ulm university universität
uulm

Vorlesungsbegleiter zu Systemnahe Software I WS 2017/2018

Andreas F. Borchert
Matthias Grabert
Johannes Mayer
Franz Schweiggert

Fakultät für Mathematik und Wirtschaftswissenschaften
Institut für Numerische Mathematik
(zuvor gemeinsam entwickelt im ehemaligen Institut für
Angewandte Informationsverarbeitung)

Hinweise:

- Auf eine detaillierte Unterscheidung zwischen *BSD-Unix*, *System-V-Unix* oder *Linux* wird hier verzichtet. Stattdessen dient der IEEE Standard 1003.1 (POSIX) weitgehend als Grundlage.
- Die enthaltenen Beispiel-Programme wurden zum großen Teil unter Linux entwickelt und sind weitestgehend unter Solaris getestet (für konstruktive Hinweise sind die Autoren dankbar).
- Die Beispiele sollen jeweils gewisse Aspekte verdeutlichen und erheben nicht den Anspruch von Robustheit und Zuverlässigkeit. Man kann alles anders und besser machen.
- Details zu den behandelten bzw. verwendeten Systemaufrufen sollten jeweils im **Manual** bzw. den entsprechenden Header-Files nachgelesen werden.
- Die Sprache C dient in erster Linie als *Werkzeug* zur Darstellung systemnaher Konzepte!

Dennis M. Ritchie, 1941–2011
Ausschnitt eines Fotos von Denise Panyik-Dale, CC-BY-2.0

Dieses Skript ist Dennis M. Ritchie gewidmet, ohne dessen Beiträge es nicht denkbar wäre. Dennis M. Ritchie hat nicht nur die Programmiersprache C entworfen und implementiert, sondern auch zusammen mit Ken Thompson Unix entwickelt. Mit C wurde erstmals eine höhere Programmiersprache statt Assembler für die Systemprogrammierung verwendet. Dennis Ritchie und seinen Kollegen gelang es bei Unix, die Betriebsystemsschnittstellen und die Systemwerkzeuge in einem bis dahin unbekanntem Maß zu vereinfachen. Auf sie geht der Grundsatz zurück, dass jedes Werkzeug genau eine Aufgabe zu erfüllen habe und das möglichst gut.

Inhaltsverzeichnis

Kapitel 1

Entstehungsgeschichte

Die Abb. 1.1 zeigt eine vereinfachte Darstellung der Entwicklungsbeziehungen einiger Programmiersprachen.

Die Programmiersprache C

- C wurde 1972–73 von Dennis Ritchie bei den Bell Laboratories von AT&T entwickelt.
- Vorbilder waren Algol, Fortran und BCPL.
- Zu den Zielsetzungen gehörte es, eine recht einfache portable maschinennahe Sprache zu entwickeln, die ohne aufwendige Laufzeitunterstützung leicht in effizienten Maschinen-Code übersetzt werden kann. Damit gelang es, UNIX weitgehend in C zu schreiben, was die spätere Portabilität von UNIX sehr erleichtert hat.
- Ähnlich wie Assembler bot C damals nur sehr wenig Überprüfungen an (praktisch keine Typüberprüfungen, keine Kontrolle von Array-Indizes oder Zeigern) und wenig Komfort – so ließen sich nur elementare Basistypen einander zuweisen.
- Viele maschinennahe Elemente aus Assembler wurden in C übernommen wie beispielsweise der Umgang mit Arrays, die als Speicherflächen betrachtet werden, auf die Zugriffe mit Hilfe von Zeigerarithmetik erfolgen.
- 1978 erfolgten einige Erweiterungen von C (*enum, void, structure assignment, …*), die mit in das erste Buch über C von Kernighan und Ritchie integriert wurden und damit den sogenannten *K&R-Standard* begründeten.
- 1983 begannen Standardisierungsbemühungen für C, die 1989 zum ANSI-Standard X3.159-1989 führten, der auch kurz ANSI-C oder C89 genannt wird. Die wichtigste Änderung war die Einführung von Funktionsprototypen, die es bei C nun erlaubten Funktionsaufrufe gegen die Deklaration einer Funktion zu überprüfen. Ein Jahr später wurde dieser Standard mit nur minimalen Veränderungen auch von ISO (als Standard 9899:1990) übernommen.
- Weitere Erweiterungen und Überprüfungsmöglichkeiten flossen in den 1999 veröffentlichten Standard ISO 9899:1999 (C99). Einige Kleinigkeiten kamen noch in den ISO-Standard 9899:2011 hinzu (C11). Mehrere Korrekturen (jedoch keine neuen Features) wurden als ISO-Standard 9899:2018 (C18) herausgegeben. Die Vorlesung beruht auf C99 bzw. C11/C18, wobei wir mit den C11-Erweiterungen kaum in Berührung kommen werden.

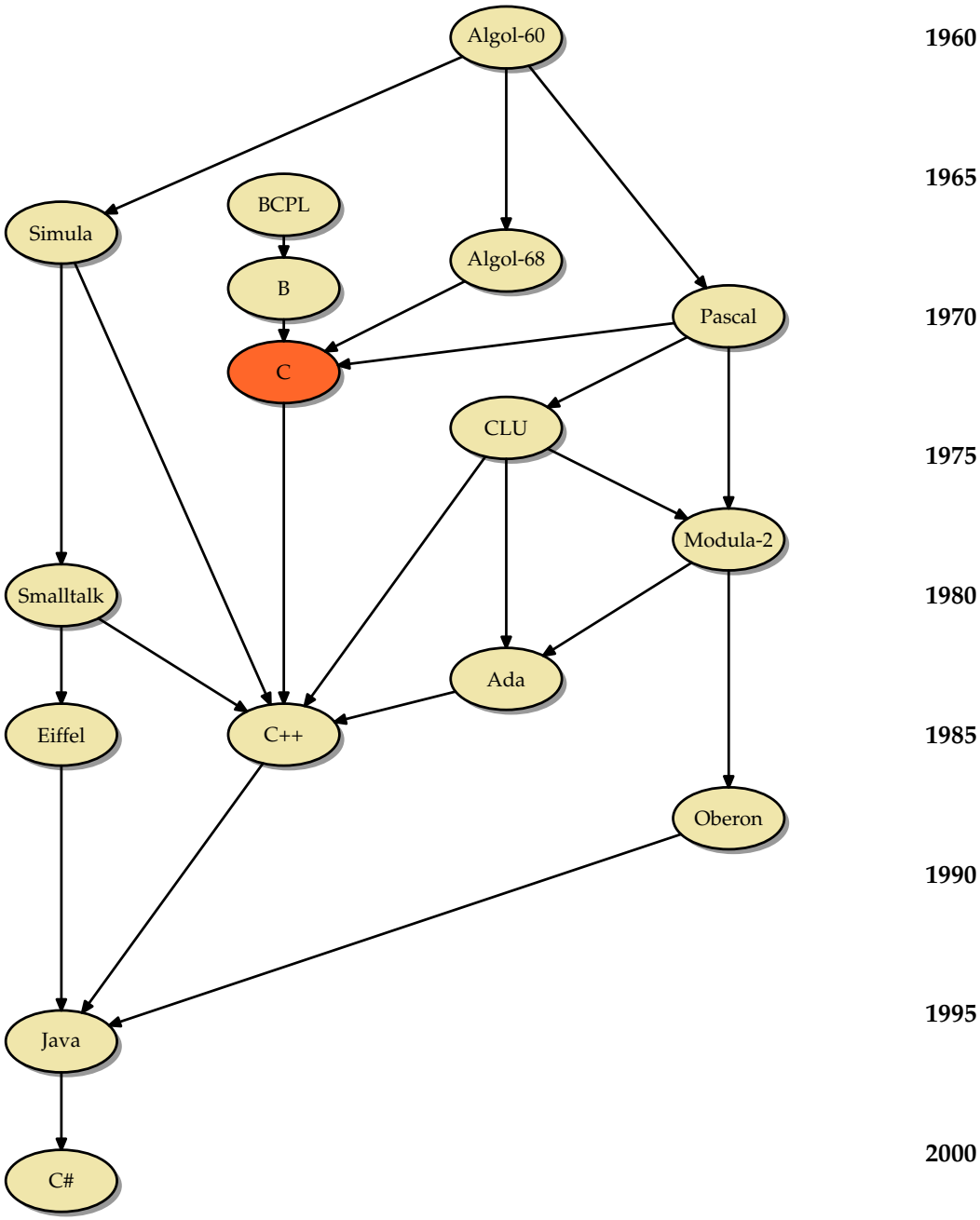

Abbildung 1.1: Entwicklungsbeziehungen einiger Programmiersprachen

- In der Vorlesung und in den Übungen wird primär mit dem *gcc* gearbeitet, der auch für Windows erhältlich ist. Hinweise dazu gibt es auf den Webseiten zur Vorlesung.
- *Literatur:* Jedes Buch, das sich mit C direkt beschäftigt wie beispielsweise [Kernighan 1990] oder [Harbison 2002]. Bücher über C++ sind aufgrund der Komplexität der objektorientierten Konzepte für die Vorlesung nicht empfehlenswert.

Andere Programmiersprachen

- C und grammatikalische Elemente aus der Programmiersprache C wurden prägend für zahlreiche weitere Programmiersprachen.
- Sehr nahe an C im Sinne der Aufwärtskompatibilität blieben die objektorientierten Erweiterungen *C++* und *Objective C*.
- Andere Sprachen haben nur Teile der Syntax übernommen (wie beispielsweise *Java*, *C#* oder *Perl*), ohne kompatibel zu sein oder die maschinennahe Denkweise zu übernehmen. So hat insbesondere Java inhaltlich nicht wenige Techniken von Eiffel oder BETA übernommen, obwohl diese beiden Sprachen syntaktisch völlig anders aussehen.

Kapitel 2

Erste Schritte mit C

Bevor wir uns mit dem Aufbau und der Syntax eines C-Programms beschäftigen, folgen zunächst einige Beispiele, um einen ersten Eindruck der Sprache C zu gewinnen.

2.1 Einige Beispiel-Programme

2.1.1 Unser erstes C-Programm

Folgendes Programm ist ein minimales „Hello World"-Beispiel in C:

Programm 2.1: Hello World – Erste Version (*hallo.c*)

```
1 main() {
2     /* puts: Ausgabe einer Zeichenkette nach stdout */
3     puts("Hallo zusammen!");
4 }
```

Übersetzung und Ausführung:

```
doolin$ gcc -Wall hallo.c
hallo.c:1: warning: return type defaults to 'int'
hallo.c: In function 'main':
hallo.c:3: warning: implicit declaration of function 'puts'
hallo.c:4: warning: control reaches end of non-void function
doolin$ a.out
Hallo zusammen!
doolin$
```

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.

Voreinstellungsgemäß geht *gcc* von *C89* aus. Es ist auch möglich, den aktuellen Standard *C99* zu verwenden, wenn dies in einer entsprechenden Option verlangt wird:

```
doolin$ gcc -Wall -std=c99 hallo.c
hallo.c:1: warning: return type defaults to 'int'
hallo.c: In function 'main':
hallo.c:3: warning: implicit declaration of function 'puts'
doolin$
```

Interessanterweise führt das hier dazu, dass die Warnung über das fehlende **return** weggefallen ist, da dies für *main()* in *C99* nicht mehr vorgeschrieben ist.

Im Vergleich zu *Java* fällt auf, dass bei *main()* die Angabe der Kommandozeilenparameter fehlt. In C ist es zulässig, dies wegzulassen, wenn diese nicht benötigt werden. Ferner fehlt die Angabe einer Klasse, eines Pakets oder eines Moduls, da all dies in C nicht existiert. Stattdessen bestehen C-Quellen im wesentlichen nur aus einer Ansammlung von Variablen- und Funktionsvereinbarungen.

2.1.2 Eine bessere Welt

An den verbliebenen Warnungen bei der Ausführung von Programm 2.1 ist zu erkennen, dass das erste Beispiel nicht ganz vollständig war. Folgendes Beispiel ist nun eine erweiterte und verbesserte Version:

Programm 2.2: Hello World – Verbesserte Version (*hallo1.c*)

```
1 #include <stdio.h> /* Standard-I/O-Bibliothek einbinden */
2
3 int main() {
4     /* puts: Ausgabe eines Strings nach stdout */
5     puts("Hallo zusammen!");
6     /* Programm explizit mit Exit-Status 0 beenden */
7     return 0;
8 }
```

Folgende Änderungen sind (gegenüber Programm 2.1) erfolgt:

- Da die Ausgabefunktion *puts()* nicht bekannt war, hat der Übersetzer geraten. Nun ist diese Funktion durch das Einbinden 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.

  Dieser wird von der *Shell* unmittelbar nach der Ausführung des Programms in der Variablen *$?* (genauer: die Variable hat den Bezeichner *?*) bereit gestellt und kann durch das Kommando **echo** *$?* angezeigt werden (das Dollarzeichen vor dem Variablennamen veranlasst die Shell, den Wert dieser Variablen zu substituieren).

  Eine normale (d.h. erfolgreiche) Beendigung wird durch den *Exit-Status* 0 signalisiert; alles andere steht für „nicht erfolgreich“ (oft: Fehler) bei der Ausführung.

