- 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.

Übersetzung des ersten C-Programms 33

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

- 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.

```
clonard$ gcc -fmessage-length=70 -Wall -std=c11 hallo.c
hallo.c:1:1: warning: return type defaults to 'int' [enabled by
   default]
 main() {
 ^
hallo.c: In function 'main':
hallo.c:3:4: warning: implicit declaration of function 'puts'
   [-Wimplicit-function-declaration]
    puts("Hallo zusammen!");
    ^
clonard$
```

- Voreinstellungsgemäß geht *gcc* von *C89* aus. Es ist auch möglich, mit der Option „*-std=c99*“ den jüngeren Standard *C99* oder mit „*-std=c11*“ den aktuellen Standard von 2011 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: 4.7.1 (Solaris/Intel, z.B. Thales), 4.7.2 (Debian, Pool in E.44) oder 4.8.0 (Solaris/SPARC, z.B. Theseus).

hallo1.c

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

int main() {
   /* puts: Ausgabe eines Strings nach stdout */
   puts("Hallo zusammen!");
   /* Programm explizit mit Exit-Status 0 beenden */
   return 0;
}
```

- 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.)
- Der Rückgabewert der *main()*-Funktion, welcher durch **return** 0 gesetzt wird, ist der *Exit-Status* des Programms. Fehlt dieser, führt dies ab *C99* implizit zu einem ein Exit-Status von 0.

```
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.

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.

Berechnung von Quadratzahlen

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.

Ausgabe mit *printf*

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 spezifizieren.
- „%4d“ bedeutet hier, dass ein Wert des Typs **int** auf eine Breite von 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 von frühen C-Versionen nicht unterstützt wurde.

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 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.
- Das Vorliegen von Einlesefehlern sollte immer überprüft werden. 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.

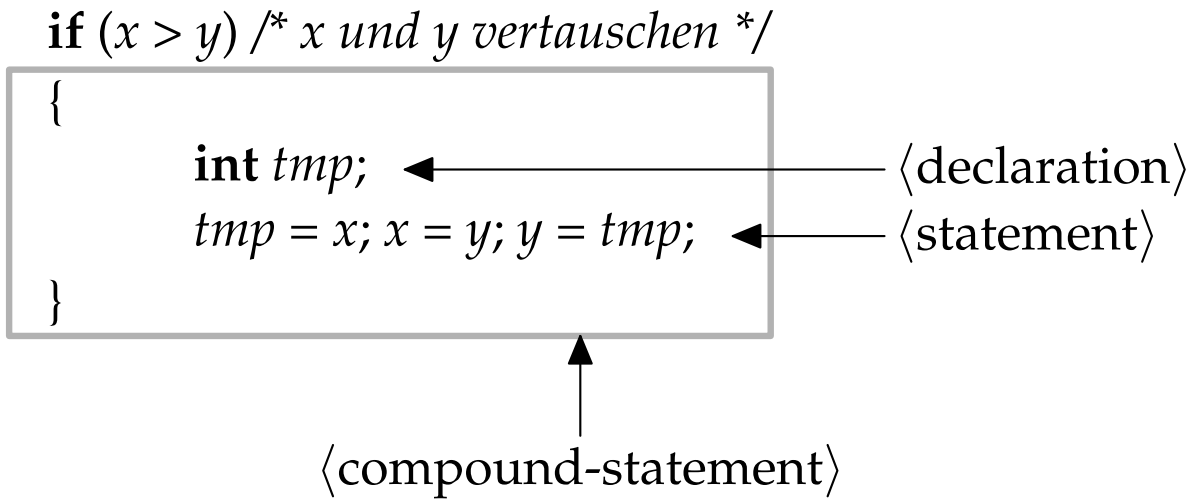
$$
\begin{array}{rcl}
\langle\text{translation-unit}\rangle & \longrightarrow & \langle\text{top-level-declaration}\rangle \\
& \longrightarrow & \langle\text{translation-unit}\rangle\ \langle\text{top-level-declaration}\rangle \\
\langle\text{top-level-declaration}\rangle & \longrightarrow & \langle\text{declaration}\rangle \\
& \longrightarrow & \langle\text{function-definition}\rangle \\
\langle\text{declaration}\rangle & \longrightarrow & \langle\text{declaration-specifiers}\rangle \\
& & \langle\text{initialized-declarator-list}\rangle\ \text{„;“} \\
\langle\text{declaration-specifiers}\rangle & \longrightarrow & \langle\text{declaration-specifier}\rangle \\
& & [\ \langle\text{declaration-specifiers}\rangle\ ] \\
\langle\text{declaration-specifier}\rangle & \longrightarrow & \langle\text{storage-class-specifier}\rangle \\
& \longrightarrow & \langle\text{type-specifier}\rangle \\
& \longrightarrow & \langle\text{type-qualifier}\rangle \\
& \longrightarrow & \langle\text{function-specifier}\rangle
\end{array}
$$

