Programmierung in C

Programmentwicklungszyklus

Editieren-Compilieren-Linken-Ausführen-Debugging

Syntaxdefinition mittels EBNF

C - Sprachkonzepte

Quelltext Beispiele

Literatur/Informationsquellen

- Harbison/Steele: C: A Reference Manual. Prentice Hall 1995
- Isernhaben, Rolf/Hartmut Helmke: Softwaretechnik in C und C++. Hanser Verlag 2004
- Krüger, Guido: Programmieren in C. Grundlagen, Konzepte, Übungen. Addison Wesley 1995
- Kernighan, Brian W./Dennis M.Ritchie: Programmieren in C. dt.Übersetzung, Hanser Verlag 1983
- P.J.Plauger/James Brodie: Referenzhandbuch Standard C
  (Für den neuen ANSI- und ISO-Standard)
  Vieweg 1990 (Übersetzung von Microsoft Press 1989)
- ANSI-C: ANSI/ISO 9899:1990 - Programming Language C (C90)
  ANSI/ISO 9899:1999 - Programming Language C (C99) ($18)
  http://webstore.ansi.org/ansidocstore/, http://www.iso.org
- Newsgroup: de.comp.lang.c
- Jutta Degener: http://snake.cs.tu-berlin.de:8081/~jutta/c/
- http://www.c-programme.de/ , http://gcc.gnu.org/ , http://www.snippets.org/
  (Quelltexte für jeden Zweck!)
- Brian W. Kernighan: Programming in C: A Tutorial. Bell Laboratories, Murray Hill, N. J. 1974
  (siehe INTERNET)
- Dinkum C99 Library: http://www.dinkumware.com/manuals/reader.aspx?lib=c99
- GNU C-Compiler unter Windows (GCC)

Historie

1967 BCPL=Basic Combined Programming Language (M.Richards)
- ➔ 1970 Weiterentwicklung von BCPL zur Prog.sprache B für DEC PDP-7 (K.Thompson)
  - ➔ 1971 Programmiersprache C für PDP-11(D.Ritchie)
    - ➔ 1978 Quasistandard K&R-C (Kernighan & Ritchie)
      - ➔ 1983 Programmiersprache C++ (Stroustrup/AT&T)
        - ➔ 1989/90 ANSI C89/ISO-Standard C90
          - ➔ 1999 ISO Standard C99 - ISO/IEC 9899:1999
            - ➔ 2011 ISO Standard C11 - ISO/IEC 9899: 2011
              - ➔ 2018 ISO Standard C18 - ISO/IEC 9899: 2018

Nachteile von C

- ungewöhnliche Syntax, schwer erlernbar, schwer lesbare Programme
- Freiheit der Sprache verführt zu komplizierter Programmierung
- verdeckte Fehlerquellen, die vom Compiler unentdeckt bleiben
- volle Zugriffsmöglichkeit auf Hardware

Vorteile von C

- UNIX in C geschrieben
- C-Compiler erzeugen schnellen effizienten Code
- Grundlage moderner Sprachen (C++)
- Systemnahe Programmierung (auch ohne Assembler)
- Portierbarkeit
- Standardisierung ANSI-C

Zyklus der Programmentwicklung (Turnaroundzyklus - Überblick)

- Modellbildung (Entwurf der Algorithmen und Datenstrukturen)
- Editor → erzeugt Quelldatei *.c
- Compiler → erzeugt Objektdatei *.o
- Linker → erzeugt standardgemäß ein ausführbares Programm a.out
- Debugger ("Entwanzer") → schrittweise Abarbeiten und Testen eines Programmes
- die durch den Linker angebundene Laufzeitumgebung "überwacht" die Abarbeitung des Programmes

Dateien

*.a	Programmbibliothek
*.c	C-Quelltext
*.h	(Header-)Deklarationsdatei
*.i	C-Quellcode nach Präprozessordurchlauf
*.o	Objektcode
*.s	Assembler-Quellcode

Headerdateien

- Headerdatei = Includefile: Informationen über ein Modul, insbesondere...
  1. Funktionsköpfe = Prototypen und
  2. Datendeklarationen
- Definition: Code einer Funktion oder Bytes einer Datenstruktur, exakt 1× pro Programm
- Deklaration: Ankündigung einer Funktion oder Datenstruktur, beliebig oft
- Definitionen und Deklarationen müssen im Einklang stehen
- Folge: Trennung in...
  - Includefiles : enthalten Deklarationen
  - Implementierungsdateien: enthalten Definitionen
- Includefile nennt nur öffentliche, keine modul-internen (lokalen) Angaben
- Includefile beim Übersetzen von Nutzer-Modulen "einschließen" (Konsistenz muss gesichert sein)

Objektfiles

- Isoliert übersetztes Modul liefert Objektfile
- Objektfile enthält...
  1. Binärcode übersetzter Funktionen (prozessorabhängige Binaries der Zielplattform)
  2. Offene Referenzen für Aufruf an Funktionen in fremden Modulen
  3. Einstiegspunkte für Funktionen im eigenen Modul

Binden

- Programm = mehrere Objektfiles
- Werkzeug Linker bindet alle Objektfiles zum kompletten Programm = Executable, Binary
- Linker prüft ob alle Referenzen gesättigt sind

Bibliotheken

- Library = Packung mit vielen Objektfiles, kein Programm
- Vorteile: bequemer Transport, Zusammenschluß verwandter Module
- erscheint nach außen wie großes Objektfile
- Werkzeug Archiver (Name historisch begründet) produziert, zerlegt Libraries
- viele Funktionen oft gebraucht
  (mathematische Funktionen, I/O-Funktionen, Systemcalls, ...)
- vorübersetzt in Objektfiles, gepackt als Bibliotheken
- "ANSI-C" definiert Sprache und Inhalt der Bibliotheken = Standardbibliotheken
- systemweit verfügbar, verlässliche Grundlage

Standardbibliotheken

- /lib/libc.a Ein- und Ausgabe, Zeichenketten, Speicherverwaltung, Systemdienste
- /lib/libm.a mathematische Bibliothek (sin x, cos x, exp x, ln x, ...)

