bei Rekursion kann eine lokale Variable mehrfach instantiiert werden.

varinit.c

```
#include <stdio.h>

int main() {
    int i; /* left uninitialized */
    int j = i; /* effect is undefined, yet compilers accept it */
    printf("%d\n", j);
}
```

- In Java durften lokale Variablen solange nicht verwendet werden, solange sie nicht in jedem Falle initialisiert worden sind. Dies wird bei Java vom Übersetzer zur Übersetzzeit überprüft.
- In C geschieht dies nicht. Der Wert einer uninitialisierten lokalen Variable ist undefiniert.
- Um das Problem zu vermeiden, sollten lokale Variablen entweder bei der Deklaration oder der darauffolgenden Anweisung initialisiert werden.
- Der gcc warnt bei eingeschalteter Optimierung und bei neueren Versionen auch ohne Optimierung. Viele Übersetzer tun dies jedoch nicht.

Bei etwas älteren gcc-Übersetzern:

```
clonard$ gcc --version | sed 1q
gcc (GCC) 4.1.1
clonard$ gcc -std=gnu99 -Wall -o varinit varinit.c && ./varinit
4
clonard$ gcc -O2 -std=gnu99 -Wall -o varinit varinit.c && ./varinit
varinit.c: In function 'main':
varinit.c:5: warning: 'i' is used uninitialized in this function
7168
clonard$
```

Bei einer etwas neueren gcc-Version:

```
clonard$ gcc --version | sed 1q
gcc (GCC) 4.8.0
clonard$ gcc -std=gnu11 -fmessage-length=70 -Wall -o varinit varinit.c
varinit.c: In function 'main':
varinit.c:5:8: warning: 'i' is used uninitialized in this function
[-Wuninitialized]
    int j = i; /* effect is undefined, yet compilers accept it */
           ^
clonard$ varinit
0
clonard$
```

- Kommentare beginnen mit „/\*“, enden mit „\*/“, und dürfen nicht geschachtelt werden.
- Alternativ kann seit C99 in Anlehnung an C++ ein Kommentar auch mit „//“ begonnen werden, der sich bis zum Zeilenende erstreckt.
- Kommentarzeichen werden innerhalb von konstanten Zeichen oder Zeichenketten nicht als solche erkannt.



|                 |               |                 |                 |                       |
|-----------------|---------------|-----------------|-----------------|-----------------------|
| <b>auto</b>     | <b>else</b>   | <b>long</b>     | <b>switch</b>   | <b>_Atomic</b>        |
| <b>break</b>    | <b>enum</b>   | <b>register</b> | <b>typedef</b>  | <b>_Bool</b>          |
| <b>case</b>     | <b>extern</b> | <b>restrict</b> | <b>union</b>    | <b>_Complex</b>       |
| <b>char</b>     | <b>float</b>  | <b>return</b>   | <b>unsigned</b> | <b>_Generic</b>       |
| <b>const</b>    | <b>for</b>    | <b>short</b>    | <b>void</b>     | <b>_Imaginary</b>     |
| <b>continue</b> | <b>goto</b>   | <b>signed</b>   | <b>volatile</b> | <b>_Noreturn</b>      |
| <b>default</b>  | <b>if</b>     | <b>sizeof</b>   | <b>while</b>    | <b>_Static_assert</b> |
| <b>do</b>       | <b>inline</b> | <b>static</b>   | <b>_Alignas</b> | <b>_Thread_local</b>  |
| <b>double</b>   | <b>int</b>    | <b>struct</b>   | <b>_Alignof</b> |                       |

- Im übrigen sind alle Namen, die mit einem Unterstrich beginnen und von einem weiteren Unterstrich oder einem Großbuchstaben gefolgt werden, reserviert. Beispiel: `__func__`.
- Im übrigen sind diverse Namen aus der Standardbibliothek reserviert.
- Da C mit einem globalen Namensraum arbeitet, sind Namenskonflikte ein Problem.

- C ist eine über Jahrzehnte gewachsene Programmiersprache.
- So gab es ursprünglich keinen Datentyp *bool*. Stattdessen wurde einfach **int** verwendet. Viele Programmierer definierten sich gerne selbst aus Gründen der Lesbarkeit einen Typ *bool* mitsamt Definitionen für *true* und *false*.
- Als in C90 ein entsprechender Datentyp eingeführt wurde, sollten aus Gründen der Aufwärtskompatibilität Programme mit selbstdefinierten *bool*, *true* und *false* nicht beeinträchtigt werden.
- Deswegen wurde für den Boolean-Datentyp das Schlüsselwort **\_Bool** vergeben. Dieses wird aber typischerweise nicht direkt verwendet. Stattdessen können mit Hilfe von **#include** `<stdbool.h>` darauf basierende Definitionen für *bool*, *true* und *false* bezogen werden.
- Analog gibt es `<stdalign.h>`, `<complex.h>`, `<assert.h>` und `<threads.h>`.

sum.c

```
#include <stdio.h>
#include <stdbool.h>
int main() { bool EOF = false; int sum = 0;
    do {
        int value; EOF = (scanf("%d", &value) != 1);
        if (!EOF) sum += value;
    } while (!EOF); printf("sum = %d\n", sum);
}
```

- Dieses nicht sehr elegant geschriebene Programm lässt sich nicht übersetzen und die Fehlermeldungen geben Rätsel auf:

```
clonmel$ gcc -Wall -o sum sum.c
In file included from /usr/include/stdio.h:66:0,
        from sum.c:1:
sum.c: In function 'main':
sum.c:3:19: error: expected identifier or '(' before '-' token
    int main() { bool EOF = false; int sum = 0;
                ^
sum.c:5:22: error: lvalue required as left operand of assignment
        int value; EOF = (scanf("%d", &value) != 1);
                        ^
clonmel$
```

```
clonmel$ gcc -E sum.c | sed -n '/^#/d; /main/, $p'
int main() {
    _Bool (-1)
            =
            0
            ; int sum = 0;

    do {
        int value;
            (-1)
            = (scanf("%d", &value) != 1);

        if (!
            (-1)
            ) sum += value;
    } while (!
            (-1)
            ); printf("sum = %d\n", sum);
}
clonmel$
```

- Dieses Rätsel löst sich erst, wenn das Ergebnis des Präprozessors betrachtet wird. Im gesamten Programmtext wurde offenbar *EOF* durch  $(-1)$  ersetzt.

```
clonmel$ find /usr/include -type f |
> xargs perl -ne 'print "$ARGV:$_" if /^#\s*define\s*EOF\b/'
/usr/include/iso/stdio_iso.h:#define      EOF      (-1)
clonmel$
```

- Durch **#include** <stdio.h> werden direkt oder indirekt (durch weitere **#include**-Direktiven) Makros definiert, die zum Textersatz führen.
- Konkret definiert <stdio.h> das Makro *EOF* als (-1).
- Im Zweifelsfall hilft es, den Programmtext nach dem Präprozessor anzusehen und per *brute force* nach einem entsprechenden Makro zu suchen.