Die Übersetzung und Ausführung von Programm 2.2 liefert nun:

```
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.

2.1.3 Quadratisch, praktisch, gut

Programm 2.3 berechnet die ersten 20 Quadratzahlen und gibt sie auf der Standardausgabe aus:

Programm 2.3: Berechnung von Quadratzahlen mit einer for-Schleife (*quadrate.c*)

```
1  #include <stdio.h>
2
3  const int MAX = 20; /* globale Integer-Konstante */
4
5  int main() {
6      int n; /* lokale Integer-Variable */
7
8      puts("Zahl␣|␣Quadratzahl");
9      puts("-----+-------------");
10     for (n = 1; n <= MAX; n++) {
11         printf("%4d␣|␣%7d\n", n, n * n); /* formatierte Ausgabe */
12     }
13 }
```

An obigem Programm-Beispiel lässt sich erkennen, wie *globale* Variablen und *lokale* Variablen vereinbart werden können. Die globale Variable wurde als *Konstante* definiert. Außerdem wird die Funktion *printf()* zur formatierten Ausgabe verwendet. Mit einer **for**-Schleife werden die ersten zwanzig natürlichen Zahlen durchlaufen.

Programm 2.4 ist mit einer **while**-Schleife implementiert, die äquivalent zur zuvor vorgestellten **for**-Schleife aus Programm 2.3 ist.

Programm 2.4: Berechnung von Quadratzahlen mit einer while-Schleife (*quadrate1.c*)

```
1  #include <stdio.h>
2
3  const int MAX = 20;
4
5  int main() {
6      int n;
7
8      puts("Zahl␣|␣Quadratzahl");
9      puts("-----+-------------");
10     n = 1; /* wird vor dem ersten Durchlauf ausgefuehrt */
11     while (n <= MAX) { /* Bedingung wird vor jedem Durchlauf getestet */
12         printf("%4d␣|␣%7d\n", n, n * n);
13         n = n + 1; /* wird nach jedem Durchlauf ausgefuehrt */
14     }
15 }
```

2.1.4 Euklidischer Algorithmus

Programm 2.5 implementiert den bekannten Euklidischen Algorithmus zur Bestimmung des *größten gemeinsamen Teilers* zweier natürlicher Zahlen; hier wird die Funktion *scanf()* zum Einlesen der beiden Zahlen von der Standardeingabe benutzt.

Programm 2.5: Euklidischer Algorithmus (*euklid.c*)

```
1  #include <stdio.h>
2
3  int main() {
4      int x, y, x0, y0;
5
6      printf("Geben␣Sie␣zwei␣positive␣ganze␣Zahlen␣ein:");
7      /* das Resultat von scanf ist die
8          Anzahl der eingelesenen Zahlen
9      */
10     if (scanf("%d␣%d", &x, &y) != 2) { /* &-Operator konstruiert Zeiger */
11         return 1; /* Exit-Status ungleich 0 => Fehler */
12     }
13
14     x0 = x; y0 = y;
15
16     while (x != y) {
17         if (x > y) {
18             x = x - y;
19         } else {
20             y = y - x;
21         }
22     }
23
24     printf("ggT(%d,␣%d)␣=␣%d\n", x0, y0, x);
25
26     return 0;
27 }
```

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. Mit dem Operator & wird hier jeweils ein *Zeiger* auf die folgende Variable „konstruiert". Der Wert eines Zeigers ist die *Hauptspeicher-Adresse* der Variablen, auf die er zeigt (daher wird in diesem Zusammenhang der Operator & auch als *Adressoperator* bezeichnet).

Damit ist der Zeigerwert (= Adresse) zwar lokal zu *scanf()*, jedoch kann dadurch (innerhalb von *scanf()*) auf die lokalen Variablen in *main()* „durchgegriffen" werden. Auf diese Weise kann *scanf()* die eingelesenen Zahlen in *x* und *y* ablegen. Dies mag hier genügen – später beschäftigen wird uns noch ausführlich mit Zeigern.

Programm 2.6 demonstriert die Erstellung und Verwendung von Funktionen in C.

Programm 2.6: Euklidischer Algorithmus als Funktion (*euklid1.c*)

```
1  #include <stdio.h>
2
3  int ggt(int x, int y) {
4      while (y != 0) {
```

```
5           int tmp = x % y; /* Divisionsrest == wiederholte Subtraktion */
6           x = y; y = tmp;
7       }
8       return x;
9   }
10
11  int main() {
12      int x, y;
13
14      printf("Geben␣Sie␣zwei␣positive␣ganze␣Zahlen␣ein:");
15      if (scanf("%d␣%d", &x, &y) != 2) /* &−Operator konstruiert Zeiger */
16          return 1; /* Exit−Status ungleich 0 => Fehler */
17
18      printf("ggT(%d,␣%d)␣=␣%d\n", x, y, ggt(x, y));
19
20      return 0;
21  }
```

Die Berechnung des ggT wurde einfach vom Hauptprogramm in die (neu angelegte) Funktion *ggt()* „ausgelagert“. Aufgrund der Werteparameter-Semantik bei Funktionsaufrufen müssen wir die Eingaben x und y nicht mehr kopieren (wie im vorigen Beispiel).

2.2 Aufbau eines C-Programms

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

⟨translation-unit⟩	⟶	⟨top-level-declaration⟩
	⟶	⟨translation-unit⟩ ⟨top-level-declaration⟩
⟨top-level-declaration⟩	⟶	⟨declaration⟩
	⟶	⟨function-definition⟩
⟨declaration⟩	⟶	⟨declaration-specifiers⟩
		⟨initialized-declarator-list⟩ „;“
⟨declaration-specifiers⟩	⟶	⟨storage-class-specifier⟩ [⟨declaration-specifiers⟩]
	⟶	⟨type-specifier⟩ [⟨declaration-specifiers⟩]
	⟶	⟨type-qualifier⟩ [⟨declaration-specifiers⟩]
	⟶	⟨function-specifier⟩ [⟨declaration-specifiers⟩]

Hinweis: Die hier und im Folgenden vorgestellten Auszüge der Grammatik wurden weitgehend [Harbison 2002] entnommen und entsprechen dem Stand von *C99*.

2.2.1 Anweisungsblöcke

Wie in Java unterstützt C eine *Blockstruktur* in Form eines *Anweisungsblocks* (*compound statement*). Variablen-Vereinbarungen dürfen an beliebiger Stelle eines *Anweisungsblocks* stehen und sind dann bis zum Ende des jeweiligen Blocks sichtbar. (Zu beachten ist, dass dies

erst ab *C99* gilt, bei *C89* sind Vereinbarungen nur zu Beginn des Blocks zulässig.) Anweisungsblöcke erlauben es, mehrere Anweisungen zusammenzufassen und *Sichtbarkeits-/Lebensdauerbereiche* zu definieren; siehe Abb. 2.1 .

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

⟨compound-statement⟩ ⟶ „{“ [⟨declaration-or-statement-list⟩] „}“

⟨declaration-or-statement-list⟩ ⟶ ⟨declaration-or-statement⟩
⟶ ⟨declaration-or-statement-list⟩
⟨declaration-or-statement⟩

⟨declaration-or-statement⟩ ⟶ ⟨declaration⟩
⟶ ⟨statement⟩

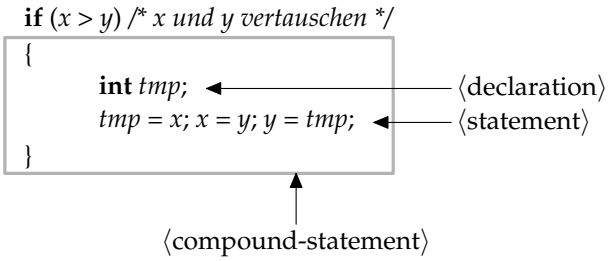

Abbildung 2.1: Anweisungsblock

Anmerkungen zu Abb. 2.1 :

- Die *Gültigkeit* von *tmp* erstreckt sich auf den umrandeten Anweisungsblock.
- Mit **int** *tmp*; wird eine lokale Variable mit dem Datentyp **int** deklariert. **int** ist ein Schlüsselwort und steht für *integer*, also eine ganze Zahl.
- Für *Zuweisungen* wird in C analog zu Java der Operator = verwendet. Als *Vergleichsoperator* kommt (wie auch in Java) zum Einsatz.

Auf die Verwendung eines Anweisungsblocks könnte bei Schleifen verzichtet werden, wenn der Schleifeninhalt ohnehin nur aus einer einzigen Anweisung besteht. Dennoch empfiehlt sich die konsequente Benutzung geschweifter Klammern, um die Lesbarkeit zu

erhöhen, Mehrdeutigkeiten zu vermeiden (wie beispielsweise beim *dangling else*) und das Hinzufügen weiterer Anweisungen zu vereinfachen. Für Java wird dies aus den gleichen Gründen analog empfohlen [Sun 1999].

2.2.2 Kommentare

Kommentare beginnen mit „/*“, enden mit „*/“, und dürfen nicht geschachtelt werden. Alternativ kann seit C99 ein Kommentar auch mit „//“ begonnen werden, der sich bis zum Zeilenende erstreckt.

2.2.3 Namen/Bezeichner

Namen bzw. *Bezeichner* bestehen aus Buchstaben und Ziffern, wobei das erste Zeichen ein Buchstabe sein muss. Zu den Buchstaben wird auch der *Unterstrich* „_“ gezählt.

2.2.4 Schlüsselworte

Die folgende Tabelle enthält alle *Schlüsselworte* von C99:

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

Einige der Schlüsselwörter wie etwa **auto** oder **register** existieren nur noch aus Kompatibiltätsgründen zu früheren Versionen und einige (**_Bool**, **_Complex** und **_Imaginary**) werden nur intern verwendet. So steht beispielsweise der üblicherweise verwendete Typname *bool***bool** zur Verfügung, wenn die entsprechende Header-Datei mit **#include** <stdbool.h> eingebunden wird.

Bei C11 kommen noch folgende Schlüsselwörter hinzu:

_Alignas	**_Noreturn**
_Alignof	**_Static_assert**
_Atomic	**_Thread_local**
_Generic	

2.2.5 Leerzeichen

Zu den als Trenner dienenden *Leerzeichen* (zusammenfassend auch Leerraum bzw. im Englischen *white space characters* genannt) gehören neben dem eigentlichen Leerzeichen auch Tabulatoren (horizontal und vertikal), Zeilentrenner (*line feed*) und der Seitenvorschub (*form feed*).

Diese werden in C nicht weiter voneinander unterschieden, d.h. jede Sequenz von Leerzeichen ist äquivalent zu einem einzelnen Leerzeichen. Die Ausnahmen hiervon sind nur die Kommentare, die mit „//“ beginnen und die (noch einzuführenden) Direktiven des Präprozessors, die jeweils durch das Zeilenende beendet werden.

Kapitel 3

Ein erster Blick auf den Präprozessor

3.1 Makroprozessoren

Makroprozessoren sind Programme, die weitgehend den Eingabetext in den Ausgabetext unverändert kopieren, jedoch selektiv einzelne Zeichenfolgen durch andere Texte ersetzen. Die Ersetzungen werden über (üblicherweise parametrisierbare) Makros definiert.

Makros erlauben es, den Schreibaufwand zu reduzieren, die Duplikation von Texten zu vermeiden, Programmtexte übersichtlicher zu gestalten und künftige Änderungen zu erleichtern. Aus diesem Grunde waren Makroprozessoren schon lange vor C in Verbindung mit Assembler und anderen Programmiersprachen im Einsatz. Neben auf bestimmte Sprachen spezialisierten Makroprozessorn gibt es auch sprachunabhängige Prozessoren wie etwa *m4*:

