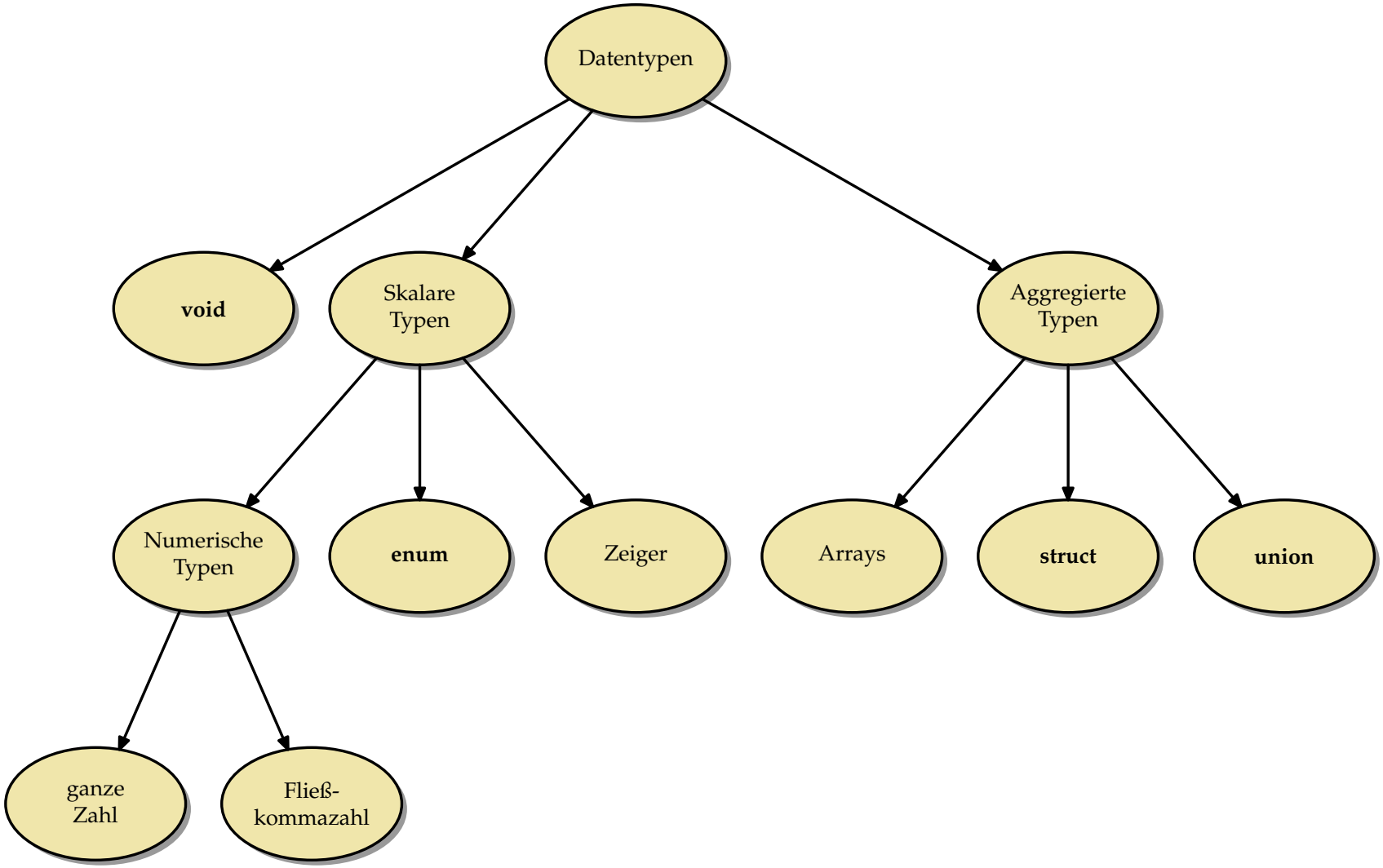
Datentypen legen

- 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, der ⟨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.