- Eine Übersetzungseinheit (*translation unit*) in C ist eine Folge von *Vereinbarungen*, zu denen Funktionsdefinitionen, Typ-Vereinbarungen und Variablenvereinbarungen gehören.

Anweisungen in C

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

$$
\begin{array}{rcl}
\langle \text{compound-statement} \rangle & \longrightarrow & \text{„}\{\text{“} \\
 & & [\ \langle \text{declaration-or-statement-list} \rangle\ ] \\
 & & \text{„}\}\text{“} \\
\langle \text{declaration-or-statement-list} \rangle & \longrightarrow & \langle \text{declaration-or-statement} \rangle \\
 & \longrightarrow & \langle \text{declaration-or-statement-list} \rangle \\
 & & \langle \text{declaration-or-statement} \rangle \\
\langle \text{declaration-or-statement} \rangle & \longrightarrow & \langle \text{declaration} \rangle \\
 & \longrightarrow & \langle \text{statement} \rangle
\end{array}
$$



- Mit **int** *tmp*; wird eine lokale Variable mit dem Datentyp **int** deklariert.
- Die Sichtbarkeit von *tmp* erstreckt sich auf den umrandeten Anweisungsblock.
- 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 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 neueren gcc-Versionen:

```
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.

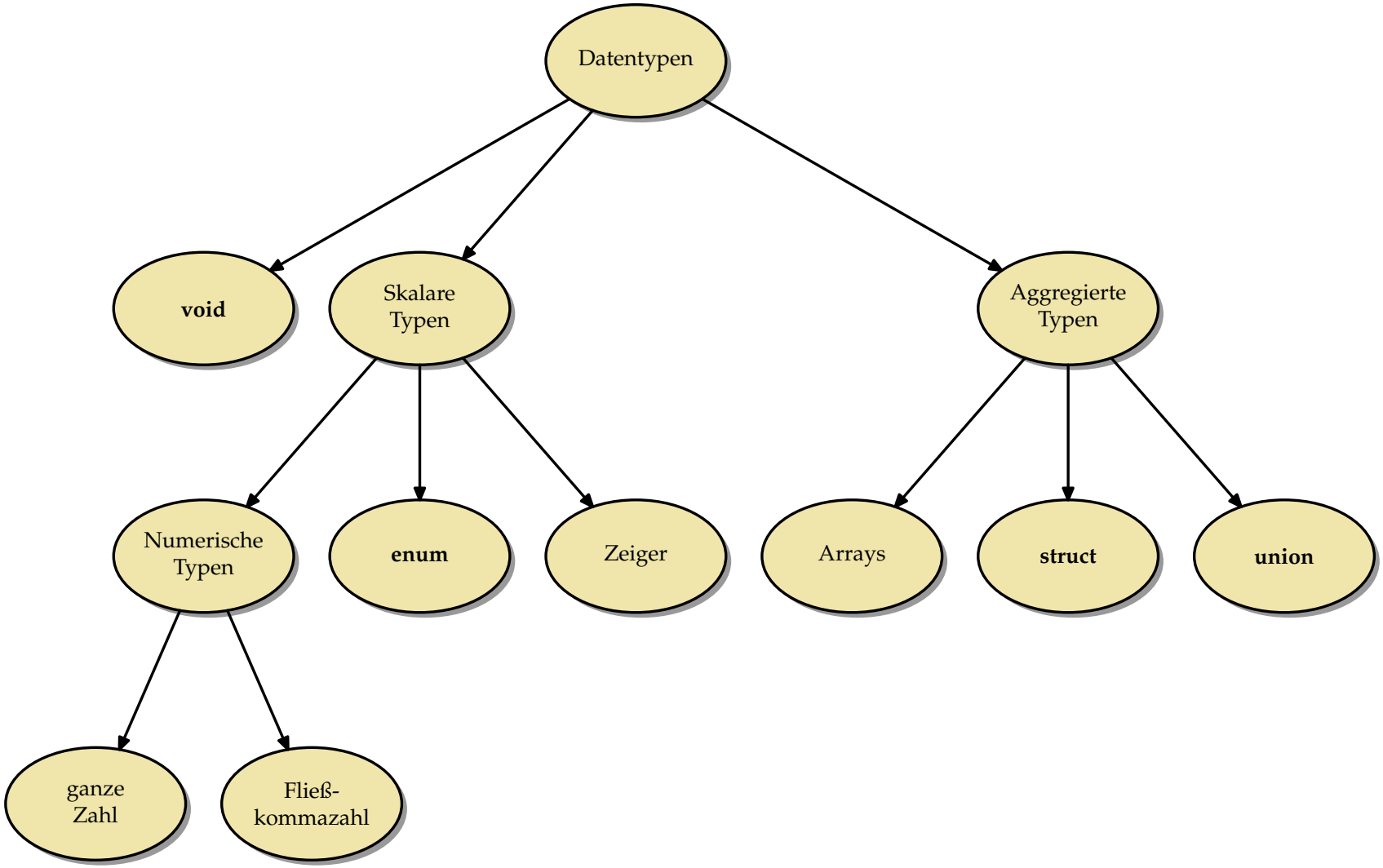
auto	**else**	**long**	**switch**	**_Atomic**
break	**enum**	**register**	**typedef**	**_Bool**
case	**extern**	**restrict**	**union**	**_Complex**
char	**float**	**return**	**unsigned**	**_Generic**
const	**for**	**short**	**void**	**_Imaginary**
continue	**goto**	**signed**	**volatile**	**_Noreturn**
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.

Datentypen legen

- den *Speicherbedarf*,
- die *Interpretation* des Speicherplatzes sowie
- die *erlaubten Operationen* fest.

Übersicht


Datentypen
void
Skalare Typen
Aggregierte Typen
Numerische Typen
enum
Zeiger
Arrays
struct
union
ganze Zahl
Fließ-kommazahl

- Im einfachsten Falle lässt sich eine Variablenvereinbarung sehr einfach zerlegen in die Angabe eines Typs und die des Variablennamens:
  int *i*;
  Hier ist *i* der Variablenname und **int** der Typ.
- Diese Zweiteilung entspricht soweit der Grammatik:

$$
\begin{array}{rcl}
\langle \text{declaration} \rangle & \longrightarrow & \langle \text{declaration-specifiers} \rangle \\
 & & [\ \langle \text{init-declarator-list} \rangle\ ] \\
\langle \text{declaration-specifiers} \rangle & \longrightarrow & \langle \text{storage-class-specifier} \rangle \\
 & & [\ \langle \text{declaration-specifiers} \rangle\ ] \\
 & \longrightarrow & \langle \text{type-specifier} \rangle \\
 & & [\ \langle \text{declaration-specifiers} \rangle\ ] \\