C-Compiler

- Zortech-C (heute Symantec)
- GNU-C: gcc, g++ - GNU project C and C++ Compiler (gcc-2.96)
  SEE ALSO: cpp(1), as(1), ld(1), gdb(1), adb(1), dbx(1), sdb(1).
  `gcc', `cpp', `as', `ld', and `gdb' entries in info.
  siehe: http://gcc.gnu.org/onlinedocs/gcc-3.2.2/gcc/
- Turbo-C (Borland)
- Microsoft-C
- UNIX-Compiler cc

GNU-Compiler Aufruf: `gcc [-Option [...]] Dateiname [,[...]]`

Optionen
-c nur compilieren, nicht linken
-v kommentiert die Übersetzungsschritte
-E nur Präprozessoraufruf
-D externe Makrodefinition
-S nur Assembleraufruf
-O Optimizer anschalten (Codeoptimierung)
-g Debuginformationen generieren
-W Standard-Compilerwarnungen aktivieren
-Wall alle Compilerwarnungen aktivieren
-oName das lauffähige gelinkte Programm mit dem Namen Name versehen

Übersetzen: `cc [<Übersetzeroptionen>] <Datei(en)> [<Binderoptionen>]`

Optionen

-D<name>[=<wert>]	Definition für den C-Präprozessor
-I<includeverzeichnis>	Includeverzeichnis (Standard=/usr/include)
-g	erzeugt zusätzliche Debuginformationen für sdb
-l<lib>	durchsucht angegebene Bibliothek mit den Dateinamen lib<lib>.a
-L<dir>	durchsucht das Verzeichnis <dir> nach den mittels -l spez. Libraries (Standard= /lib und /usr/lib)
-o<name>	Name für das ausführbare Programm (Standard = a.out)
-O	Optimierung des Objektcodes
-p	generiert zusätzlichen Code zur Laufzeitanalyse
-P	nur Präprozessor
-U<name>	Präprozessor-Anweisung: Variable <name> wird undefiniert
-v	alle vom Rahmenprogramm aufgerufenen Einzelprogramme ausweisen
-w	alle Warnungen unterdrücken

shell-Script zur "komfortablen" Kompilierung eines C-Quelltextes

```
if cc -o $1 $1.c
then
echo "Datei $1.c wurde erfolgreich kompiliert" >&2
echo "Programm $1 kann nun gestartet werden" >&2
exit 1
fi
echo "Datei $1.c konnte nicht kompiliert werden" >&2
exit 1
```

symbolischer Debugger

```
$ sdb - debuggt eine core-Datei und zeigt den Laufzeitfehler an
```

Debugger Kommandos am Prompt

```
<zeilen-nr>b  setzt Breakpoint auf Zeile
B   Liste gesetzter Breakpoints
c   continue = Programmfortsetzung nach Haltepunkt
<zeilen-nr>d  delete = löscht Breakpoint auf dieser Zeile
<zeilen-nr>g  go = Programmfortsetzung auf Zeile
k   kill = Programmabbruch
l   list = Ausgabe der zuletzt ausgeführten Zeile
q   quit = Beenden von sdb
r   run = Programmstart
s   step = Ausführung der nächsten Zeile
t   trace = Ausgabe der Aufruf-Rückverfolgungsliste
w   window = Ausgabe von 10 Quelltextzeilen um die aktuelle Position
<funktion>:<variable>   Ausgabe des Wertes von Variable innerhalb der Funktion
<funktion>:<variable>!<wert>  Setzen von Variable innerhalb der Funktion auf
                              den Wert
```

make = Werkzeug zur automatisierten Übersetzung eines C-Projektes mit Modulabhängigkeiten und Berücksichtigung des Datei-Datums

- make geht davon aus, dass sich im aktuellen Verzeichnis eine Datei makefile (Makefile) befindet, in der diese Abhängigkeiten beschrieben sind (mit variablen Einstiegspunkten)
- makefile besteht aus

a) dependency lines (beschreiben Abhängigkeiten): `prog1: prog1.o main.o com1.o`

b) command lines = Kommandos, falls die Quelldateien jünger als die Ziel-Dateien sind;
diese Kommandos werden durch Semikolon getrennt oder in einer neuen Zeile mit <TAB> eingerückt
(\ bedeutet, Fortsetzung in nächster Zeile)
Bsp:
```
prog1.o: prog1.c; cc -c prog1.c
    main.o:main.c
    <TAB> cc -c main.c
```

c) Kommentare (beginnen mit # bis zum Zeilenende)

d) Makros (für Listen, Compileroptionen, ...) der Form
```
VARIABLE = wert
```
Bsp:
```
PROG = prog1.o main.o com1.o
    prog1: $(PROG)< cc -o prog1 $(PROG)
```

e) vordefinierte Makros
```
CC = cc
AS = as
```

f) build-in Makros (interne Makros)
```
makroname = string
```
➔ Zugriff: `$(makroname)` oder
`${makroname}`