```
doolin$ cat morgen.m4
define(`gm', `Guten Morgen, $1!')dnl
gm(Anne)
Guten Abend, Marianne!
gm(Heinz)
doolin$ m4 morgen.m4
Guten Morgen, Anne!
Guten Abend, Marianne!
Guten Morgen, Heinz!
doolin$
```

3.2 Integration eines Makroprozessors

Die Integration eines Makroprozessors in die Programmiersprache C erlaubte es, in der eigentlichen Sprache auf fortgeschrittene Techniken zu verzichten, die teilweise aufwendig zu implementieren gewesen wären:

- Die ursprüngliche Programmiersprache (entsprechend dem K&R-Standard) verzichtete auf Konstantendefinitionen. Stattdessen wurden entsprechende Makros verwendet.
- C verzichtet auf ein Modulkonzept. Stattdessen werden die Funktionsdeklarationen in separate Dateien ausgelagert, die dann mit *include*-Anweisungen des Präprozes-

sors in alle C-Quellen hineinkopiert werden, die diese benötigen. Entsprechend bleibt dem Übersetzer die Aufgabe erspart, Bibliotheken ausfindig zu machen.

- Generische Programmiertechniken (*templates*) und eingebettete Funktionen (*inline functions*) lassen sich ebenfalls in rudimentärer Form auf Basis von Makros realisieren.

Die vereinfachte Implementierung dieser Techniken zieht jedoch auch wesentliche Nachteile mit sich:

- Statt sich auf eine in sich konsistente Sprache zu beschränken, werden zwei voneinander unabhängige Sprachen miteinander vermischt: Die „Wirtssprache“ (wie etwa C) und die Sprache des Präprozessors. Das schafft viele Probleme, da sich beispielsweise die Fehlermeldungen des Übersetzers auf die Ausgabe des Präprozessors beziehen und nicht etwa auf die ursprüngliche Quelle.
- Die Semantik der Parameterübergabe bei einem Makro (*call by text*) weicht dramatisch ab von der Semantik der Parameterübergabe der Wirtssprache (bei C *call by value*).

Bjarne Stroustrup, der Autor der nachfolgenden Programmiersprache C++, identifiziert in [Stroustrup 1994] den in C integrierten Makroprozessor (kurz Cpp genannt) als eines der Hauptprobleme:

> The character and file orientation of the preprocessor is fundamentally at odds with a programming language designed around the notions of scopes, types, and interfaces.
> ...
> I'd like to see Cpp abolished.

Die neueren Standards für C bemühen sich, einige der Techniken direkt in C anzubieten, die zuvor nur über den Cpp zur Verfügung standen:

- C89 führte Konstantendefinitionen ein.
- C99 unterstützt eingebettete Funktionen.

Entsprechend sollte die Benutzung des Präprozessors minimiert werden.

3.3 Cpp – der C-Präprozessor

Dem eigentlichen C-Compiler ist der Präprozessor *cpp* vorgeschaltet. Dieser wird automatisch mit dem Aufruf von *gcc* (und jedem anderen C-Compiler) aktiviert. Er kann auch direkt aufgerufen werden mit *cpp* oder *gcc -E* (siehe *man gcc* bzw. *man cpp*).

Direktiven des Cpp beginnen direkt am Zeilenanfang (ohne führende Leerzeichen!) mit einem #. Danach folgen der Name der Direktive und die zugehörigen Parameter. Zwischen dem führenden # und dem Namen der Direktive sind (abgesehen vom Zeilentrenner) beliebig viele Leerzeichen zulässig.

3.4 define-Direktive

Makros werden mit der Direktive *define* definiert. Makrodefinitionen selbst werden durch den Cpp durch eine Leerzeile ersetzt.

Programm 3.1: Verwendung der define-Direktive (*makros.c*)

```
1 #define MAX 10 /* so kann eine Konstante definiert werden */
2 #define MAX1 = 10 /* so nicht! */
3 #define MAX2 10; /* so auch nicht! */
4
5 int main() {
6       int x = MAX; /* OK: int x = 10; */
7       int y = MAX1; /* FALSCH: int y = = 10; */
8       int z = MAX2 + 1; /* FALSCH: int z = 10; + 1; */
9 }
```

Das Beispiel Programm 3.1 zeigt einige der Fallstricke beim Umgang mit der *define*-Direktive. Der *Cpp* liefert zu diesem Programm folgende Ausgabe:

```
doolin$ gcc -E makros.c
# 1 "makros.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "makros.c"




int main() {
    int x = 10;
    int y = = 10;
    int z = 10; + 1;
}
doolin$
```

Wenn die Kommentare bei der Ausgabe erhalten bleiben sollten, so empfiehlt sich die Option *-C*: *gcc -E -C makros.c*. Die Angaben der Zeilennummern lassen sich mit der Option *-P* unterdrücken:

```
doolin$ gcc -E -P makros.c




int main() {
    int x = 10;
    int y = = 10;
    int z = 10; + 1;
}
doolin$
```

Welche *Unterschiede* gibt es zwischen *Variablen* und *Makros*?

Beispiele:

- *Variable:* **const int** *MAX* = 3;
- *Makro:* **#define** MAX 3

Eine Variable ist ein Name für eine Speicherstelle. Die 3 steht also irgendwo im Speicher. Dagegen wird beim Makro nur die 3 an der Stelle eines Makroaufrufs eingesetzt. Entsprechend findet sich das Makro auch nicht mehr zur Laufzeit.

3.5 include-Direktive

Der Ersatztext einer *include*-Direktive ist der Inhalt der genannten Datei. Damit werden i. A. Vereinbarungen oder andere Direktiven in die zu übersetzende Quelle „hereinkopiert". Diese Dateien heißen im Kontext von C-Programmen *Header-Dateien* (bzw. *header files*) und haben üblicherweise die Endung *.h*.

Es gibt eine ganze Reihe solcher Header-Dateien im Verzeichnis */usr/include* – diese sind die Schnittstellen der C-Bibliothek und entsprechen (im Rahmen der beschränkten Möglichkeiten von C) den öffentlichen Teilen einer Java-Bibliothek. Dies ist, wie später gezeigt wird, auch der Weg zur Modularisierung in C.

Die folgenden beiden Dateien sind ein Beispiel für die Verwendung der *include*-Direktive:

Programm 3.2: Verwendung der *include*-Direktive (*makros1.c*)

```
1 #include "defs.h" /* Einfuegen von defs.h */
2
3 int main() {
4     int x = MAX;
5 }
```

Programm 3.3: Eine winzige Header-Datei (*defs.h*)

```
1 #define MAX 3
2 int y = MAX;
```

Der *Cpp* liefert zu diesem Programm nun die folgende Ausgabe:

```
doolin$ gcc -E makros1.c
# 1 "makros1.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "makros1.c"
# 1 "defs.h" 1

int y = 3;
# 2 "makros1.c" 2

int main() {
    int x = 3;
}
doolin$
```

Kapitel 4

Ein- und Ausgabe

4.1 *stdin, stdout* und *stderr*

Standardmäßig gibt es drei Kanäle für die Ein- und Ausgabe. Die *Standardeingabe* (*stdin*) entspricht der Eingabe auf der Konsole. Entsprechend ist die *Standardausgabe* (*stdout*) die „normale“ Ausgabe auf der Konsole. Mit *stderr* wird die *Fehler- bzw. Diagnoseausgabe* bezeichnet. In der Shell (= Kommandozeile) kann *stdin* mittels <, *stdout* mittels > und *stderr* mittels 2>*err* umgelenkt werden:

Programm 4.1: Ausgabe mit *puts()* und *fputs()* (*out.c*)

```
1 #include <stdio.h>
2
3 int main() {
4     /* puts gibt am Ende einen Zeilentrenner aus */
5     puts("Ich komme nach stdout ...");
6     /* fputs fuegt keinen Zeilentrenner hinzu */
7     fputs("Ich komme nach stderr ...\n", stderr);
8 }
```

```
thales$ gcc -Wall out.c
thales$ a.out
Ich komme nach stdout ...
Ich komme nach stderr ...
thales$ a.out > out.stdout 2> out.stderr
thales$ cat out.stdout
Ich komme nach stdout ...
thales$ cat out.stderr
Ich komme nach stderr ...
thales$
```

4.2 Ausgabe nach stdout

Die Funktion *puts()* gibt eine Zeichenkette auf die Standardausgabe aus:

int *puts*(**const char*** *s*);

Der *Rückgabewert* von *puts()* ist die Anzahl der geschriebenen Zeichen. Nach der Ausgabe der Zeichenkette gibt *puts()* noch einen zusätzlichen Zeilentrenner aus – im Gegensatz zur Funktion *printf()*.

Die Funktion *printf()* kann formatiert in die Standardausgabe schreiben:

int *printf*(**const char*** *format*, /**args* */...);

- *printf()* liefert die *Anzahl* der ausgegebenen Zeichen zurück.

- Ein *Format* ist eine Zeichenfolge, die aus direkt auszugebendem Text und eingestreuten Platzhaltern besteht. Beispiel:
  printf("Ich␣bin␣%d␣Jahre␣alt.", 3);
  Hier ist *%d* der Platzhalter.

- Von Ausnahmen abgesehen benötigen Platzhalter jeweils einen oder auch mehrere Parameter. Diese werden nacheinander der Parameterliste hinter dem Format entnommen. Fehlen am Ende die Parameter zu einem der Platzhalter, so ist das Resultat nicht definiert.

- Platzhalter bestehen aus dem Zeichen %, den Optionen (*Flags*), einer Mindestbreite (*MinWidth*), der Genauigkeit (*Precision*) und einem Zeichen, das die Art des Platzhalters bzw. der durchzuführenden Konvertierung beschreibt (*ConvChar*):

⟨ConvSpec⟩	⟶	„%“ { ⟨Flag⟩ } [⟨MinWidth⟩] [„.“ ⟨Precision⟩] ⟨SizeModifier⟩ ⟨ConvChar⟩
⟨Flag⟩	⟶	„-“ | „+“ | „ “ | „#“ | „0“ | ...
⟨MinWidth⟩	⟶	⟨Digit⟩ { ⟨Digit⟩ } | „*“
⟨Precision⟩	⟶	⟨Digit⟩ { ⟨Digit⟩ } | „*“
⟨SizeModifier⟩	⟶	„ll“ | „l“ | „L“ | „h“ | „hh“ | „j“ | „z“ | „t“
⟨ConvChar⟩	⟶	„d“ | „i“ | „o“ | „u“ | „x“ | „X“ | ...

  Beispiele für Platzhalter: *%-3s*, *%d*, *%3.2f*

- * als minimale Breite oder Genauigkeit bedeutet, dass der nächste Parameter aus der verbliebenen Parameterliste als ganze Zahl behandelt wird, die die minimale Breite bzw. Genauigkeit in variabler Weise angibt.

Es gibt unter anderem folgende *Konvertierungszeichen* (*ConvChar*):

Zeichen für die Konvertierung	Effekt
`d,i,o,u,x,X`	ganze Zahl: `d`, `i` und `u` für Dezimaldarstellung, wobei `u` von einer Darstellung ohne Vorzeichen ausgeht (*unsigned*); `x,X` für eine hexadezimale Darstellung, wobei `x` die Kleinbuchstaben „a"'–"'f" und `X` die Großbuchstaben „A"'–"'F" benutzt; unmittelbar davor kann ein `h` (für **short int**) oder `l` (für **long int**) stehen
`f,F`	Reelle Zahlen in Gleitkommadarstellung ([-]mmm.nnnnnnn): die Anzahl der Nachkommastellen wird durch die Genauigkeitsangabe festgelegt; Voreinstellung: 6
`e,E`	Reelle Zahlen in Exponentialdarstellung; der Exponent beginnt mit „e"' oder "'E" in Abhängigkeit von dem Konvertierungszeichen
`g,G`	Reelle Zahlen entweder in Gleitkomma- oder Exponentialdarstellung in Abhängigkeit von der darzustellenden Zahl und der gewählten Genauigkeit
`a,A`	Reelle Zahlen in Exponential- und Hexadezimaldarstellung; dies kann sinnvoll sein, um Rundungsfehler bei der Ausgabe zu vermeiden
`c`	Zeichen: der zugehörige Parameter ist eine ganze Zahl, die implizit zu einem **unsigned char** konvertiert wird
`s`	Zeichenketten: der Parameter muß eine durch ein Null-Byte terminierte Zeichenfolge sein; eine Genauigkeitsangabe wird als maximal auszugebende Zeichenzahl interpretiert
`p`	Zeiger: die Adresse wird numerisch in einem systemabhängigen Format (typischerweise hexadezimal) ausgegeben
`%`	die Folge %% gibt ein % aus

Für die die Optionen (*Flags*) gibt u. a. folgende Wahlmöglichkeiten:

Option	Bedeutung
-	linksbündige Ausgabe
+	auch ein positives Vorzeichen wird ausgegeben
Leerzeichen	statt dem pos. Vorzeichen wird ein Leerzeichen ausgeben
0	statt Leerzeichen wird der Freiraum mit Nullen aufgefüllt
#	alternative Ausgabevariante: in Kombination mit „o" führt dies zur Ausgabe einer führenden 0, bei „x"' wird "'0x" zu Beginn ausgegeben und bei reellen Zahlen wird immer ein Dezimalpunkt ausgegeben, selbst wenn dem keine Ziffern mehr folgen.

Bei numerischen Datentypen ist es notwendig, den genauen Parametertyp zu spezifizieren (*SizeModifier*), wenn es nicht um **int** oder **double** handelt:

Zeichen	Datentyp
ll	**long long int** oder **unsigned long long int**
l	**long int** oder **unsigned long int**
L	**long double**
h	**short int** oder **unsigned short int**
hh	**char** oder **unsigned char** (ab *C99*)
j	*intmax_t* oder *uintmax_t*
z	*ssize_t* oder *size_t*
t	*ptrdiff_t*

Programm 4.2 veranschaulicht die Verwendung von *printf()* und *puts()*:

Programm 4.2: Ausgabe mit *puts()* und *printf()* (*stdout.c*)