Ganzzahlige Datentypen

⟨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 Ausnahmen 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**:

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

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

- Vorsicht: Im Wertebereich ist -2^{n-1} enthalten, jedoch nicht 2^{n-1}, somit liefert das unäre Minus angewendet auf die niedrigste Zahl nicht das erwartete Resultat.

- **Einer-Komplement**:

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

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.

C unterstützt einige Bit-Operationen für ganzzahlige Datentypen:

$\sim x$	bitweises Komplement; bei ganzen Zahlen ohne Vorzeichen entspricht dies dem maximalen Wert des Typs abzüglich von x.
$x \mid y$	bitweise Oder-Operation
$x \,\&\, y$	bitweise Und-Operation
$x \,\hat{}\, y$	bitweise Exklusiv-Oder-Operation
$x << i$	Verschiebeoperation mit dem Wert $x \times 2^i$, wobei $i \geq 0$ gelten und das Ergebnis repräsentierbar sein muss, damit es wohldefiniert ist
$x >> i$	Verschiebeoperation mit dem Wert $\lfloor \frac{x}{2^i} \rfloor$. Bei vorzeichenbehafteten ganzen Zahlen lässt der Standard den Effekt undefiniert, falls $x < 0$.

Mit Hilfe der Bit-Operationen lassen sich für Mengen mit sehr kleinen Kardinalitäten ganze Zahlen ohne Vorzeichen verwenden. Seien A und B Variablen des Typs **unsigned int**, die zwei Mengen repräsentieren. Die Elemente der Menge werden durch ganze Zahlen repräsentiert von 0 bis maximal $WORD\_BIT-1$ (aus **#include** <limits.h>).

Mengenoperation	Bit-Operation in C
$A \cup B$	$A \mid B$
$A \cap B$	$A \,\&\, B$
$A \setminus B$	$A \,\&\, \sim B$
$\{i\}$	$1 << i$

Was passiert bei Überläufen?

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? 73

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*

Einige typische Konstellationen haben sich durchgesetzt:

	32-Bit	64-Bit		
		LLP	LP	ILP
sizeof(short)	2	2	2	2
sizeof(int)	4	4	4	8
sizeof(long int)	4	4	8	8
sizeof(long long int)	8	8	8	8
sizeof(void*)	4	8	8	8

- Das 32-Bit-Modell lässt sich durch die Übersetzungsoption „-m32“ auswählen. Unter Solaris ist das die Voreinstellung.
- Mit „-m64“ gibt es einen 64-Bit-Adressraum. Unter Linux ist dies die Voreinstellung. Linux, Solaris und OS X unterstützen das LP-Modell, Microsoft Windows das LLP-Modell.

Um portabel programmieren zu können ohne Kenntnis des verwendeten Modells, empfiehlt es sich, folgende vordefinierten Datentypen zu verwenden, die über <*stddef.h*> zugänglich sind:

- *size_t* ist der ganzzahlige Datentyp ohne Vorzeichen, den **sizeof** zurückliefert. Dieser Typ kann auch grundsätzlich als Index für Arrays verwendet werden. (Bei LLP oder LP kann **int** dafür zu klein sein.)
- *uintptr_t* (ohne Vorzeichen) bzw. *intptr_t* (mit Vorzeichen) können einen Zeigerwert aufnehmen.
- *ptrdiff_t* (mit Vorzeichen) wird für die Differenz zweier Zeiger zurückgeliefert.

Normalerweise haben diese ganzzahligen Typen die gleiche Größe (nämlich die von **sizeof**(**void**∗)), aber auf einigen historischen Architekturen (z.B. mit segmentierten Adressräumen) können diese voneinander abweichen.

- 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*> bzw. ab C11 über <*uchar.h*> auch *char16_t* und *char32_t*.

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)

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);
   }
}
```