$@ = Name des aktuellen Zieles
$$@ = Name des aktuellen Zieles in einer Abhängigkeitsbeschreibung
$* = Name des aktuellen Zieles ohne Suffix
$? = Liste der Dateien, die jünger sind als die von diesen abhängigen Dateien
$< = Name eines neueren Objektes entsprechend den Suffixregeln
$% = Name einer Objektdatei aus einer Bibliothek

g) Marken
```
marke:
```
spezielle Zielangaben: .DEFAULT oder .IGNORE oder .PRECIOUS oder .SILENT oder .SUFFIXES

Aufruf: make [options] [makro definitionen] [marke]

```
-f <filename> anstatt makefile anderes File
-p Ausgabe aller Makrodefinitionen und target-Beschreibungen
-i kein Abbruch bei Fehler
-k bei Fehler Abbruch des aktuellen Eintrages aber Fortsetzung mit
   nichtabhängigen Einträgen
-s keine Anzeige der Kommandos
-r interne Abhängigkeitsregeln werden nicht benutzt
-n Kommandos werden für Testzwecke ausgegeben aber nicht ausgeführt
-t Zielobjekt enthält neueres Datum, ohne dass eine Generierung erfolgt
-d Ausgabe erweiterter Informationen (Datumsausgaben) für Testzwecke
```

weitere Tools

```
ar funktion bibliothek dateien  archive
lint [optionen] quelltexte
```
➔ scharfer C-Parser

grundlegende Problemstellungen für Programmiersprachen

- Datentypen (=Wertevorrat, den die Variablen annehmen können)
- Sichtbarkeit und Lebensdauer für Variablen, Bezeichner, ... in Abhängigkeit des Kontextes ihrer Definition
- Operatoren und Ausdrücke (logische, arithmetische)
- Funktionen und Prozeduren
- mathematische Funktionen
- Stringfunktionen
- Steueranweisungen (Sequenz, Alternative, Zyklen)
- Präprozessoranweisungen (Compileroptionen)

Lebensdauer = Zeitspanne in der eine Variable einen Speicherplatz besitzt (belegt, allokiert)

allgemeine Prinzipien

- Bezeichner sind case-sensitiv
- Bezeichner ohne Umlaute, ohne Sonderzeichen und mit Buchstaben beginnend
- Bezeichner (für Typen, Variablen und Funktionen) dürfen keine reservierten Schlüsselwörter sein
- max. Länge von Bezeichnern ist Compiler-abhängig
- max.Zeilenlänge=?
- Leerzeichen, Tabulator und CRLF sind Trennsymbole
- Formatierung des Quelltextes beliebig (aber: ungarische Notation oder gar Formatierungsrelevanz)
- C-Konventionen für Bezeichner und Formatierung
- Fortsetzungssymbol "\"

Präprozessor

- Preprocessor-#include-Anweisungen
- Preprocessor-#define-Anweisungen
- Textersatz/Alias
- Makros

```
#include <stdio.h>
#define MAXIMUM 100
#define PI 3.14 const
#define <name> <ersatztext>
#define SQUARE(x) ((x)*(x))
```

bedingte Kompilierung:

```
#if TESTPHASE == 1
# define PROGRAMMVERSION "Testversion"
# elif TESTPHASE == 2
# define PROGRAMMVERSION "Alpha-Release"
# else
# define PROGRAMMVERSION "Final-Release"
#endif
```

Programmiersprache C

Syntaxbeschreibung mittels

a) EBNF = Erweiterte Backus-Naur-Form
= Erweiterung der BNF von J.Backus und P.Naur zur Definition von Algol60
= Metasprache zur Definition der Syntax einer Programmiersprache

Terminalzeichen = Schlüsselworte der zu definierenden Sprache
Nichtterminalsymbole = Variablen
Metazeichen = Zusammenhang zwischen Terminalzeichen und Nichtterminalsymbolen

::=	Definition
|	Alternative
[]	Option (Möglichkeitssymbol)
{}	Iteration (keinmal, einmal oder mehrmals)
()	Gruppierung
..	Bereich

b) Syntaxdiagramme

C-Syntaxbeschreibung in EBNF

```
lexikalische Sprachelemente
Kommentar::= /* Zeichen */
Bezeichner::= (Buchstabe | Unterstrich) {Buchstabe | Unterstrich | Ziffer}
Konstante::= Zahl | " { Zeichen } " | 'Zeichen'
Zahl::= GanzZahl | [+|-] Ziffer {Ziffer} [.{Ziffer}] [(e|E) [+|-]
                  Ziffer {Ziffer}]
GanzZahl::= [+|-] Ziffer {Ziffer} [ l | L | u | U ]   |
            [+|-] 0 OctZiffer {OctZiffer} [ l | L | u | U ]   |
            [+|-] 0xHexZiffer {HexZiffer} [ l | L | u | U ]

  hierbei bedeuten: l, L  long und u, U unsigned
Buchstabe::= (A..Z | a..Z)
Unterstrich::= _
Ziffer::= 0..9
OctZiffer::= 0..7
HexZiffer::= 0..9 | A..F | a..f
Zeichen::= alle verfügbaren Zeichen

Ausdrücke
Ausdruck::= EinfachAusdruck | PrefixOperator Ausdruck | Ausdruck
PostfixOperator
            | sizeof Ausdruck | Ausdruck BinOperator Ausdruck |
            Ausdruck , Ausdruck | Ausdruck ? Ausdruck : Ausdruck
```

```
EinfachAusdruck::= Bezeichner | Konstante |
                   (Ausdruck) | LWert.Bezeichner |
                   EinfachAusdruck ( [Namensliste] ) |
                   EinfachAusdruck [ Ausdruck ] |
                   EinfachAusdruck -> Bezeichner
PrefixOperator::= * | & | - | + | ! | ~ | ++ | --
PostfixOperator::= ++ | --
LWert::= Bezeichner | EinfachAusdruck [ Ausdruck ] |
               EinfachAusdruck -> Bezeichner |
               LWert.Bezeichner | * Ausdruck |  ( LWert)
BinOperator::=  *  |  /  |  % |
                +  |  -  |
                >> | <<  |
                <  |  >  |  <=  | >=  |
                == |  != |
                &  |  ^  |  |   |  && |  ||  |
                =  | +=  |  -=  |  *= |  /=  | %= |
               >>= | <<= |  &=  |  ^= | |=
KonstAusdruck::= wie Ausdruck, aber zur Übersetzungszeit bekannt
```