```
1  #include <stdio.h> /* enthaelt die Deklarationen aller Ein-/Ausgabe-Funktionen */
2
3  int main() {
4      puts("----+----+----+----+----+----+"); /* Lineal ;-) (ohne "\n"!)
           */
5
6      printf("%s_\n", "Donaudampfschiff"); /* "\n" erzeugt Zeilenumbruch */
7      printf("%20s_\n", "Donaudampfschiff"); /* min. Breite */
8      printf("%-20s_\n", "Donaudampfschiff"); /* min. Breite + linksbuendig */
9      printf("%.10s_\n", "Donaudampfschiff"); /* max. Breite */
10     printf("%-10.10s_\n", "Donau");  /* min. & max. Breite + linksbuendig */
11
12     puts("----+----+----+----+----+----+");
13
14     printf("%d_\n", 254); /* dezimal */
15     printf("%5d_\n", 254); /* dezimal (mit min. Breite) */
16     printf("%x_\n", 254); /* hexadezimal */
17
18     puts("----+----+----+----+----+----+");
19
20     printf("%f_\n", 3.1415926);  /* Fliesskomma */
21     printf("%10f_\n", 3.1415926); /* min. Breite */
22     printf("%.3f_\n", 3.1415926); /* Anzahl der Nachkommastellen */
23     printf("%10.3f_\n", 3.1415926); /* min. Breite + Anz. d. Nachkommast. */
24     printf("%+10.3f_\n", 3.1415926); /* Vorzeichen immer anzeigen */
25 }
```

```
thales$ gcc -Wall -std=c99 stdout.c
thales$ a.out
----+----+----+----+----+----+
Donaudampfschiff_
    Donaudampfschiff_
Donaudampfschiff    _
Donaudampf_
Donau     _
----+----+----+----+----+----+
254_
  254_
fe_
----+----+----+----+----+----+
3.141593_
  3.141593_
3.142_
     3.142_
    +3.142_
thales$
```

Weitere Hinweise zu *printf()* und *puts()* finden sich in den zugehörigen Manualseiten (*man -s 3c printf* (Solaris), *man 3 printf* (Linux) bzw. *man puts*).

4.3 Ausgabe nach stderr

Die Funktion *fputs()* gibt die Zeichenkette *s* in die Dateiverbindung *stream* – in unserem Fall *stderr* – aus:

int *fputs*(**const char*** *s*, *FILE** *stream*);

Der einzige Unterschied zu *puts()* – abgesehen von dem zusätzlichen Argument – ist, dass von *fputs()* nur die übergebene Zeichenkette ausgegeben wird, wohingegen *puts()* noch einen Zeilenumbruch anhängt.

Bsp.: *fputs*("Hallo", *stderr*);

Die Funktion *fprintf()* kann analog zu *printf()* formatiert ausgeben.

int *fprintf*(*FILE** *stream*, **const char*** *format*, /**args* */...);

Im Unterschied zu *printf()* erwartet *fprintf()* noch die Angabe einer Dateiverbindung. Für Diagnosemeldungen ist in <*stdio.h*> die Variable *stderr* definiert (manchmal auch als Makro).

Bsp.: *fprintf*(*stderr*, "Hallo");

4.4 Eingabe von stdin

Die Funktion *scanf()* liest formatiert von der Standardeingabe ein:

int *scanf*(**const char*** *format*, ...);

- Das Format besteht aus Zeichen für die Konvertierung und weiteren Zeichen.
- Konvertierung:
  % gefolgt von optionalen Zeichen zur Modifikation, gefolgt von einem Konvertierungszeichen
- Andere Zeichen (außer Konvertierungszeichen und Leerzeichen) müssen mit den Zeichen im Eingabestrom übereinstimmen. Leerzeichen, Tabs (\t) und Zeilentrenner (\n), veranlassen *scanf()* alle Leerzeichen der Eingabe zu überlesen.
- Zu beachten ist, dass C nur die Werteparameterübergabe unterstützt. Aus diesem Grund muss bei *scanf()* der *Adressoperator* vor dem einzulesenden Parameter eingefügt werden. (Bei Arrays und Zeigern fällt das weg – dazu später mehr.) Damit wird ein Zeiger auf die Variable übergeben und *scanf()* greift über diesen Zeiger auf die Variable durch.

Bsp.: *scanf*("%d", &*n*); liest in die ganzzahlige Variable *n* einen Wert von der Standardeingabe (*stdin*) ein.

scanf() hat einen ganzzahligen Rückgabewert, der die Anzahl der tatsächlich erfolgten Variablenzuweisungen wiedergibt.

Konvertierungszeichen	Wirkung
d	Dezimal-Konstante
o	Oktal-Konstante
x,X	Hexadezimal-Konstante
f	Gleitkommazahl
s	Zeichenfolge bis zum nächsten Leerzeichen (einschließlich '\t' oder '\n'); ein Null-Byte ('\0') wird angefügt
c	Nächstes Zeichen; um das nächste nicht-leere Zeichen zu lesen: %1c
%	Liest %-Zeichen ohne Zuweisung

Vor dem Konvertierungszeichen kann mit einer Dezimalzahl die maximale Feldlänge spezifiziert werden. Wenn ein Stern vor dem Konvertierungszeichen angegeben wird, unterbleibt die Zuweisung.

Programm 4.3: Eingabe mit *scanf()* (*scanf.c*)

```
1  #include <stdio.h>
2
3  int main() {
4      int anzahl, i, j;
5      float f;
6      char s[50];
7
8      anzahl = scanf("i=%d␣%f␣%s", &i, &f, s);
9      puts("------------------------------------");
10     printf("Anzahl:␣%d␣|␣i=%d,␣f=%f,␣s=%s\n", anzahl, i, f, s);
11     puts("------------------------------------");
12
13     anzahl = scanf("%2s␣%*d␣%2d", s, &i);
14     puts("------------------------------------");
15     printf("Anzahl:␣%d␣|␣s=%s,␣i=%d\n", anzahl, s, i);
16     puts("------------------------------------");
17 }
```

```
thales$ gcc -Wall -std=c99 scanf.c
thales$ a.out
i= 1 .2 hallo welt
------------------------------------
Anzahl: 3 | i=1, f=0.200000, s=hallo
------------------------------------
------------------------------------
Anzahl: 1 | s=we, i=1
------------------------------------
thales$ a.out
j= 4711 0815
------------------------------------
Anzahl: 0 | i=4, f=0.000000, s=
------------------------------------
------------------------------------
Anzahl: 2 | s=j=, i=8
------------------------------------
thales$ a.out
1234 a
------------------------------------
Anzahl: 0 | i=4, f=0.000000, s=
------------------------------------
------------------------------------
Anzahl: 1 | s=12, i=4
------------------------------------
thales$
```

Die Funktion *gets()* liest eine Zeile von der Standardeingabe (*stdin*) in eine Zeichenfolge ein:

```
char* gets(char* s);
```

Wurde ein Zeilentrenner in der Eingabe gefunden, so wird dieser von *gets()* entfernt – im Gegensatz zu *fgets()*. Außerdem kann *gets()* nicht überprüfen, ob hinter dem Zeiger *s* genügend Speicherplatz vorhanden ist. Ist die Eingabezeile länger als der zur Verfügung stehende Platz, kommt es zu einem *Puffer-Überlauf*.

Die Funktion *fgets()* hingegen liest aus einer beliebigen Dateiverbindung (*file pointer*) und zwar max. soviele Zeichen, wie im Puffer untergebracht werden können:

char* *fgets*(**char*** *s*, **int** *n*, *FILE** *stream*);

Mit *n* wird dabei die Größe des Puffers angegeben, wobei dann maximal $n-1$ Zeichen gelesen werden (es muss ja noch das abschließende Null-Byte in den Puffer geschrieben werden – dazu aber später mehr). Folgendes Beispiel illustriert die Verwendung dieser beiden Funktionen:

Programm 4.4: Eingabe mit *gets()* und *fgets()* (*in.c*)

```
1  #include <stdio.h>
2
3  const int BUFSIZE = 10;
4
5  int main() {
6      char s[BUFSIZE];
7
8      fputs("Geben␣Sie␣eine␣Zeile␣ein:␣", stdout);
9      /* eine Zeile von stdin einlesen */
10     gets(s);
11     printf("ECHO:␣>%s<\n", s);
12
13     /* SICHERERE VARIANTE */
14     fputs("Geben␣Sie␣eine␣Zeile␣ein:␣", stdout);
15     /* eine Zeile von stdin einlesen,
16         max. aber nur BUFSIZE-1 Zeichen */
17     fgets(s, BUFSIZE, stdin);
18     printf("ECHO:␣>%s<\n", s);
19 }
```

```
thales$ gcc in.c
thales$ a.out
Geben Sie eine Zeile ein: Wow, C macht ja richtig Spass!
ECHO: >Wow, C macht ja richtig Spass!<
Geben Sie eine Zeile ein: Naja, ein Versuch war's wert!
ECHO: >Naja, ein<
thales$
```

4.5 Weitere Ein- und Ausgabe-Funktionen

Zum Einlesen und Ausgeben von einzelnen Zeichen gibt es die Bibliotheksfunktionen *getc()*, *fgetc()*, *getchar()*, *ungetc()*, *putc()*, *fputc()*, *putchar()*. Nähere Infos dazu gibt es in den zugehörigen Manualseiten (*man getc*, …).

Kapitel 5

Kontrollstrukturen

5.1 Übersicht

⟨expression-statement⟩ ⟶ ⟨expression⟩ „;“
⟨labeled-statement⟩ ⟶ ⟨label⟩ „:“ ⟨statement⟩
⟨label⟩ ⟶ ⟨named-label⟩
⟶ ⟨case-label⟩
⟶ ⟨default-label⟩
⟨case-label⟩ ⟶ **case** ⟨constant-expression⟩
⟨default-label⟩ ⟶ **default**
⟨iterative-statement⟩ ⟶ ⟨while-statement⟩
⟶ ⟨do-statement⟩
⟶ ⟨for-statement⟩
⟨conditional-statement⟩ ⟶ **if** „(“ ⟨expression⟩ „)“ ⟨statement⟩
⟶ **if** „(“ ⟨expression⟩ „)“ ⟨statement⟩
else ⟨statement⟩
⟨while-statement⟩ ⟶ **while** „(“ ⟨expression⟩ „)“ ⟨statement⟩
⟨do-statement⟩ ⟶ **do** ⟨statement⟩ **while** „(“ ⟨expression⟩ „)“
⟨for-statement⟩ ⟶ **for** „(“ [⟨initial-clause⟩] „;“ [⟨expression⟩] „;“
[⟨expression⟩] „)“ ⟨statement⟩
⟨initial-clause⟩ ⟶ ⟨expression⟩
⟶ ⟨declaration⟩
⟨switch-statement⟩ ⟶ **switch** „(“ ⟨expression⟩ „)“ ⟨statement⟩
⟨break-statement⟩ ⟶ **break** „;“
⟨continue-statement⟩ ⟶ **continue** „;“
⟨return-statement⟩ ⟶ [⟨expression⟩] „;“
⟨goto-statement⟩ ⟶ **goto** ⟨named-label⟩ „;“
⟨null-statement⟩ ⟶ „;“

Anmerkung 1: In C ist das Semikolon „;“ der *Abschluss* einer Anweisung (*statement*) und nicht *Trenner* zwischen zwei Anweisungen (wie in einigen anderen Programmiersprachen).

Anmerkung 2: *Ausdrücke* (*expressions*) sind deswegen als Anweisungen sinnvoll, da sie auch Nebeneffekte haben können, wie zum Beispiel der Aufruf einer Funktion, die Inkrementierung einer Variablen oder eine Zuweisung.

Bsp.: *puts*(”Hallo␣Welt!”); bzw. $x = y$++;

5.2 if-Anweisung

Erst in C99 wurde ein Boolean-Datentyp eingeführt. Zuvor wurde stattdessen **int** verwendet, wobei 0 für *false* stand und alle anderen Werte für *true*. Beginnend mit C99 wird der Datentyp *bool* in <*stdbool.h*> definiert zusammen mit den zugehörigen Konstanten *true* und *false*. Dessen ungeachtet blieb die Semantik erhalten, dass generell ganzzahlige Werte bei Bedingungen akzeptiert und entsprechend der alten Regeln interpretiert werden.

bool	int-Wert
true	1 bzw. ungleich 0
false	0

Eine typische Fehlerquelle:

if ($j = 5$) *tu_etwas*();

Gemeint war aber folgendes:

if ($j == 5$) *tu_etwas*();

Die erste Version ist syntaktisch korrekt. *tu_etwas*() wird aber immer aufgerufen, denn die Bedingung $j = 5$ hat als Nebeneffekt die Wertzuweisung von 5 an die Variable j und der Wert der Bedingung ist der zugewiesene Wert, also 5, d. h. ungleich 0, also *true*.

Die Anweisung im (möglicherweise nicht vorhandenen) *else-Zweig* wird ausgeführt, wenn die Bedingung nicht zutrifft. Das **else** wird jeweils dem „nächsten“ **if** zugeordnet:

Programm 5.1: Geschachtelte **if**-Anweisungen mit **else** (*if.c*)

```
1  #include <stdio.h>
2
3  int main() {
4      int n;
5
6      /* ganze Zahl einlesen */
7      printf("n␣=␣");
8      if (scanf("%d", &n) != 1)
9          return 1;
10
11     if (n >= 0)
12         if (n >= 5)
13             puts("n␣>=␣5");
14         else /* zu wem gehoert dieses else wohl? */
15             puts("else");
16     return 0;
17 }
```