 & \longrightarrow & \langle \text{type-qualifier} \rangle \\
 & & [\ \langle \text{declaration-specifiers} \rangle\ ] \\
 & \longrightarrow & \langle \text{function-specifier} \rangle \\
 & & [\ \langle \text{declaration-specifiers} \rangle\ ]
\end{array}
$$

- Leider trennt die Syntax nicht in jedem Falle sauber den Namen von dem Typ.
- Beispiel:
  int∗ *ip*;
- Hier besteht die linke Seite, d.h. der ⟨declaration-specifier⟩ nur aus **int**. Der Dereferenzierungs-Operator wird stattdessen syntaktisch der rechten Seite, dem ⟨init-declarator-list⟩ zugeordnet.
- Dies hat zur Konsequenz, dass bei
  int∗ *ip1*,*ip2*;
  ip1 und *ip2* unterschiedliche Typen erhalten. So ist *ip1* ein Zeiger auf **int**, während *ip2* schlicht nur den Typ **int** hat.

- Zu den skalaren Datentypen gehören alle Typen, die entweder numerisch sind oder sich zu einem numerischen Typ konvertieren lassen.
- Ein Wert eines skalaren Datentyps kann beispielsweise ohne weitere Konvertierung in einer Bedingung verwendet werden.
- Entsprechend wird die 0 im entsprechenden Kontext auch als Null-Zeiger interpretiert oder umgekehrt ein Null-Zeiger ist äquivalent zu *false* und ein Nicht-Null-Zeiger entspricht innerhalb einer Bedingung *true*.
- Ferner liegt die Nähe zwischen Zeigern und ganzen Zahlen auch in der von C unterstützten Adressarithmetik begründet.

⟨integer-type-specifier⟩ ⟶ ⟨signed-type-specifier⟩
⟶ ⟨unsigned-type-specifier⟩
⟶ ⟨character-type-specifier⟩
⟶ ⟨bool-type-specifier⟩

⟨signed-type-specifier⟩ ⟶ [**signed**] **short** [**int**]
⟶ [**signed**] **int**
⟶ [**signed**] **long** [**int**]
⟶ [**signed**] **long long** [**int**]

⟨unsigned-type-specifier⟩ ⟶ **unsigned short** [**int**]
⟶ **unsigned** [**int**]
⟶ **unsigned long** [**int**]
⟶ **unsigned long long** [**int**]

⟨character-type-specifier⟩ ⟶ **char**
⟶ **signed char**
⟶ **unsigned char**

⟨bool-type-specifier⟩ ⟶ **_Bool**

- Die Spezifikation eines ganzzahligen Datentyps besteht aus einem oder mehreren Schlüsselworten, die die Größe festlegen, und dem optionalen Hinweis, ob der Datentyp vorzeichenbehaftet ist oder nicht.
- Fehlt die Angabe von **signed** oder **unsigned**, so wird grundsätzlich **signed** angenommen.
- Die einzigen Ausnahme hiervon sind **char** und **_Bool**.
- Bei **char** darf der Übersetzer selbst eine Voreinstellung treffen, die sich am effizientesten auf der Zielarchitektur umsetzen lässt.

Auch wenn Angaben wie **short** oder **long** auf eine gewisse Größe hindeuten, so legt keiner der C-Standards die damit verbundenen tatsächlichen Größen fest. Stattdessen gelten nur folgende Regeln:

- Der jeweilige „größere“ Datentyp in der Reihe **char**, **short**, **int**, **long**, **long long** umfasst den Wertebereich der kleineren Datentypen, d.h. **char** ist nicht größer als **short**, **short** nicht größer als **int** usw.
- Für jeden der ganzzahligen Datentypen gibt es Mindestintervalle, die abgedeckt sein müssen. (Die zugehörige Übersichtstabelle folgt.)
- Die korrespondierenden Datentypen mit und ohne Vorzeichen (etwa **signed int** und **unsigned int**) belegen exakt den gleichen Speicherplatz und verwenden die gleiche Zahl von Bits. (Entsprechende Konvertierungen erfolgen entsprechend der Semantik des Zweier-Komplements.)

In C werden alle ganzzahligen Datentypen durch Bitfolgen fester Länge repräsentiert: $\{a_i\}_{i=1}^{n}$ mit $a_i \in \{0, 1\}$. Bei ganzzahligen Datentypen ohne Vorzeichen ergibt sich der Wert direkt aus der binären Darstellung:

$$a = \sum_{i=1}^{n} a_i 2^{i-1}$$

Daraus folgt, dass der Wertebereich bei n Bits im Bereich von 0 bis $2^n - 1$ liegt.

Bei ganzzahligen Datentypen mit Vorzeichen übernimmt a_n die Rolle des Vorzeichenbits. Für die Repräsentierung gibt es bei C11 (und den früheren Standards) nur drei zugelassene Varianten:

- **Zweier-Komplement**:

$$a = \sum_{i=1}^{n-1} a_i 2^{i-1} - a_n 2^n$$

  Wertebereich: $\left[-2^{n-1}, 2^{n-1} - 1\right]$
  Diese Darstellung hat sich durchgesetzt und wird von fast allen Prozessor-Architekturen unterstützt.

- **Einer-Komplement**:

$$a = \sum_{i=1}^{n-1} a_i 2^{i-1} - a_n (2^n - 1)$$

Wertebereich: $\left[-2^{n-1}+1, 2^{n-1}-1\right]$
Vorsicht: Es gibt zwei Repräsentierungen für die Null. Es gilt:
$-a == \sim a$
Diese Darstellung gibt es auf einigen historischen Architekturen wie etwa der PDP-1, der UNIVAC 1100/2200 oder der 6502-Architektur.

- **Trennung zwischen Vorzeichen und Betrag**:

$$a = (-1)^{a_n} \sum_{i=1}^{n-1} a_i 2^{i-1}$$

Wertebereich: $\left[-2^{n-1}+1, 2^{n-1}-1\right]$
Vorsicht: Es gibt zwei Repräsentierungen für die Null.
Diese Darstellung wird ebenfalls nur von historischen Architekturen verwendet wie etwa der IBM 7090.

Was passiert bei Überläufen? 65

Was passiert bei einer Addition, Subtraktion oder Multiplikation, die den Wertebereich des jeweiligen Datentyps verlässt?

- Bei vorzeichenbehafteten ganzen Zahlen ist das Resultat undefiniert. In der Praxis bedeutet dies, dass wir die repräsentierbaren niederwertigen Bits im Zweierkomplement erhalten.
- Bei ganzen Zahlen ohne Vorzeichen stellt C sicher, dass wir das korrekte Resultat modulo 2^n erhalten.

Alle gängigen Prozessorarchitekturen erkennen einen Überlauf, aber C ignoriert dieses. Das wird in Java genauso gehandhabt.

Was passiert, wenn durch 0 geteilt wird? 66

div0.c