Anweisungen

```
Anweisung::= Block | Ausdruck; |
             if (Ausdruck) Anweisung |
             if (Ausdruck) Anweisung else Anweisung |
             while ( Ausdruck ) Anweisung |
             do Anweisung while ( Ausdruck ) |
             while ( Ausdruck ) ; |
             for ([Ausdruck];[Ausdruck];[Ausdruck]) Anweisung |
             switch (Ausdruck) { {case KonstAusdruck: Anweisung}
                                   [default: Anweisung]
                                 }             |
             break; | continue; | return Ausdruck; | goto Bezeichner;|
             Bezeichner: Anweisung |  ;
Block::= { {DatenDeklaration} {Anweisung}  }
```

Deklarationen (Typ-, Variablenvereinbarungen)

d.h. Definitionen und Typ-Zuordnung
(Typ = Wertevorrat, den eine Variable annehmen kann)

```
Deklaration::= DatenDeklaration | Funktionsdeklaration
DatenDeklaration::= EinfachDeklaration |
                    StruktDeklaration |      (d.h. Strukturen/
Variablenverbund)
                    EnumDeklaration          (d.h. Aufzählungen)
EinfachDeklaration::= {Speicherklasse} {TypModi} Typname [*] Bezeichner
                      [ [GanzZahl] | () ] [Init] {, [*] Bezeichner
                     [ [GanzZahl] | () ] [Init] };
StruktDeklaration::= struct [Bezeichner] { {Datendefinition} {BitFeld} }
                           [Namensliste] [Init] ; |
union [Bezeichner] {  {Datendefinition}  } [Namensliste] [Init] } ;
EnumDeklaration::= enum [Bezeichner]  { Bezeichner [=GanzZahl] {, Bezeichner
                        [=GanzZahl] }  }  [Namensliste] ;
Funktionsdeklaration::= {TypModi} Typname [*] Bezeichner (Namensliste)
                        {Datendeklaration} Block
Speicherklasse::= auto | static | extern | register | typedef
TypModi::= short | long
Typname::= char | short | int | long | unsigned | float | double | Bezeichner
Init::= =Konstante | = {Konstante {, Konstante} }
Bitfeld::= unsigned Bezeichner : GanzZahl; | unsigned: GanzZahl;
Namensliste::= Bezeichner { , Bezeichner }
```

Präprozessor

```
PräprozessorAnweisungen::=
   #define Bezeichner {Zeichen}   |
   #define Bezeichner  (Namensliste) {Zeichen}  |
   #undef Bezeichner  |
   #include <Dateiname>  |
   #include "Dateiname"  |
   #if KonstAusdruck {Zeichen}
        [ #elif KonstAusdruck {Zeichen} ]
        [ #else {Zeichen} ]
       #endif        |
   #ifdef Bezeichner {Zeichen}
        [ #elif Bezeichner {Zeichen} ]
        [ #else {Zeichen} ]
        #endif       |
   #ifndef Bezeichner {Zeichen}
        [ #elif Bezeichner {Zeichen} ]
        [ #else {Zeichen} ]
        #endif       |
   #line GanzZahl Dateiname

Dateiname::= {Zeichen} [.{Zeichen}
```

```
Programm
Programm::= { [PräprozessorAnweisung] [Deklaration] [Anweisung]  main
() { [Deklaration] [Anweisung] }  }
```

Mein erstes C-Programm:

```
/* Hello World Programm in C */
main()
{
  printf("Hello World\n");
}
```

Variablen (Datentyp & Gültigkeitsbereich)

lokale Variable

```
main()
{
   int i,j;
}
```

Die Lebensdauer beschränkt sich auf die
Ausführungszeit von main().

globale Variablen (werden außerhalb von Funktionen definiert)

```
float x;     /* globale Variable */
test_funktion()
{
    float z; /* lokale Variable */
    /* Anweisungen */
}
main()
{
   int i,j;
   /* Anweisungen */
}
```

```
Beispiel Deklarationen
char c;
int i,j,k;
float x,y,z;
char *dateiname;
int i, j=0;
char chlf='\n';
char *hello="Hello world\n";
```

Ausdrücke

a+=b bedeutet a=a+b (Vorteil besteht darin, dass die Adresse von a nur einmal bestimmt werden muß)

c=(a+=b) bedeutet a=a+b und c=a+b (eine Zuweisung ist selbst ein Ausdruck, der einen Wert annimmt; Ausdrücke ohne Zuweisung, wie z.B. a++ haben einen Sinn, aufgrund ihrer Nebeneffekte/Seiteneffekte)