```
thales$ gcc -Wall if.c
thales$ a.out
n = 10
n >= 5
thales$ a.out
n = 4
else
thales$ a.out
n = -1
thales$
```

Im Folgenden ist das obige Programm durch einen Anweisungsblock übersichtlicher und eindeutiger gestaltet:

Programm 5.2: Sauber geklammerte **if**-Anweisungen mit **else** (*if1.c*)

```
1  #include <stdio.h>
2
3  int main() {
4      int n;
5
6      /* ganze Zahl einlesen */
7      printf("n = ");
8      if (scanf("%d", &n) != 1)
9          return 1;
10
11     if (n >= 0) { /* macht die Sache klarer! */
12         if (n >= 5) {
13             puts("n >= 5");
14         } else { /* keine Frage mehr! */
15             puts("else");
16         }
17     }
18     return 0;
19 }
```

Wie in diesem Beispiel sollten generell die bedingt auszuführenden Anweisungsteile in geschweifte Klammern gesetzt werden, auch wenn dies bei nur einer einzelnen Anweisung wegfallen könnte. (Natürlich ist dies nur eine sinnvolle Richtlinie, die in Einzelfällen wie bei der Verwendung von **break** oder **continue** auch verletzt werden kann.)

Programm 5.3: **else**-**if**-Kette (*elseif.c*)

```
1  /*
2      Lese ganze Zahlen von der Standardeingabe und fasse
3      hintereinanderfolgende Zahlen des gleichen Werts in der Ausgabe zusammen
4  */
5  #include <stdio.h>
6  #include <stdbool.h>
7
8  int main() {
9      bool first = true; /* noch nichts eingelesen? */
10     int current; /* aktuell eingelesene Zahl */
```

```
11      int last; /* zuvor eingelesene Zahl (falls !first) */
12      int count; /* Anzahl der aufeinanderfolgenden gleichen Zahlen */
13
14      while (scanf("%d", &current) > 0) {
15          if (first) {
16              first = false;
17              count = 1;
18          } else if (current == last) {
19              ++count;
20          } else {
21              printf("%d␣x␣%d\n", count, last);
22              count = 1;
23          }
24          last = current;
25      }
26      if (!first) {
27          printf("%d␣x␣%d\n", count, last);
28      }
29  }
```

C bietet kein Schlüsselwort für **else**-**if**. Stattdessen beginnt der **else**-Fall sofort mit einem **if**. Bei **else**-**if**-Ketten sollte auf ein zusätzliches Einrücken und das Verwenden geschweifter Klammern verzichtet werden, so dass die Kette als solche gut erkennbar bleibt.

5.3 while-Schleife

Die **while**-Schleife führt die Anweisung bzw. den Anweisungsblock solange aus, wie die Bedingung *true* ist, d.h. der Wert des Bedingungsausdrucks ungleich 0 ist. Die Überprüfung der Bedingung findet jeweils *vor* einem Schleifendurchlauf statt. (Es ist also auch möglich, dass der Schleifenrumpf überhaupt nicht durchlaufen wird.)

Beispiel: Zeichenweises Lesen von *stdio* und Zählen der Leerzeichen (Die Bibliotheks-Funktion *getchar()* aus <*stdio.h*> liest ein Zeichen von *stdin* und liefert es als **int**-Wert. Wenn dieser Wert nicht-negativ ist, war die Einlese-Operation erfolgreich. Wird hingegen *EOF* zurückgeliefert (in <*stdio.h*> als -1 definiert), konnte kein Zeichen mehr eingelesen werden.

Programm 5.4: **while**: Zählen von Leerzeichen (*getchar.c*)

```
1   #include <stdio.h>
2
3   int main() {
4       int ch, anzahl = 0;
5
6       while ((ch = getchar()) != EOF) {
7           if (ch == '␣') {
8               anzahl++;
9           }
10      }
11
12      printf("Anzahl␣der␣Leerzeichen:␣%d\n", anzahl);
13  }
```

Hinweise zu diesem Programm:

- Zu beachten ist hier, dass die Variable *ch* hier als **int** deklariert wird und nicht etwa als **char**, da nur der Wertebereich des Datentyps **int** ausreicht, um alle möglichen Zeichenwerte und *EOF* voneinander unterscheiden zu können. In C wird **char** als kleiner ganzzahliger Datentyp betrachtet und entsprechend sind **char** und **int** miteinander kompatibel. Wenn hier **char** verwendet wird, ist nicht einmal definiert, in welcher Form es schief gehen kann, da bei **char** nicht feststeht, ob es mit oder ohne Vorzeichen kommt. Kommt es ohne Vorzeichen, erhalten wir hier eine Dauerschleife. Andernfalls könnte es sein, dass etwa das Zeichen '\0377' mit *EOF* zusammenfällt.
- Der Inkrement-Operator ++ erhöht die Variable um 1. (Dazu später noch mehr.)

5.4 do-while-Schleife

Die **do-while**-Schleife führt die Anweisung bzw. den Anweisungsblock solange aus, wie die Bedingung *true* ist, d.h. der Wert des Bedingungsausdrucks ungleich Null ist. Die Überprüfung der Bedingung findet jeweils *nach* einem Schleifendurchlauf statt. (Es gibt also immer mindestens einen Schleifendurchlauf.)

Programm 5.5: **do-while**: Zählen von Leerzeichen bis zum Zeilenende (*getchar1.c*)

```
1  #include <stdio.h>
2
3  int main() {
4      int ch, anzahl = 0;
5
6      do {
7          if ((ch = getchar()) == ' ') {
8              anzahl++;
9          }
10     } while (ch != '\n' && ch != EOF);
11
12     printf("Anzahl der Leerzeichen in der ersten Zeile: %d\n", anzahl);
13 }
```

Programm 5.6: **do-while**: Überzählige Leerzeichen herausfiltern (*ignore.c*)

```
1  #include <stdio.h>
2  #include <ctype.h> /* wg. isspace() */
3
4  void skip_spaces() {
5      int ch;
6
7      do {
8          ch = getchar();
9      } while (ch != EOF && isspace(ch));
10
11     if (ch != EOF) {
12         /* wir haben ein Zeichen zu weit gelesen
13             => wieder zurueck in die Eingabe damit
14             VORSICHT: nur fuer ein Zeichen garantiert! */
15         ungetc(ch, stdin);
```

```
16      }
17  }
18
19  int main() {
20      int ch;
21
22      while ((ch = getchar()) != EOF) {
23          putchar(ch);
24          if (ch == '␣') {
25              skip_spaces();
26          }
27      }
28  }
```

Anmerkungen:

- Die Angabe von **void** als Datentyp für den Rückgabewert einer Funktion bedeutet, dass diese Funktion keine Werte zurückgibt.
- Die Funktion *ungetc*() stellt ein Zeichen zurück in den Eingabepuffer. Allerdings können nicht beliebig viele Zeichen auf diese Weise zurückgegeben werden. Garantiert ist dies für nur jeweils ein Zeichen – bis dieses gelesen ist (dann für das nächste usw.).

5.5 for-Schleife

Die **for**-Anweisung hat folgende Grundstruktur:

```
1   for (/* Initialisierung */; /* Bedingung */; /* Inkrementierung */)
2       /* Anweisung */
```

Ein typisches Beispiel für die Verwendung einer **for**-Schleife ist das „Hochzählen“ einer Zähl-Variable in einem bestimmten Bereich:

```
int i;
for (i = 1; i <= 10; i++) {
    printf("%d\n", i);
}
```

Alternativ zu einer **for**-Schleife kann auch die äquivalente **while**-Schleife verwenden werden:

```
/* Initialisierung */
while (/* Bedingung */) {
    /* Anweisung */
    /* Inkrement */
}
```

Bei obigem Beispiel sieht das dann wie folgt aus:

```
int i;
i = 1;
while (i <= 10) {
    printf("%d\n", i);
    i++;
}
```

Beginnend mit C99 ist es auch zulässig, die Schleifenvariable innerhalb des Initialisierungsteils zu deklarieren:

```
for (int i = 1; i <= 10; i++) {
    printf("%d\n", i);
}
```

In diesem Falle ist die Schleifenvariable *i* nur innerhalb der Schleife sichtbar. Dies ist vorzuziehen, da dies die Lesbarkeit und Wartbarkeit des Programmtexts erhöht. Sonst tendieren Schleifenvariablen dazu, die Liste der lokalen Variablen unübersichtlich zu machen und Konflikte können nicht sicher ausgeschlossen werden.

Jeder der drei Ausdrücke in einer **for**-Schleife kann auch *leer* sein; eine „leere" Bedingung stellt eine stets erfüllte Bedingung dar. Somit wird die Schleife in diesem Fall zur *Endlosschleife*, die nur mit **return** oder **break** wieder verlassen werden kann!

Endlosschleife:

```
while (1) { /* ... */ };
```

oder

```
for(;;) { /* ... */ };
```

5.6 continue-Anweisung

Die **continue**-Anweisung dient dazu, vorzeitig den nächsten Schleifendurchlauf zu starten, d.h. die restlichen Anweisungen des Anweisungsteils werden übersprungen, so dass bei der **for**-Schleife noch die inkrementierende Anweisung durchgeführt wird und danach der nächste Schleifentest erfolgt.

Programm 5.7: Verwendung von **continue** (*continue.c*)

```
1  #include <stdio.h>
2
3  int main() {
4      for (int i = 1; i <= 20; i++) {
5          if (i % 4 == 0) continue;
6          printf("%d\n", i);
7      }
8  }
```

Programm 5.8: Zusammenhang zwischen **for**, **while** und **continue** (*continue1.c*)

```
1  #include <stdio.h>
2
3  int main() {
4      /* Folgendes Beispiel zeigt, dass die Umformulierung einer
5          for– in eine while–Schleife bzgl. continue nicht ganz identisch ist!
6      */
7      int i = 1;
8      while (i <= 20) {
9          // FALSCH: Endlosschleife mit i=4,4,4,...
10         if (i % 4 == 0) continue;
11         printf("%d\n", i);
12         i++;
13     }
14 }
```

5.7 break-Anweisung

Die **break**-Anweisung dient zum vorzeitigen Verlassen der innersten Schleife oder **switch**-Anweisung (analog zu Java).

Programm 5.9: Verwendung von **break** (*break.c*)

```
1  #include <stdio.h>
2
3  int ggt(int x, int y) {
4      while (y != 0) {
5          int tmp = x % y;
6          x = y; y = tmp;
7      }
8      return x;
9  }
10
11 int main() {
12     for(;;) {
13         printf("Geben Sie zwei positive ganze Zahlen ein: ");
14         int x, y;
15         if (scanf("%d %d", &x, &y) != 2) break;
16         printf("ggT(%d, %d) = %d\n", x, y, ggt(x, y));
17     }
18 }
```

5.8 switch-Anweisung

Die **switch**-Anweisung kann zu einer Fallunterscheidung verwendet werden. Programm 5.10 zeigt eine erste Verwendung der *switch*-Anweisung. Programm 5.11 ist eine Anwendung, bei der verschiedene Fälle gemeinsam behandelt werden.

Programm 5.10: Beispiel für die **switch**-Anweisung (*switch.c*)

```
1  #include <stdio.h>
2
3  int main() {
4      int i;
5
6      printf("Geben Sie eine ganze Zahl:\n");
7      while (scanf("%d", &i) > 0) {
8          switch (i) {
9              case 0:
10                 printf("0 eingegeben\n");
11                 break; /* springt ans Ende von switch */
12             case 1:
13                 printf("1 eingegeben\n");
14                 break; /* dito */
15             default: /* fuer alle anderen Faelle, also nicht 0 oder 1 */
16                 printf("Weder 0 noch 1\n");
17         } /* Ende von switch */
```

```
18            printf("Noch␣eine␣Zahl?\n");
19        }
20    }
```

Zur Semantik und Verwendung:

- Nach der Auswertung des **switch**-Ausdrucks wird bei der Anweisung des passenden **case** fortgefahren.
- Auch die folgenden „Fälle“ werden abgearbeitet, falls dies nicht durch ein explizites **break** verhindert wird.
- Der **switch**-Ausdruck muss einen ganzzahligen Typ oder einen Aufzählungstyp (**enum**) haben.
- Der Ausdruck bei einem **case** muß ein *konstanter* Ausdruck sein. Zulässig sind ganzzahlige Werte, Zeichenkonstanten oder Konstanten eines Aufzählungstyps (**enum**).
- Die bei den einzelnen Fällen angegebenen Konstanten müssen zueinander disjunkt sein.
- Trifft keiner der Fälle zu, geht es bei **default** weiter, falls dieser existiert. Ansonsten wird die gesamte **switch**-Anweisung übersprungen.
- Aus Gründen der Lesbarkeit empfiehlt es sich, den **default**-Fall immer explizit (am Ende der **switch**-Anweisung) mit aufzuführen. Und auch dieser Fall sollte sicherheitshalber mit einem **break** verlassen werden.

Programm 5.11: Beispiel für die **switch**-Anweisung, bei der mehrere Fälle gemeinsam behandelt werden (*switch1.c*)