```
int main() {
   int i = 1; int j = 0;
   int k = i / j;
   return k;
}
```

- Dies ist generell offen.
- Es kann zu einem undefinierten Resultat führen oder zu einem Abbruch der Programmausführung.
- Letzteres ist die Regel.

```
clonard$ gcc -std=gnu11 -Wall -o div0 div0.c && ./div0
Arithmetic Exception (core dumped)
clonard$
```

Datentyp	Bits	Intervall	Konstanten
signed char	8	$[-127, 127]$	*SCHAR_MIN*, *SCHAR_MAX*
unsigned char	8	$[0, 255]$	*UCHAR_MAX*
char	8		*CHAR_MIN*, *CHAR_MAX*
short	16	$[-32767, 32767]$	*SHRT_MIN*, *SHRT_MAX*
unsigned short	16	$[0, 65535]$	*USHRT_MAX*
int	16	$[-32767, 32767]$	*INT_MIN*, *INT_MAX*
unsigned int	16	$[0, 65535]$	*UINT_MAX*
long	32	$\left[-2^{31}+1, 2^{31}-1\right]$	*LONG_MIN*, *LONG_MAX*
unsigned long	32	$[0, 4294967295]$	*ULONG_MAX*
long long	64	$\left[-2^{63}+1, 2^{63}-1\right]$	*LLONG_MIN*, *LLONG_MAX*
unsigned long long	64	$\left[0, 2^{64}-1\right]$	*ULLONG_MAX*

- Der Datentyp **char** orientiert sich in seiner Größe typischerweise an dem Byte, der kleinsten adressierbaren Einheit.
- In <*limits.h*> findet sich die Konstante *CHAR_BIT*, die die Anzahl der Bits bei **char** angibt. Dieser Wert muss mindestens 8 betragen und weicht davon auch normalerweise nicht ab.
- Der Datentyp **char** gehört mit zu den ganzzahligen Datentypen und entsprechend können Zeichen wie ganze Zahlen und umgekehrt behandelt werden.
- Der C-Standard überlässt den Implementierungen die Entscheidung, ob **char** vorzeichenbehaftet ist oder nicht. Wer sicher gehen möchte, spezifiziert dies explizit mit **signed char** oder **unsigned char**.
- Für größere Zeichensätze gibt es den Datentyp *wchar_t* aus <*wchar.h*>.

Zeichenkonstanten werden in einfache Hochkommata eingeschlossen, etwa 'a' (vom Datentyp **char**) oder *L*'a' (vom Datentyp *wchar_t*).
Für eine Reihe von nicht druckbaren Zeichen gibt es Ersatzdarstellungen:

\a	BEL	*alert*, Signalton
\b	BS	*backspace*
\f	FF	*formfeed*
\n	LF	*newline*, Zeilentrenner
\r	CR	*carriage return*, „Wagenrücklauf“
\t	HT	Horizontaler Tabulator
\v	VT	Vertikaler Tabulator
\\	\	„Fluchtsymbol“
\'	'	einfaches Hochkomma
\0	NUL	Null-Byte
\ddd		ASCII-Code (oktal)

Zeichen als ganzzahlige Werte

rot13.c

```
#include <stdio.h>

const int letters = 'z' - 'a' + 1;
const int rotate = 13;
int main() {
   int ch;
   while ((ch = getchar()) != EOF) {
      if (ch >= 'a' && ch <= 'z') {
         ch = 'a' + (ch - 'a' + rotate) % letters;
      } else if (ch >= 'A' && ch <= 'Z') {
         ch = 'A' + (ch - 'A' + rotate) % letters;
      }
      putchar(ch);
   }
}
```