Bsp.	Ergebnis	Nebeneffekt
`x++`	`x`	`x=x+1`
`x=y++`	`y`	`x=y, y=y+1`
`++x`	`x+1`	`x=x+1`
`a==b`	`<>0 falls gleich` `=0 falls ungleich`	
`0 && 1`	`0`	
`2==2 && x<=x`	`ungleich 0`	
`a && a`	`a`	
`0 || 1`	`ungleich 0`	
`a || a`	`ungleich 0`	
`1<<5`	`32`	
`22<<1`	`44`	
`a<<0`	`a`	
`~1`	`-2`	
`~a`	`-a-1`	

Operatoren für Ausdrücke (1)

Additions-Operator	a + b
Subtraktions-Operator	a - b
Multiplikations-Operator	a * b
Divisions-Operator	a / b
Restwert-Operator (modulo)	a % b
Additionszuweisung	a += b
Subtraktionszuweisung	a -= b
Multiplikationszuweisung	a *= b
Division und Zuweisung	a /= b
Restwertbildung (modulo) und Zuweisung	a %= b
bitweise-UND Zuweisung	a &= b
bitweise-ODER Zuweisung	a |= b
bitweise-XOR Zuweisung	a^=b
Linksverschiebeoperator Zuweisung	a<<=b
Rechtsverschiebeoperator Zuweisung	a>>=b
bitweise-UND	a & b
bitweise-ODER	a | b
bitweise exclusiv-ODER	a^b
Linksverschiebeoperator	a<<b
Rechtsverschiebeoperator	a>>b
Einerkomlement/Negation	~a (=-a-1)

Operatoren für Ausdrücke (2)

Postfix-Inkrement	a++
Postfix-Dekrement	a--
Präfix-Inkrement	++a
Präfix-Dekrement	--a
sizeof-Operator	sizeof(a)
Gleichheit	a==b
Ungleichheit	a != b
Kleinergleich	a<=b
Größergleich	a>=b
Kleiner	a<b
größer	a>b
logisches UND	a && b
logisches ODER	a || b
Kommaoperator	a,b (Wert = Wert von b)
Bedingung (einziger dreistelliger Operator)	a?b:c
Funktionsaufruf	f(x)
Typkonvertierung	(type) a
Adressoperator	&i
Umleitungsoperator	*i Variable auf die i zeigt
Feldindexoperator	a[i]
Elementauswahl-Operator	a.b (Strukturname.Elementname)
Elementkennzeichnungs-Operator	a -> b (entspricht (*a).b , d.h. das Element b einer Struktur/Union, auf die der Pointer a zeigt)

weitere Beispiele:

```
unsigned char a,b,c; /* Bitnummer: 7654 3210 */
 a=0x11; /* = 17 Bitmuster: 0001 0001 */
 b=0x0F; /* = 15 0000 1111 */
 c=a & b; /* c wird gesetzt auf: 0000 0001 */
 c=a | b; /* c wird gesetzt auf: 0001 1111 */
 c=a ^ b; /* c wird gesetzt auf: 0001 1110 */
 c=a << 1; /* c wird gesetzt auf: 0010 0010 */
 c=b >> 2; /* c wird gesetzt auf: 0000 0011 */
 c=~a; /* c wird gesetzt auf: 1110 1110 */
strlen("Hello")    liefert den Wert 5
printf("OK\n")     liefert den Wert 3
if (x==y) i=1; else i=0;
x==y?(i=1):(i=0);
i=(x==y)?1:0;
i=x==y?1:0;
```

Auswertungsreihenfolgen
Links-Assoziativität / Rechts-Assoziativität

Ein-/Ausgabe (stdio.h)

Daten-Ausgabe: printf("ausgabetext");

Daten-Eingabe: scanf("formatstring", &variablenname);
zeichen = getch(); /* ein Zeichen einlesen */

Formatzeichen:

-	linksbündig
+	Ausgabe des Vorzeichens "+" oder "-"
leerzeichen	Leerzeichenausgabe, falls positives Vorzeichen (Platzhalter)
0	führende Nullen bei numerischen Werten
#	bei Oktal- oder Hexadezimaldarstellung: Wert mit vorangestelltem 0 bzw. 0x bei %e, %E, %f Wert mit Dezimalpunkt bei %g, %G: Wert mit Dezimalpunkt und Nachkommanullen

Formatstrings (bestimmen Form der Ausgabe):

%i	Werte vom Typ Integer
%d	Werte vom Typ Integer
%f	Werte vom Typ Float in der Darstellung [-]ddd.ddddd (Standard 6 Stellen hinter dem Komma)
%F	Werte vom Typ Float in der Festkommadarstellung bzw. INF, INFINITY, NAN (neu in C99)
%e	Werte vom Typ Float in der Exponentialdarstellung mit [-]d.dddedd
%E	Werte vom Typ Float in der Exponentialdarstellung mit [-]d.dddEdd
%g	Werte vom Typ Float in der Festkomma- oder Exponentialdarstellung (je nach Exponent)
%G	Werte vom Typ Float in der Festkomma- oder Exponentialdarstellung (je nach Exponent)
%a	hexadezimale Gleitpunktdarstellung (neu in C99)
%A	hexadezimale Gleitpunktdarstellung (neu in C99)
%s	Werte vom Typ String (Zeichenkette)
%c	Werte vom Typ Character
%p	Wert als Pointer/Adresse anzeigen
%o	vorzeichenlose (unsigned) ganze Oktalzahl
%u	vorzeichenlose (unsigned) ganze Zahl
%x	vorzeichenlose ganze Hexadezimalzahl mit den Ziffern 0...9,a,b,c,d,e,f
%X	vorzeichenlose ganze Hexadezimalzahl mit den Ziffern 0...9,A,B,C,D,E,F
%%	das Zeichen % selbst