```
1   #include <stdio.h>
2   #include <stdbool.h>
3
4   bool ispunc(char arg) {
5       switch (arg) {
6           case '.': /* gemeinsamer Fall ... */
7           case ',': /* ... da break fehlt! */
8           case ':':
9           case ';':
10          case '!':
11              return true; /* fuer all die obigen Faelle! */
12          default:
13              return false;
14      }
15  }
16
17  int main() {
18      char ch;
19
20      printf("Geben␣Sie␣ein␣Zeichen␣ein:␣");
21
22      if (scanf("%c", &ch) > 0) {
23          if (ispunc(ch)) {
24              puts("Interpunktion");
```

```
25              } else {
26                  puts("keine␣Interpunktion");
27              }
28          } else {
29              puts("\nNichts␣eingegeben!");
30          }
31  }
```

Kapitel 6

Ausdrücke

Ein *Ausdruck* besteht aus *Operatoren* und *Operanden*, wie zum Beispiel zu einer Addition der Operator + und die Summanden gehören.

6.1 Operanden

6.1.1 Links- und Rechts-Werte

⟨expression⟩ ⟶ ⟨comma-expression⟩
⟨comma-expression⟩ ⟶ ⟨assignment-expression⟩
⟶ ⟨comma-expression⟩ „,“ ⟨assignment-expression⟩
⟨assignment-expression⟩ ⟶ ⟨conditional-expression⟩
⟶ ⟨unary-expression⟩ ⟨assignment-op⟩ ⟨assignment-expression⟩

Zuweisungen gehören in C zu den Ausdrücken. Wegen der Asymmetrie einer Zuweisung, bei der links etwas Veränderbares stehen muss, während auf der rechten Seite ein beliebiger Ausdruck stehen kann, wird zwischen Links- und Rechts-Werten unterschieden bzw. zwischen *unary-expression* und *assignment-expression*.

In C ist ein Objekt eine Speicherfläche, deren Inhalt ausgelesen und verändert werden kann. Ein *Links-Wert* ist ein Ausdruck, der ein Objekt identifiziert. Ob ein Links-Wert zum Auslesen oder zum Verändern des Objekts dient, hängt vom Kontext ab. Steht ein Links-Wert auf der linken Seite einer Zuweisung, so wird das Objekt entsprechend verändert und der Wert des Zuweisungsausdrucks (*assignment-expression*) ergibt sich aus dem neuen Wert des Objekts.

Beispiele für Links-Werte sind Variablen, dereferenzierte Zeiger oder indizierte Arrays:

```
int i; int a[10]; int* p = &i;
struct int2 { int i, j; } s; struct int2* sp = &s;
i = 1; /* Links−Wert ist eine Variable */
*p = 2; /* Links−Wert ist ein derefenzierter Zeiger */
a[2] = 3; /* Links−Wert ist ein indiziertes Array */
s.i = 4; /* Links−Wert ist ein Feld eines Verbundstyps */
sp−>j = 5; /* Links−Wert ist ein Feld eines Verbundstyps */
```

Allerdings ist dabei zu beachten, dass in einigen Fällen auch syntaktisch zulässige Links-Werte nicht links von einer Zuweisung stehen dürfen wie beispielsweise im Falle von Konstantenvariablen:

```
const int i = 1;
i = 2; /* syntaktisch korrekt, jedoch semantisch nicht zulaessig */
```

Mehrfachzuweisungen sind zulässig:

```
int i, j, k;
i = j = k = 1; /* entspricht i = (j = (k = 1)) */
```

Eine Zuweisung liefert jedoch keinen Links-Wert:

```
int i, j, k;
(i += j) += k; /* ist nicht zulaessig */
```

Rechts-Werte können Links-Werte sein oder beliebige andere Ausdrücke, bei denen es nicht mehr darauf ankommt, dass ein konkretes Objekt damit verbunden sein muss. So ist beispielsweise das Ergebnis einer Addition nur noch ein Rechts-Wert.

6.1.2 Operanden im Einzelnen

⟨unary-expression⟩ ⟶ ⟨postfix-expression⟩
⟶ ⟨sizeof-expression⟩
⟶ ⟨unary-minus-expression⟩
⟶ ⟨unary-plus-expression⟩
⟶ ⟨logical-negation-expression⟩
⟶ ⟨bitwise-negation-expression⟩
⟶ ⟨address-expression⟩
⟶ ⟨indirection-expression⟩
⟶ ⟨preincrement-expression⟩
⟶ ⟨predecrement-expression⟩
⟨postfix-expression⟩ ⟶ ⟨primary-expression⟩
⟶ ⟨subscript-expression⟩
⟶ ⟨component-selection-expression⟩
⟶ ⟨function-call⟩
⟶ ⟨postincrement-expression⟩
⟶ ⟨postdecrement-expression⟩
⟶ ⟨compound-literal⟩
⟨primary-expression⟩ ⟶ ⟨identifier⟩
⟶ ⟨constant⟩
⟶ ⟨parenthesized-expression⟩
⟨subscript-expression⟩ ⟶ ⟨postfix-expression⟩ „[“ ⟨expression⟩ „]“
⟨component-selection-expression⟩ ⟶ ⟨direct-component-selection⟩
⟶ ⟨indirect-component-selection⟩
⟨direct-component-selection⟩ ⟶ ⟨postfix-expression⟩ „.“ ⟨identifier⟩
⟨indirect-component-selection⟩ ⟶ ⟨postfix-expression⟩ „->“ ⟨identifier⟩
⟨function-call⟩ ⟶ ⟨postfix-expression⟩
„(“ [⟨expression-list⟩] „)“
⟨compound-literal⟩ ⟶ „(“ ⟨type-name⟩ „)“
„{“ ⟨initializer-list⟩ [„,“] „}“
⟨constant⟩ ⟶ ⟨integer-constant⟩
⟶ ⟨floating-constant⟩
⟶ ⟨character-constant⟩
⟶ ⟨string-constant⟩

Namen (*identifier*) können im Rahmen eines *primary-expression* sich auf eine Variablenvereinbarung beziehen, eine Funktion oder einen der Werte eines Aufzählungstyps:

- Variablennamen sind in der Regel zulässige Links-Werte. Es gibt jedoch eine wichtige Ausnahme: Der Name einer Array-Variablen steht für die Adresse des ersten Feldes. Da die Adresse konstant ist, kann ihr auch nichts zugewiesen werden. Folgende Konstruktion ist also nicht zulässig:

```
int a[10], b[10];
a = b; /* FALSCH: a ist kein Links−Wert */
```

- Interessanterweise geht dies aber bei Verbundtypen, bei denen der Variablenname jeweils das vollständige Objekt repräsentiert:

```
struct int2 { int i, j; } a, b;
a = (struct int2){1, 2}; /* Aggregrate zulaessig ab C99 */
a = b; /* zulaessig */
```

  (Dies entspricht der *deep-copy*-Semantik in Modula-2 oder Oberon. Es weicht allerdings von der Semantik des *shallow-copy* in Java ab.)

- Konsequenterweise bedeutet dies, dass auch Arrays einander zugewiesen werden können, wenn diese in Verbundtypen eingepackt werden:

```
struct int10 { int i[10]; } a, b;
a = b; /* zulaessig */
```

- Funktionsnamen *ohne* Parameterliste, d.h. auch ohne Klammern, werden als konstante Zeiger auf die Funktion interpretiert:

```
int (*writestring)(const char* s); /* Funktionszeiger */
writestring = puts; /* Zeiger auf die Funktion kopieren ... */
(*writestring)("Hallo zusammen!"); /* ... und aufrufen */
```

  (Funktionszeiger können in C dazu dienen, OO-Techniken rudimentär nachzubilden.)

Der Typ einer Konstanten (*constant*) ergibt sich direkt aus der lexikalischen Analyse:

```
int i = 1; /* integer−constant: Datentyp int */
double d = 1.23e−45 /* floating−constant: Datentyp double */
char c = 'a'; /* character−constant: Datentyp char */
char* s = "hello world"; /* string−constant: Datentyp char* */
```

Bei Bedarf kann der Datentyp auch feiner variiert werden. Auch gibt es Alternativen bei der Darstellung:

```
unsigned long ul = 4294967295UL; /* Datentyp unsigned long: 2^32 − 1 */
long long ll = 1099511627776LL; /* Datentyp long long */
int hex = 0x1ffee; /* Datentyp int in hexadezimaler Notation */
float f = 1.23e−45F; /* Datentyp float */
char newline = '\n'; /* Datentyp char: benanntes Sonderzeichen */
char bell = '\007'; /* Datentyp char in oktaler Darstellung */
char* s = "Hallo zusammen!\007\n";
```

Zeichenketten-Konstanten werden immer implizit durch ein Null-Byte abgeschlossen. Zu beachten ist, dass Zeichenketten-Konstanten nicht die gesamte Zeichenkette repräsentieren, sondern nur einen konstanten Zeiger auf das erste Zeichen. Eine Initialisierung eines Zeichen-Arrays mit einer Zeichenketten-Konstanten ist jedoch zulässig. Dies erlaubt auch eine automatisierte Festlegung der Array-Größe:

```
char greeting[] = "Hallo zusammen!\n";
```

Bei einem Funktionsaufruf ist die Reihenfolge, in der die aktuellen Parameter bewertet werden, nicht definiert. Jedoch darf davon ausgegangen werden, dass sich die Auswertungen der einzelnen Parameter nicht vermischen, d.h. die Auswertung beginnt mit einem der Parameter, diese wird vollständig ausgeführt, dann wird ein weiterer Parameter ausgewählt, dieser ebenfalls vollständig bewertet usw. bis die Auswertung aller Parameter abgeschlossen ist.

6.2 Operatoren

6.2.1 Übersicht

Die folgende Tabelle (die weitgehend [Harbison 2002] entnommen wurde) gibt einen Überblick der Operatoren, der Vorränge und der Bindungsreihenfolgen (Assoziativität):

Symbole	Art	Typ	Vorrang	Assoziativität
Namen und Konstanten	einfache Symbole	primär	16	—
a[*i*]	Indizierung	Postfix	16	von links
f()	Funktionsaufruf	Postfix	16	von links
.	Feldauswahl	Postfix	16	von links
−>	indirekte Feldauswahl	Postfix	16	von links
++ −−	Inkrement, Dekrement	Postfix	16	von links
(*Typ*) {*Init*}	Aggregrat	Postfix	16	von links
sizeof	Speichergröße	Präfix	15	von rechts
~	bitweise Negation	Präfix	15	von rechts
!	logische Negation	Präfix	15	von rechts
− +	arithmetische Vorzeichen	Präfix	15	von rechts
&	Adress-Operator	Präfix	15	von rechts
*	Dereferenzierung	Präfix	15	von rechts
(*Typ*)	Typ-Konvertierung	Präfix	14	von rechts
* / %	multiplikative Operatoren	dyadisch	13	von links
+ −	additive Operatoren	dyadisch	12	von links
<< >>	Bit-Verschiebungen	dyadisch	11	von links
< > <= >=	Vergleiche	dyadisch	10	von links
== !=	Äquivalenz	dyadisch	9	von links
&	bitweises Und	dyadisch	8	von links
^	bitweises Exklusiv-Oder	dyadisch	7	von links
|	bitweises Inklusiv-Oder	dyadisch	6	von links
&&	logisches Und	dyadisch	5	von links
||	logisches Oder	dyadisch	4	von links
? :	Auswahl	triadisch	3	von rechts
= += −= *= /= %= <<= >>= &= ^= |=	Zuweisung	dyadisch	2	von rechts
,	sequentiell	dyadisch	1	von links

6.2.2 Monadische Postfix-Operatoren

Monadische Operatoren, auch unäre Operatoren genannt, stehen entweder vor oder hinter dem Operanden. Letztere werden Postfix-Operatoren genannt. Prominentes Beispiel hierfür sind die Postfix-Inkrement- und Dekrement-Operatoren, die als Operanden einen Links-Wert erwarten, diesen um eins erhöhen oder verringern. Sie liefern jedoch als Ausdrucks-Wert den alten Links-Wert zurück:

```
int i = 0; int j;
j = i++; /* i = 1, j = 0 */
```

6.2.3 Monadische Präfix-Operatoren

Die Mehrheit der monadischen Operatoren geht dem zugehörigen Operanden voran:

⟨sizeof-expression⟩ ⟶ **sizeof** „(“ ⟨type-name⟩ „)“
⟶ **sizeof** ⟨unary-expression⟩
⟨unary-minus-expression⟩ ⟶ „-“ ⟨cast-expression⟩
⟨unary-plus-expression⟩ ⟶ „+“ ⟨cast-expression⟩
⟨logical-negation-expression⟩ ⟶ „!“ ⟨cast-expression⟩
⟨bitwise-negation-expression⟩ ⟶ „~“ ⟨cast-expression⟩
⟨address-expression⟩ ⟶ „&“ ⟨cast-expression⟩
⟨indirection-expression⟩ ⟶ „*“ ⟨cast-expression⟩
⟨preincrement-expression⟩ ⟶ „++“ ⟨unary-expression⟩
⟨predecrement-expression⟩ ⟶ „–“ ⟨unary-expression⟩
⟨cast-expression⟩ ⟶ ⟨unary-expression⟩
⟶ „(“ ⟨type-name⟩ „)“ ⟨unary-expression⟩

Wie sich der Grammatik entnehmen lässt, gibt es die Operatoren ++ und −− auch als Präfix-Operatoren. Im Unterschied zu den gleichnamigen Postfix-Operatoren liefern sie als Ausdruck den Links-Wert nach der Inkrementierung bzw. Dekrementierung zurück:

```
int i = 0; int j;
j = ++i; /* i = 1, j = 1 */
```

Anders als in anderen Programmiersprachen mit expliziten Zeigern ist der Dereferenzierungs-Operator * in C ein Präfix-Operator. Das hat zur Konsequenz, dass Klammern unvermeidlich sein können, wenn Post- und Präfix-Operatoren gemischt verwendet werden:

```
struct int2 { int i, j; } a; struct int2* p;
p = &a;
*p.i = 1; /* FALSCH: p.i ist kein Zeiger */
(*p).i = 1; /* OK: Zuerst den Zeiger dereferenzieren, dann selektieren */
```

Da (∗.*p*).*i* so umständlich aussieht, gibt es den Operator −>, der es erlaubt, auf die Klammern zu verzichten: *p*−>*i*. Dem Operator −> muss immer ein Feldname folgen.

Der Adressoperator & liefert die Adresse eines Links-Werts. Der Datentyp ist dann jeweils der zugehörige Zeigertyp. Aus **int** wird so **int***. Aus **int*** würde, falls der Operator ein weiteres Mal zum Einsatz kommt, **int****. Beispiele:

```
int i; int* ip; int** ipp;
ipp = &ip; *ipp = &i; /* ip = &i */
**ipp = 2; /* i = 2 */
```

Der Name eines Feldes steht bereits für die Anfangsadresse des Felds:

```
int a[10]; int* ip;
ip = &a; /* FALSCH: a ist bereits eine konstante Adresse */
ip = a; /* OK: ip zeigt jetzt auf a[0] */
ip = &a[0]; /* OK: umstaendlichere Alternative */
ip = &a[3]; /* OK: ip zeigt auf a[3] */
```

Das logische Komplement liefert *true*, falls der Operand 0 oder *false* ist, ansonsten *false*. Wie bei allen anderen logischen Operatoren werden beliebige skalare Operanden akzeptiert und das Resultat ist sowohl zu den ganzzahligen Typen einschließlich *bool* kompatibel.

Der **sizeof**-Operator liefert die benötigte Speicherfläche für den angegebenen Typ. Der Typ wird entweder explizit angegeben (und muss dann in Klammern stehen) oder implizit durch den Typ des folgenden Ausdrucks. Im letzteren Fall wird der Ausdruck nicht bewertet, sondern nur dessen Typ vom Übersetzer bestimmt.

Üblicherweise wird die benötigte Speicherfläche in Bytes gemessen, aber das muss nicht überall der Fall sein. Zumindest gab es (mittlerweile historische) Architekturen mit so „exotischen“ Konstellationen wie 5-Bit-Bytes und 36-Bit-Wörtern, für die auch C-Übersetzer entwickelt wurden, so dass C nicht festlegt, in welchen Einheiten Speicher gemessen wird. Deswegen sollten auch nicht die üblichen ganzzahligen Datentypen wie etwa **int** oder **unsigned long** dafür verwendet werden, sondern *size_t* aus <*stddef.h*>.

Folgendes Beispiel illustriert die Verwendung einiger unärer Operatoren:

Programm 6.1: Verwendung unärer Operatoren (*unaer.c*)

```
1  #include <stdio.h>
2  #include <stddef.h>
3
4  int main() {
5      int i = 3; /* vorinitialisierte Integervar. */
6      int* p; /* Zeiger auf eine Integervar. */
7      int a[50]; /* Integer-Array */
8      size_t sizeof_a; /* Groesse des Arrays */
9      size_t sizeof_p; /* Groesse des Zeigers p */
10
11     p = &i; /* Adresse von i wird p zugewiesen */
12     printf("i=%d, p=%p (Adresse), *p=%d (Wert)\n", i, p, *p);
13
14     printf("i++=%d, ", i++); /* nachher inkrementieren */
15     printf("i=%d\n", i);
16
17     printf("++i=%d, ", ++i); /* vorher inkrementieren */
18     printf("i=%d\n", i);
19
20     printf("!0=%d, !1=%d, !2=%d\n", !0, !1, !2); /* log. Negation */
21
22     printf("i=%08x, ~i=%08x\n", i, ~i); /* bitweises (Einer-)Komplement */
23
24     p = a; sizeof_a = sizeof(a); sizeof_p = sizeof(p);
25     /* Format %zd: size_t dezimal */
26     printf("sizeof(a)=%zd, sizeof(p)=%zd\n", sizeof_a, sizeof_p);
27 }
```

```
doolin$ gcc -Wall -std=c99 unaer.c
doolin$ a.out
i=3, p=ffbff65c (Adresse), *p=3 (Wert)
i++=3, i=4
++i=5, i=5
!0=1, !1=0, !2=0
i=00000005, ~i=fffffffa
sizeof(a)=200, sizeof(p)=4
doolin$
```

Hinweis: Bei älteren C-Bibliotheken, die noch nicht den C99-Standard vollständig unterstützen, kann es sein, dass *%zd* noch nicht funktioniert. Dann sollte notfalls *%lu* mit einer zugehörigen Typ-Konvertierung verwendet werden. Beispiel:

printf(ßizeof(**int**) = *%lu\n"*,␣(unsigned␣long)␣sizeof(int));

6.2.4 Dyadische Operatoren

Dyadische Operatoren, auch binäre Operatoren genannt, werden in C durchweg in der Infix-Notation verwendet:

⟨logical-or-expression⟩ ⟶ ⟨logical-and-expression⟩
⟶ ⟨logical-or-expression⟩
„||“ ⟨logical-and-expression⟩
⟨logical-and-expression⟩ ⟶ ⟨bitwise-or-expression⟩
⟶ ⟨logical-and-expression⟩
„&&“ ⟨bitwise-or-expression⟩
⟨bitwise-or-expression⟩ ⟶ ⟨bitwise-xor-expression⟩
⟶ ⟨bitwise-or-expression⟩
„|“ ⟨bitwise-xor-expression⟩
⟨bitwise-xor-expression⟩ ⟶ ⟨bitwise-and-expression⟩
⟶ ⟨bitwise-xor-expression⟩
„|“ ⟨bitwise-and-expression⟩
⟨bitwise-and-expression⟩ ⟶ ⟨equality-expression⟩
⟶ ⟨bitwise-and-expression⟩
„^“ ⟨equality-expression⟩
⟨equality-expression⟩ ⟶ ⟨relational-expression⟩
⟶ ⟨equality-expression⟩
⟨equality-op⟩ ⟨relational-expression⟩
⟨equality-op⟩ ⟶ „==“ | „!=“
⟨relational-expression⟩ ⟶ ⟨shift-expression⟩
⟶ ⟨relational-expression⟩
⟨relational-op⟩ ⟨shift-expression⟩
⟨relational-op⟩ ⟶ „<“ | „<=“ | „>“ | „>=“
⟨shift-expression⟩ ⟶ ⟨additive-expression⟩
⟶ ⟨shift-expression⟩
⟨shift-op⟩ ⟨additive-expression⟩
⟨shift-op⟩ ⟶ „«“ | „»“
⟨additive-expression⟩ ⟶ ⟨multiplicative-expression⟩
⟶ ⟨additive-expression⟩
⟨add-op⟩ ⟨multiplicative-expression⟩
⟨add-op⟩ ⟶ „+“ | „-“
⟨multiplicative-expression⟩ ⟶ ⟨cast-expression⟩
⟶ ⟨multiplicative-expression⟩
⟨mult-op⟩ ⟨cast-expression⟩
⟨mult-op⟩ ⟶ „*“ | „/“ | „%“

Seit *C89* wird zugesichert, dass Ausdrücke in der genannten Reihenfolge ausgewertet werden. Das bedeutet, dass die scheinbare Kommutativität einiger Operatoren wie + oder ∗ durch den Übersetzer nicht für Optimierungszwecke ausgenutzt werden darf, wenn die Reihenfolge in Bezug auf Überläufe oder die Genauigkeit (bei Gleitkommazahlen!) relevant sein könnte. Eine Änderung der Reihenfolge ist somit für den Übersetzer nur dann

zulässig, wenn dies auf keinen Fall das Resultat ändert. Das bedeutet jedoch nicht, dass Seiteneffekte in einer garantierten Reihenfolge erfolgen. Diese bleiben undefiniert. Beispiel:

```
int i = 1, j;
j = (i += 10) % ++i;
/* Welchen Wert hat j? 0 oder 11? */
```

Bei den logischen Operatoren wird zunächst der linke Operand ausgewertet und dann nur für den Fall, dass daraus noch nicht das Resultat folgt, der zweite Operand bewertet. Die einzelnen Operanden müssen jeweils einen beliebigen skalaren Typ haben, der mit 0 vergleichbar ist. Das Resultat ist 0 (*false*) oder 1 (*true*) entsprechend dem Wertebereich von *bool*.

Die Operanden eines bitweisen Operators müssen ganzzahlig sein und werden zunächst aneinander angeglichen. Danach werden die Operanden als (inzwischen gleich lange) Bitfelder betrachtet, bei denen die Operation dann paarweise für alle Bits durchgeführt wird (das erste Bit des ersten Operanden mit dem ersten Bit des zweiten Operanden ergibt das erste resultierende Bit usw.).

Die Schiebe-Operatoren << und >> multiplizieren bzw. dividieren ganzzahlige Werte mit Zweier-Potenzen, d.h. $i << n$ entspricht $i * 2^n$ und $i >> n$ entspricht der ganzzahligen Division $i/2^n$. Die Schiebe-Operationen sind nicht definiert, wenn n negativ oder größer oder gleich der Zahl der zur Verfügung stehenden Bits ist. Ferner ist es der Implementierung überlassen, wie $i >> n$ umgesetzt wird, wenn i negativ ist.

Die bitweisen Operatoren können in Kombination mit den Schiebe-Operationen dazu genutzt werden, um Mengen als Bitfelder darzustellen, indem die einzelnen Elemente i durch die Werte 2^i dargestellt werden:

Mengen-notation	in C
$A \cup B$	$A \mid B$
$A \cap B$	$A \,\&\, B$
$A \setminus B$	$A \,\&\, \sim B$
$\{i\}$	$1 << i$
$i \in A$	$(1 << i) \,\&\, A$

Beim Divisions-Operator / werden zunächst die Operanden aneinander angeglichen und dann entsprechend des größten gemeinsamen Typs durchgeführt. D.h. bei ganzzahligen Operanden wird eine ganzzahlige Division durchgeführt (mit einer Rundung in Richtung zur 0, falls das Resultat nicht exakt ist) und bei Fließkommazahlen bzw. gemischten Operanden kommt es zu einer Fließkomma-Division.

Folgendes Beispiel illustriert die Verwendung binärer Operatoren:

Programm 6.2: Verwendung binärer Operationen (*binaer.c*)

```
1  #include <stdio.h>
2
3  /* Ausgabe einer nicht−negativen ganzen Zahl in Binaerdarstellung;
4     die Zahl der ausgegebenen Zeichen wird zurueckgegeben
5  */
6  unsigned int print_binary(unsigned int n, unsigned int minwidth) {
7     unsigned int rn; /* n in bit−maessig umgedrehter Reihenfolge */
8     unsigned int printed = 0; /* Anzahl der ausgegebenen Zeichen */
9     unsigned int count = 0; /* Anzahl der relevanten Ziffern */
10
11    /* Reihenfolge der Bits umdrehen */
```

```
12      for (rn = 0; n != 0; n >>= 1) {
13          rn = (rn << 1) | (n & 1);
14          ++count;
15      }
16      /* Ausgabe der fuehrenden Nullen, falls notwendig */
17      if (minwidth > count) {
18          minwidth -= count;
19          while (minwidth > 0) {
20              putchar('0'); ++printed; --minwidth;
21          }
22      }
23      /* Ausgabe der relevanten Ziffern */
24      for (; count > 0; rn >>= 1) {
25          if (rn & 1) {
26              putchar('1');
27          } else {
28              putchar('0');
29          }
30          ++printed; --count;
31      }
32      return printed;
33  }
34
35  int main() {
36      unsigned int i = 17, j = 3;
37
38      printf("      i = ");  print_binary(i, 5); puts("");
39      printf("      j = ");  print_binary(j, 5); puts("");
40
41      /* Bitweises UND, ODER und XOR (= exklusives ODER) */
42      printf("    i&j = ");  print_binary(i & j, 5);
43      printf(",  i|j = ");  print_binary(i | j, 5);
44      printf(",  i^j = ");  print_binary(i ^ j, 5);
45      puts("");
46
47      /* Bit-Verschiebungen (nicht zyklisch!) */
48      printf("  i>>1 = ");  print_binary(i >> 1, 5);
49      printf(", i<<1 = ");  print_binary(i << 1, 5);
50      puts("");
51
52      /* Bit-Verschiebungen bei negativen Zahlen (undefiniert!)
53          (bei der Konvertierung von signed zu unsigned garantiert
54          C hier das Zweier-Komplement bei negativen Zahlen)
55      */
56      printf("    -1 = ");
57      unsigned int bits = print_binary(-1, 0); puts("");
58      printf(" -1>>1 = ");  print_binary(-1 >> 1, bits); puts("");
59      printf(" -1<<1 = ");  print_binary(-1 << 1, bits); puts("");
60  }
```

