- 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.
- In C gibt es statt Klassen und Methoden nur globale Funktionen, die Parameter erhalten und einen Wert zurückliefern. Da die Funktionen 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. (Die Parameter für die Kommandozeilenargumente können aber weggelassen werden, wenn diese nicht benötigt werden.)

hallo.c

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

- Dieses Programm gibt den gezeigten Text mit einem abschließenden Zeilentrenner aus, – 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 verwandelt.
- Zeichenketten werden (wie in den Assembler-Programmen *hello.s* und *hello-x86.s*) durch Zeiger auf eine nullbyte-terminierte Sequenz von Zeichen im Speicher repräsentiert.

Übersetzung des ersten C-Programms


```
theon$ 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!");
    ^~~~
theon$ a.out
Hallo zusammen!
theon$
```

- 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 (u.a. der POSIX-Standard) 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), 6.3.0 (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 dies zuvor implizit angenommen.)

Übersetzung des verbesserten C-Programms

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

- 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.

Erzeugter Assemblertext: SPARC

```
        .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.

Erzeugter Assemblertext: Intel/x86


```
        .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.

quadrate.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.

quadrate.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 des entsprechenden Arguments 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.

Euklidischer Algorithmus

euklid.c

```
#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.

Einlesen mit *scanf*

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.

$$
\begin{aligned}
\langle \text{translation-unit} \rangle &\longrightarrow \langle \text{external-declaration} \rangle \\
&\longrightarrow \langle \text{translation-unit} \rangle\ \langle \text{external-declaration} \rangle \\
\langle \text{external-declaration} \rangle &\longrightarrow \langle \text{declaration} \rangle \\
&\longrightarrow \langle \text{function-definition} \rangle
\end{aligned}
$$

- 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.

Deklarationen in C

⟨declaration⟩	⟶	⟨declaration-specifiers⟩
		⟨init-declarator-list⟩ „;“
	⟶	⟨static-assert-declaration⟩
⟨declaration-specifiers⟩	⟶	⟨declaration-specifier⟩
		[⟨declaration-specifiers⟩]
⟨declaration-specifier⟩	⟶	⟨storage-class-specifier⟩
	⟶	⟨type-specifier⟩
	⟶	⟨type-qualifier⟩
	⟶	⟨function-specifier⟩
	⟶	⟨alignment-specifier⟩

- Bei einer einfachen Variablendeklaration **int** $i = 7$; ist „**int**“ der ⟨type-specifier⟩, während „$i = 7$“ zur ⟨init-declarator-list⟩ gehört.

$$
\begin{array}{rcl}
\langle \text{function-definition} \rangle & \longrightarrow & \langle \text{declaration-specifiers} \rangle \ \langle \text{declarator} \rangle \\
 & & [\ \mathit{declaration - list}] \\
 & & \langle \text{compound-statement} \rangle \\
\langle \text{compound-statement} \rangle & \longrightarrow & \text{„\{“}\ \ [\ \langle \text{block-item-list} \rangle\ ]\ \ \text{„\}“} \\
\langle \text{block-item-list} \rangle & \longrightarrow & \langle \text{block-item} \rangle \\
 & \longrightarrow & \langle \text{block-item-list} \rangle \ \langle \text{block-item} \rangle \\
\langle \text{block-item} \rangle & \longrightarrow & \langle \text{declaration} \rangle \\
 & \longrightarrow & \langle \text{statement} \rangle
\end{array}
$$

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

Beispiel für eine Funktion

euklid2.c

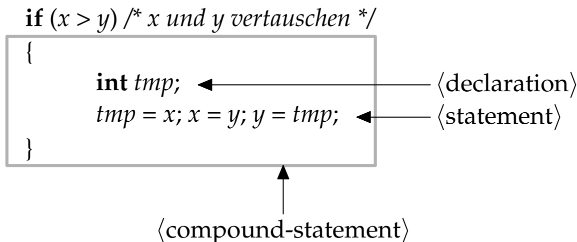
```
#include <stdio.h>

int ggt(int a, int b) {
   while (a != b) {
      if (a > b) {
         a -= b;
      } else {
         b -= a;
      }
   }
   return a;
}
int main() {
   printf("Geben Sie zwei positive ganze Zahlen ein: ");
   int x, y;
   if (scanf("%d %d", &x, &y) != 2 || x <= 0 || y <= 0) {
      return 1;
   }
   printf("ggT(%d, %d) = %d\n", x, y, ggt(x, y));
}
```

Einige Regeln zu Funktionen:

- Es gilt „*declare before use*“, d.h. Funktionen sind zu deklarieren, bevor sie benutzt werden.
- Wenn eine unbekannte Funktion benutzt wird, erfolgt eine implizite Deklaration mit **int** für den Rückgabewert und eine Parameterübergabe nach festgelegten Konventionen. Dies ist grundsätzlich zu vermeiden.
- Deswegen ist darauf zu achten, dass bei jeder Bibliotheksfunktion, die zugehörige Header-Datei mit **#include** zuvor eingebunden worden ist.
- Die Parameterübergabe erfolgt immer per *call by value*.
- Bei Arrays liegen implizit Zeiger vor, so dass bei einer Parameterübergabe nur ein Zeiger per *call by value* übergeben wird und nicht etwa der Array-Inhalt kopiert wird.

$$
\begin{array}{lcl}
\langle \text{statement} \rangle & \longrightarrow & \langle \text{expression-statement} \rangle \\
 & \longrightarrow & \langle \text{labeled-statement} \rangle \\
 & \longrightarrow & \langle \text{compound-statement} \rangle \\
 & \longrightarrow & \langle \text{conditional-statement} \rangle \\
 & \longrightarrow & \langle \text{iterative-statement} \rangle \\
 & \longrightarrow & \langle \text{switch-statement} \rangle \\
 & \longrightarrow & \langle \text{break-statement} \rangle \\
 & \longrightarrow & \langle \text{continue-statement} \rangle \\
 & \longrightarrow & \langle \text{return-statement} \rangle \\
 & \longrightarrow & \langle \text{goto-statement} \rangle \\
 & \longrightarrow & \langle \text{null-statement} \rangle
\end{array}
$$



- 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.

Initialisierung lokaler Variablen

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 dürfen lokale Variablen solange nicht verwendet werden, bis zweifelsfrei sichergestellt ist, dass sie ordentlich initialisiert 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:

```
theon$ gcc -std=gnu11 -fmessage-length=82 -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 */
        ^
theon$ varinit
-32513
theon$
```

- 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 Literalen nicht als solche erkannt.

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

- 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.