%2i	ganze Zahl mit 2 Stellen
%2f	gebrochene Zahl mit 2 Stellen
%6.2f	gebrochene Zahl mit insgesamt 6 Stellen, davon 2 hinter dem Komma
%.2f	gebrochene Zahl mit 2 Stellen nach dem Komma
%[abc]	nur die Zeichen a oder b oder c sind zur Eingabe zugelassen

Bildschirmfunktionen (conio.h):

```
clrscr;          /* Clear Screen, Löschen des Bildschirminhaltes  */
gotoxy(x,y);     /* positionierte Ausgabe */
```

Steuerzeichen

\a	bzw. \007 Piepton (Bell)
\b	Backspace
\f	Seitenvorschub (FormFeed)
\n	Newline, neue Zeile (LineFeed)
\r	Wagenrücklauf (am Anfang der Zeile positionieren / Carriage Return)
\t	Tabulator
\v	vertikaler Tabulator
\ooo	Zeichen mit Oktalcode ooo
\xhh	Zeichen mit Hexadezimalcode hh
\´	Hochkomma
\"	Anführungszeichen
\\	Backslash

Trigraphen (Erzeugung bestimmter Zeichen)

Dreizeichenfolge (Trigraph) ...ersetzt das Zeichen

??=	#
??(	[
??)	]
??/	\
??'	^
??<	{
??>	}
??!	|
??-	~

Datei Ein- und Ausgabe:

```
fgets(line, MAXLINE, fp)   liest die nächste Eingabezeile aus der Datei fp
                         einschließlich Zeilenendekennung und hängt \0 an
fputs(line, fp)  Ausgabe
```

```
/* Datei einlesen */
#include <stdio.h>

char *fgets(s,n, iop);
char *s;
int n;
register FILE *iop;
{ register int c;
  register char *cs;
  cs = s;
  while ((--n>0) && ((c=getc(iop)) != EOF ))
         if (( *cs++ = c ) == '\n' ) break;
  *cs = '\0';
  return ((c==EOF && cs==s) ? NULL : s);
}
```

```
fputs(s,iop)
register char *s;
register FILE *iop;
{ register int c;
  while (c = *s++) putc(c, iop);
}
```

String-Funktionen (siehe <string.h>)

```
strcpy(ziel,quelle);                    /* Kopieren einer Zeichenkette */
strncpy(ziel,quelle);                   /* Kopieren von n Zeichen einer Zeichenkette */
gets(zeichenkette);                     /* Einlesen einer Zeichenkette */
strcat(zeichenkette1,zeichenkette2);    /* zwei Strings verknüpfen (concat) */
strncat(zeichenkette1,zeichenkette2);   /* Anhängen von n Zeichen eines Strings an anderen String */
strcmp(zeichenkette1,zeichenkette2);    /* Vergleich zweier Strings */
strncmp(zeichenkette1,zeichenkette2);   /* Vergleich n Zeichen von 2 Strings */
strlen(zeichenkette);                   /* Länge der Zeichenkette */
strchr(zeichenkette,zeichen)            /* Suchen eines Zeichens in einem String */
strrchr(zeichenkette,musterzeichen)     /* rückwärtiges Suchen eines Zeichens in einem String */
strstr(zeichenkette1,zeichenkette2)     /* Suchen eines Strings in einem String */
strpbrk(zeichenkette1,zeichenkette2)    /* 1.Vorkommen eines Zeichens aus einer Zeichenmenge */
strspn(zeichenkette1,zeichenkette2)     /* Zahl von Zeichen aus einer Zeichenmenge am Stringanfang */
strcspn(zeichenkette1,zeichenkette2)    /* Zahl von Komplementzeichen aus einer Zeichenmenge am
                                           Stringanfang */
strtok(zeichenkette1,zeichenkette2)     /* Zerteilen eines Strings nach vorgegebenen "Bruchstellen" */
strerror(fehlernummer)                  /* Text zu einer Fehlernummer */
strcoll((zeichenkette1,zeichenkette2)   /* länderspezifisches strcmp() */
strxfrm(nach,von,groesse)               /* Umwandeln länderspezifischer Strings */
memcpy(ziel,quelle,n)                   /* Kopieren von n Bytes eines Speicherbereiches */
memmove(ziel,quelle,n)                  /* Kopieren von n Bytes eines Speicherbereiches */
memset(adresse,zeichen,n)               /* n-maliges Schreibens eines Zeichens in Speicherbereich */
memcmp(adresse1,adresse2,n)             /* Vergleichen von Bytes in zwei Speicherbereichen */
memchr(adresse,suchzeichen,n)           /* Suchen eines Zeichens in einem Speicherbereich */
strtol(zeichenkette,endezeiger,basis)   /* Umwandlung eines Strings in long */
strtoll(zeichenkette,endezeiger,basis)  /* Stringumwandlung in long long */
strtoul(zeichenkette,endezeiger,basis)  /* Stringumwandlung in unsigned long */
strtoull(zeichenkette,endezeiger,basis) /* Umwandlung in unsigned long long */
strtod(zeichenkette,endezeiger)         /* Umwandlung eines Strings in double */
strtold(zeichenkette,endezeiger)        /* Stringumwandlung in long double */
atoi(zeichenkette)                      /* Umwandlung eines Strings in int */
atol(zeichenkette)                      /* Umwandlung eines Strings in long */
atoll(zeichenkette)                     /* Umwandlung eines Strings in long long */
atof(zeichenkette)                      /* Umwandlung eines Strings in double */
sscanf(zeichenkette,format, ...)        /* Umwandeln eines Strings in numerische Werte */
sprintf(zeichenkette,format,...)        /* Umwandeln numerischer Werte in Strings */
```