```
dublin$ gcc -Wall -std=c99 binaer.c
dublin$ a.out
     i = 10001
     j = 00011
   i&j = 00001,  i|j = 10011,  i^j = 10010
  i>>1 = 01000, i<<1 = 100010
    -1 = 11111111111111111111111111111111
 -1>>1 = 11111111111111111111111111111111
 -1<<1 = 11111111111111111111111111111110
dublin$
```

Anmerkung: Wenn der Schiebe-Operator >> für negative ganze Zahlen verwendet wird, bleibt es der jeweiligen Implementierung überlassen, ob Nullen oder Einsen an der höchstwertigen Stelle „nachrücken". Wenn (wie fast durchgängig üblich) das Zweier-Komplement zur Darstellung negativer Zahlen verwendet wird, bedeutet dies, dass der Wert des Ausdrucks $-1 >> 1$ entweder -1 oder 2^{n-1} ergeben kann, wobei n für die Zahl der verwendeten Bits steht.

Der Modulo-Operator % liefert bei ganzzahligen Operanden den Rest der entsprechenden ganzzahligen Division, so dass $(a/b)*b + a\%b == a$ gilt, falls a/b repräsentierbar ist. Ferner gilt $0 \leq (a\%b) < b$ für $a >= 0$ und $b > 0$. Abweichende Definitionen des Modulo-Operators existieren jedoch, falls a oder b negativ werden:

- **T-Definition:** a/b == (**int**)((**double**)a / (**double**)b)
  d.h. es wird als Resultat der ganzzahligen Division die größte ganzzahlige Zahl genommen, deren Betrag kleiner oder gleich als $\frac{a}{b}$ ist und deren Differenz weniger als 1 beträgt (Rundung in Richtung zur 0).

- **F-Definition:** a/b == (**int**)*floor*((**double**)a / (**double**)b)
  d.h. es wird die größtmögliche ganze Zahl genommen, die kleiner oder gleich $\frac{a}{b}$ ist. (Rundung in Richtung $-\infty$). Hier gilt:
  0 <= $a\%b$ && $a\%b < b$ || $b < a\%b$ && $a\%b$ <= 0

- **Nach Euklid:** Diese stimmt mit der F-Definition überein, falls y positiv ist. Es gilt jedoch
  $a/(-b) == -(a/b)$&& $a\%(-b) == a\%b$ und daraus folgt:
  0 <= $a\%b$ && $a\%b < abs(b)$

Praktisch alle gängigen Prozessor-Architekturen unterstützen die T-Definition, die aus diesem Grunde auch von vielen Programmiersprachen übernommen wurde wie beispielsweise C, C++, Modula-2 oder Java. (Im Falle von C blieb die konkrete Semantik jedoch vor C99 noch undefiniert.) Es gibt jedoch gute Gründe, die anderen Definitionen zu verwenden. So unterstützten beispielsweise Oberon und Scheme die F-Definition. Folgendes Beispiel demonstriert die Unterschiede an Beispielen:

Programm 6.3: Der Modulo-Operator (*modulo.c*)

```
1  #include <stdio.h>
2
3  /* nach der T-Definition */
4  int tmod(int a, int b) {
5      return a % b;
6  }
7  int tdiv(int a, int b) {
8      return a / b;
```

```
 9  }
10
11  /* nach der F−Definition */
12  int fmod(int a, int b) {
13      int r = a % b;
14      if ((a >= 0) == (b >= 0) || r == 0) {
15          return r;
16      } else {
17          return r+b;
18      }
19  }
20  int fdiv(int a, int b) {
21      int r = a % b;
22      int q = a / b;
23      if ((a >= 0) == (b >= 0) || r == 0) {
24          return q;
25      } else {
26          return q−1;
27      }
28  }
29
30  /* nach Euklid */
31  int emod(int a, int b) {
32      if (b > 0) {
33          return fmod(a, b);
34      } else {
35          return fmod(a, −b);
36      }
37  }
38  int ediv(int a, int b) {
39      if (b > 0) {
40          return fdiv(a, b);
41      } else {
42          return −fdiv(a, −b);
43      }
44  }
45
46  int main() {
47      int a, b;
48      printf("a␣=␣"); scanf("%d", &a);
49      printf("b␣=␣"); scanf("%d", &b);
50      printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣T−Def.␣F−Def.␣Euklid\n");
51      printf("%4d/%4d␣=␣%6d␣%6d␣%6d\n",
52          a, b, tdiv(a, b), fdiv(a, b), ediv(a, b));
53      printf("%4d%%%4d␣=␣%6d␣%6d␣%6d\n",
54          a, b, tmod(a, b), fmod(a, b), emod(a, b));
55  }
```

```
dublin$ gcc -Wall -std=c99 modulo.c
dublin$ a.out
a = 13
b = -4
              T-Def. F-Def. Euklid
  13/  -4 =     -3     -4     -3
  13%  -4 =      1     -3      1
dublin$ a.out
a = -13
b = -4
              T-Def. F-Def. Euklid
 -13/  -4 =      3      3      4
 -13%  -4 =     -1     -1      3
dublin$
```

6.2.5 Auswahl-Operator

⟨conditional-expression⟩ ⟶ ⟨logical-or-expression⟩
⟶ ⟨logical-or-expression⟩ „?“ ⟨expression⟩
„:“ ⟨conditional-expression⟩

Eine *Auswahl* bzw. ein bedingter Ausdruck besteht aus einer Bedingung, der zwei Ausdrücke folgen. Die Bedingung wird auf die bekannte Weise bewertet: Ist der Wert ungleich Null (also *true*), so wird der erste Ausdruck als Ergebnis genommen, ist er gleich Null (also *false*), so der zweite.

Ein Beispiel ist die folgende Auswahl:

$i = a > 7 ? x + y : y - 2;$

Im Prinzip ist der Auswahl-Operator eine Kurzschreibweise für eine **if**-Anweisung. Man kann obiges Beispiel ausführlicher auch wie folgt umschreiben:

if ($a > 7$) {
    $i = x + y;$
} **else** {
    $i = y - 2;$
}

6.2.6 Komma-Operator

⟨comma-expression⟩ ⟶ ⟨assignment-expression⟩
⟶ ⟨comma-expression⟩ „,“
⟨assignment-expression⟩

Durch die Verwendung des *Komma-Operators* lassen sich mehrere Ausdrücke schreiben, wo eigentlich nur ein Ausdruck erlaubt ist. Alle durch Komma voneinander getrennten

Teilausdrücke werden bewertet. Der Wert des Gesamtausdrucks ist gleich dem Wert des letzten Teilausdrucks; siehe folgendes Beispiel:

Programm 6.4: Verwendung des Komma-Operators (*komma.c*)

```
1  #include <stdio.h>
2
3  int main() {
4      int a, b, c;
5
6      /* Wert eines Komma-Ausdrucks ist gleich
7          dem Wert des LETZTEN Teil-Ausdrucks */
8      c = (a = 1, b = 2);
9      printf("a=%d, b=%d, c=%d\n", a, b, c);
10
11     puts("------------------------");
12
13     /* sinnvollere Verwendung innerhalb einer for-Schleife
14         und zum Sparen von Anweisungsbloecken ({...}) */
15     for (a = 0, b = 0, c = 0; a <= 10; a++) {
16         b -= 1, c += 2, printf("a=%2d, b=%3d, c=%2d\n", a, b, c);
17     }
18 }
```

```
dublin$ gcc -Wall -std=c99 komma.c
dublin$ a.out
a=1, b=2, c=2
------------------------
a= 0, b= -1, c= 2
a= 1, b= -2, c= 4
a= 2, b= -3, c= 6
a= 3, b= -4, c= 8
a= 4, b= -5, c=10
a= 5, b= -6, c=12
a= 6, b= -7, c=14
a= 7, b= -8, c=16
a= 8, b= -9, c=18
a= 9, b=-10, c=20
a=10, b=-11, c=22
dublin$
```

6.2.7 Zuweisungen

⟨assignment-expression⟩ ⟶ ⟨conditional-expression⟩
⟶ ⟨unary-expression⟩ ⟨assignment-op⟩ ⟨assignment-expression⟩

⟨assignment-op⟩ ⟶ „=“ | „*=“ | „/=“ | „%=“ | „+=“ | „-=“ | „«=“ | „»=“ | „&=“ | „^=“ | „|=“

Die Zuweisungs-Operatoren liefern (wie die anderen Operatoren auch) einen Wert und gleichzeitig mit der Zuweisung auch einen Nebeneffekt. Für alle Zuweisungs-Operatoren gilt dabei, dass der zugewiesene Wert gleich dem resultierenden Wert ist.

Die doppelte Funktionalität der Zuweisungs-Operatoren ermöglicht eine Mehrfachzuweisung folgender Form:

int *x*, *y*;
x = *y* = 1;

Entsprechend der Assoziativität aller Zuweisungs-Operatoren von rechts nach links wird zuerst *y* = 1 bewertet, d.h. *y* erhält den Wert 1, und dann wird der resultierende Wert von 1 an *x* zugewiesen. Dieser Wert ist auch gleichzeitig wiederum der Wert des gesamten Ausdrucks, der hier ignoriert wird.

Neben der einfachen Zuweisung gibt es eine Reihe von Zuweisungs-Operatoren, die mit einem der regulären dyadischen Operatoren verbunden sind. So kann die Zuweisung *a* = *a* **op** *b* generell zu *a* **op**= *b* verkürzt werden. Dies hat den Vorteil, dass *a* nur einmal bewertet werden muss und entsprechende Optimierungen dem Übersetzer erleichtert werden.

Programm 6.5: Zuweisungen (*zuweisung.c*)

```
1  #include <stdio.h>
2
3  int main() {
4      int a, b;
5
6      /* Kurzform fuer b = 13; a = b; */
7      a = b = 13;
8      printf("a=%d, b=%d\n", a, b);
9
10     /* a += 2 als Kurzform fuer a = a + 2
11         ACHTUNG: Die Reihenfolge der Bewertung ist nicht definiert */
12     printf("a=%d, (a+=2)=%d\n", a, a += 2);
13 }
```

```
zuweisung.c: In function `main':
zuweisung.c:12: warning: operation on `a' may be undefined
dublin$ a.out
a=13, b=13
a=15, (a+=2)=15
dublin$
```

Hinweis: Wie bereits erwähnt, ist die Reihenfolge der Parameterbewertungen nicht definiert. Deswegen liefern hier aktuellere C-Übersetzer auch eine entsprechende Warnung. Konkret wäre hier statt der Ausgabe „a=15, (a+=2)=15“ auch die Ausgabe „a=13, (a+=2)=15“ möglich gewesen.

Anhang

Literatur

M. J. Bach: *The Design of the UNIX Operating System.* Prentice Hall, 1986.

D. Comer: *Operating System Design: The XINU Approach.* Prentice Hall, 1984.

P. A. Darnell und P. E. Margolis: *C: A Software Engineering Approach.* Springer, Dritte Auflage, 2001.

D. Goldberg: *What every computer scientist should know about floating-point arithmetic.* ACM Computing Surveys, Jahrgang 23, Heft 1 vom März 1991, Seiten 5-48.

T. Handschuch: *Solaris 2 f"ur den Systemadministrator.* Solaris Galerie, IWT-Verlag, 1993.

S. P. Harbison, G. L. Steele: *C: A Reference Manual.* Fünfte Auflage, Prentice Hall, 2002.

H. Herold: *Linux-Unix-Shells.* Addison-Wesley, 1999.

B. W. Kernighan und R. Pike: *Der UNIX-Werkzeugkasten.* Hanser, 1986.

B. W. Kernighan und D. Ritchie: *Programmieren in C.* Hanser, Zweite Auflage, 1990.

D. E. Knuth: *The Art of Computer Programming*, *Volume 2*, *Seminumerical Algorithms*, Addison-Wesley, Dritte Auflage, 1997.

A. Koenig: *C Traps and Pitfalls.* Prentice Hall, 1989.

M. Rochkind: *UNIX-Programmierung f"ur Fortgeschrittene.* Hanser Verlag, 1988.

W. R. Stevens: *Advanced Programming in the UNIX Environment.* Addison-Wesley, 1992.

B. Stroustrup: *The Design and Evolution of C++.* Addison-Wesley, 1994.

Sun Microsystems: *Code Conventions for the Java Programming Language.* Zu finden im Web unter http://java.sun.com/docs/codeconv/.

A. S. Tanenbaum: *Operating Systems - Design and Implementation.* Prentice Hall, 1987.
$\rightsquigarrow$ **Bitte auf die jeweils aktuellste Auflage achten!**

Abbildungsverzeichnis

Beispiel-Programme

Index