wichtige Funktionen:(#include <math.h>)

```
rand()                  /* liefert Zufallszahlen */
sqrt(double x)          /* Quadratwurzel */
pow(double x, double y) /* x^y */
fabs(double x)          /* Absolutwert von x */
ceil(double x)          /* Aufrunden von x */
floor(double x)         /* Abrunden von x */
exp(double x)           /* e^x */
log(double x)           /* natürlicher Logarithmus von x */
log10(double x)         /* Zehnerlogarithmus von x */
sin(double x)           /* Sinus */
cos(double x)           /* Cosinus */
tan(double x)           /* Tangens */
asin(double x)          /* Arcus-Sinus */
acos(double x)          /* Arcus-Cosinus */
sinh(double x)          /* Sinus hyberbolicus */
cosh(double x)          /* Cosinus hyberbolicus */
tanh(double x)          /* Tangens hyberbolicus */
sinh(double x)          /* Sinus hyberbolicus */
cosh(double x)          /* Cosinus hyberbolicus */
tanh(double x)          /* Tangens hyberbolicus */
cbrt(double x)          /* Kubikwurzel */
trunc(double x)         /* ganzzahliger Anteil von x */
```

Trick:

```
strcopy(s1, s2) {char s1[], s2[]; int i; for (i=0;s2[i]=s1[i] !='\0'; i++);}
oder i=0; while (s2[i]=s1[i] !='\0') i++;
```

Datentypen

Speicherklassen

auto	kennzeichnet eine Variable, deren Allokation beim Eintritt in den Gültigkeitsbereich und Deallokation beim Austritt daraus erfolgt;
extern	sind globale Variablen, die für die gesamte Programmlaufzeit allokiert werden;
register	entspricht auto, nach Möglichkeit wird jedoch statt eines Speicherplatzes ein Register belegt; bei modernen Compilern ist dieses Schlüsselwort relativ obsolet, da bei der Codegenerierung in der Regel sowieso optimiert wird.
static	kennzeichnet Variablen, deren Allokation für die gesamte Programmlaufzeit erfolgt, deren Gültigkeit jedoch block- oder modullokal festgelegt ist.

Überblick
Datentypen in C99

- bool — Wahrheitswert - #include <stdbool.h>
- complex — Komplexe Zahl - #include <complex.h>
- imaginary — Komplexe Zahl
- char — Zeichen (1 Byte)
- double — Gleitkommazahl, doppelte Genauigkeit
- enum — Kennzeichnung für Aufzählungstyp
- float — Gleitkommazahl
- int — Ganzzahl
- long — Ganzzahldatentyp (auch long long)
- long int — Ganzzahldatentyp
- unsigned — Kennzeichnung zur Interpretation eines Datentyps ("ohne Vorzeichen"), z.B. "unsigned int" oder "unsigned char"
- short — Ganzzahl
- signed — Kennzeichnung zur Interpretation eines Datentyps als vorzeichenbehaftet, z.B. signed int (=int) oder signed char
- struct — Struktur (=Record)
- typedef — Eigendefinition von Datentypen
- union — Variante Struktur (=varianter Record)
- void — "leerer" Datentyp
- const — kennzeichnet einen Datentyp für einen nicht veränderbaren Speicherplatz.
- volatile — Typattribut für eine Variable, die durch externe Einflüsse (von außerhalb des Programmes) verändert werden kann, beispielsweise die Systemuhr

Datentypen

Datentyp	Anzahl Bits	Bereich
char	8	-128 … 127
signed char	8	-128 … 127
unsigned char	8	0 … 255
short	16	-32768 … 32767
unsigned short	16	0 … 65535
int	32	-2147483648 … 2147483647
unsigned int	32	0 … 4294967295
long	32	-2147483648 … 2147483647
unsigned long	32	0 … 4294967295
long long	64	-9223372036854775808 … 9223372036854775807
unsigned long long	64	0 … 18446744073709551615

```
/* /usr/include/limits.h .. stark gekürzt .. */
#define CHAR_BIT 8 /* Number of bits in a char */
#define CHAR_MAX 127 /* Max integer value of a char */
#define CHAR_MIN (-128)/* Min integer value of a char */
#define INT_MAX 2147483647 /* Max decimal value of an int */
#define INT_MIN (-2147483648)/* Min decimal value of an int */
```

Gleitkommazahlen (siehe <float.h> für Grenzwerte von Gleitpunkt-Datentypen)

```
float       4 Bytes = 32 Bits
double      8 Bytes = 64 Bits
long double 10 Bytes = 80 Bits
```

```
/* /usr/include/float.h .. stark gekürzt .. */
#define FLT_MAX 3.40282347E+38
#define FLT_MAX_10_EXP 38
#define DBL_MAX 1.7976931348623157E+308
#define DBL_MAX_10_EXP 308
```

Zeichenketten / nullterminierte Strings

char hello[] = {'h' , 'e', 'l', 'l', 'o' };

Bitfelder

```
struct {
   short pers_nr;
   char name[20];
   unsigned stand:2;
   unsigned geschlecht:1;
   unsigned alter:7;
   unsigned kindzahl:4;
   unsigned religion:2;
} person[100];
```

Regeln:

- namenlose Bitfelder (z.B. unsigned :8;) dürfen nicht am Anfang einer Struktur stehen)
- ein Bitfeld muß vom Datentyp int, unsigned int, signed int, oder char, unsigned char, signed char sein
- jedes einzelne Bitfeld darf die Länge eines Maschinenwortes (32 Bit) nicht überschreiten
- Bitfelder werden in der gleichen Speichereinheit untergebracht
- Anordnung von Bitfeldern (von rechts nach links o.u.) ist compilerabhängig
- Arrays von Bitfeldern sind nicht erlaubt
- Bitfelder besitzen keine Adressen, d.h. Adressoperator & darf nicht angewendet werden

Aufzählungen

```
enum ampel {rot, gelb, gruen} ;
enum tag {mo, die=2 , mi, do, fr, sa, so);
enum monat {JAN, FEB, MRZ, APR, MAI, JUN, JUL, AUG, SEP, OKT, NOV, DEZ);
enum boolean { FALSE, TRUE };
```

Regeln:

- enum-Werte müssen eindeutig sein
- enum-Wertenamen dürfen nicht als Variablenbezeichner verwendet werden
- enum-Wertenamen können auch Werte zugewiesen werden

Bsp: enum wochentage { MO=1, DI, MI, DO, FR, SA, SO } tag; ➔ for (tag=MO;tag<=SO;tag++) {...};

Typdefinitionen

typedef enum { false = 0; true = 1;} bool;

Strukturen

(Elemente liegen hintereinander)

```
struct {
   int pers_nr;
    char name[20];
   char vorname[20];
   char adresse[40];
   char std_oder_gehalt;
   struct {
        float gehalt;
        struct {
              float std_zahl;
           float std_lohn;
         }  lohn;
    }  geld;
}  person[1000];
```

Union

(Elemente liegen alternativ übereinander)

```
struct {
   int pers_nr;
    char name[20];
   char vorname[20];
   char adresse[40];
   char std_oder_gehalt;
   union {
        float gehalt;
        struc {
              float std_zahl;
           float std_lohn;
         }  lohn;
    }  geld;
}  person[1000];
/* gehalt und lohn liegen
übereinander */
```

Zugriff auf Komponenten/Elemente eines Strukturtyps: person[123].geld.lohn.std_zahl;

Programmablauf- Steueranweisungen

```
/* Block */
{
  deklarationen
  anweisungen
}
```

```
/* Mehrfachverzweigung */
switch (ausdruck)
{ case ausdruck1: anweisungen1;
  case ausdruck2: anweisungen2;
  ...
  case ausdruckN: anweisungenN;
  default: anweisungen;
}
```

```
/* while-Zyklus */
while (ausdruck)
{ anweisungen;}
```

```
/* do-Zyklus */
do
   anweisungen;
while (bedingung);
```

```
/* Alternative */
if (bedingung)
{ anweisungen; }
else
{ anweisungen; }
```

```
/* bedingte Zuweisung */
wert = (bedingung)?wert1:wert2;
```

```
/* FOR-Zyklus */
for (anfangswert;wiederholbedingung;Veränderung)
       { anweisungen; }
```

```
/* Sprünge */
break;
continue;
goto marke;
```

weitere Themen:

Zeiger
Strings
Zahlkonvertierungen/Typkonvertierungen
Speicherklassen (static, auto, register, global, extern, const, volatile)
Dateiarbeit (FILE, fopen(), fclose(), fscanf(), fprintf(), stdin, stdout, stderr, fread(), fwrite(), ungetc(), vprintf(), vfprintf(), sscanf(), sprintf(), snprintf(), vsprintf(), vsnprintf(), vscanf(), vfscanf(), vsscanf(), fseek(), ftell(), fsetpos(), fgetpos(), freopen(), rewind(), remove(), unlink(),rename(), setbuf(), setvbuf(),tmpnam(),fflush(), tmpfile(),perror(), strerror(), open(), creat(), close(), read(), write(), lseek(), fileno(), fdopen())
Directories (stat(), fstat(), mkdir(), rmdir(), chdir(), getcwd(), opendir(), readdir(), rewinddir(), closedir())
Prozesse/Prozessumgebung (main(), exit(), _exit(), atexit(), getenv(), putenv(), setenv(), unsetenv())
Systemfunktionen (system(), exec(), signal(), abort(), sleep())
STL - Standard Library

Dateien

```
#include <stdio.h>
#include <stdlib.h>
FILE *in, *out;
in = fopen("datei1.txt", "r"); /* zum Lesen öffnen */
out = fopen("datei2.txt", "r"); /* zum Schreiben öffnen */
fclose(in); fclose(out);
```

Datei-Modi

Modus	Bedeutung
"r" oder "rb"	(read) zum Lesen
"w" oder "wb"	(write) zum Schreiben
"a" oder "ab"	(append) zum Anhängen ans Dateiende
"r+", "r+b" oder "rb+"	Lesen und Schreiben
"w+", "w+b" oder "wb+"	Lesen und Schreiben (ohne dass Dateiinhalt gelöscht wird)
"a+", "a+b" oder "ab+"	Lesen und Anhängen (Schreiben)