Programmieren in C/C++

Prof. Dr.-ing. Detlef Justen

de.wikibooks.org

30. März 2024

Inhaltsverzeichnis

1 Vorwort

1.1 Wie ist dieses Buch entstanden

Dieses Buch stammt aus der Vorlesung 'Programmieren in C' und dient dort als Skript für diese Vorlesung. Die Motivationen für die Veröffentlichung des Skripts in Wikibooks sind:

- Bisher kein vernünftiges Lehrbuch für die Programmiersprache C gefunden. Viele Bücher basieren vorrangig auf praktischen Beispielen und vernachlässigen syntaktische Aspekte
- Bereitstellung des Wissens für Personen außerhalb der Vorlesung
- Interaktivität: Das heißt vorerst nicht, dass hier jeder mitschreiben soll. Vielmehr geht es mir um Rückmeldung bzgl. Verbesserungsvorschläge, Ergänzungsvorschläge,... über Diskussionsbeiträge

1.2 Für wen ist dieses Buch

Die Zielgruppe für dieses Buch sind Personen mit Programmierkenntnissen. Es wird vorausgesetzt, dass der Leser weiß, was eine IF-Bedingung ist und wann eine FOR-Schleife anzuwenden ist. Schwerpunkt dieses Buches ist der Syntax der Sprache C/C++, die 'Interpretationen' daraus und die praktische Anwendung hiervon. Der Syntax von C/C++ kennt nur wenige Ausnahmeregeln, so dass quasi alles an allen Stellen erlaubt ist. Für unerfahrene Nutzer dieser Sprache führt dies oftmals zu Frust und der Aussagen wie 'Scheiß Compiler' / 'Scheiß Sprache'. In der Tat liegt das Problem vor der Tastatur. Moderne Programmiersprachen haben aus dem offenem Syntax von C/C++ gelernt und den Syntax entsprechend eingeschränkt. D.h. bspw. den Komma-Operator nur in FOR-Schleifen erlaubt und Zeiger durch Referenzen ersetzt. Damit wurden diverse Fehlerquellen unterbunden, aber gleichermaßen die Flexibilität der Sprache reduziert. Bei korrekter Anwendung der Syntaxen kann auch mit C/C++ saubere, sichere und vor allem schnelle Programme geschrieben werden.

Im Vergleich zu anderen Sprachen entspricht C (mit den Worten eines geschätzten Kollegen gesagt) einem 'senkrechtstartenden Düsenjäger'. Wenn der Pilot (resp. der Programmierer) das Werkzeug nicht beherrscht, führt es schnell zum Crash ('Scheiß Compiler'/'Scheiß Sprache'). Wenn er es beherrscht, kann er die Vorteile beider Welten nutzen.

1.3 Compiler ist dein Freund und Helfer

Fehlermeldungen des Compilers werden nur ungern gelesen (derweil sie sich kryptisch anhören). Anstatt die Fehlermeldung zu lesen und zu verstehen, sehe ich oftmals folgenden Umgang mit Fehlermeldungen:

- Nutzung der Lösungsvorschläge einer integrierten Entwicklungsumgebung

- Vermeidung der Fehlermeldung durch Umstellung des Codes:

Mit Compilerfehler

```
int *foo(void) {
  int arr[10];
  ...
  return arr;
  //Error, aufgrund der Rückgabe
  //einer Adresse einer lokalen
  //Variablen
}
```

Umgehung der Fehlermeldung

```
int *foo(void) {
  int arr[10];
  int *ptr;
  ...
  ptr=arr;
  //Adresse wird zuvor in eine
  //lokale Variable kopiert, so
  //dass der Compiler nicht prüfen
  //kann, woher die Adresse stammt
  return ptr;
}
```

- Googlen nach Lösungsvorschlägen und Nutzung dieser (wobei die Autoren der Lösungsvorschläge den Fehler oftmals ebenfalls nicht verstanden haben und somit nur ein Workaround vorschlagen)
- Ignorieren von Warnings (eigentlicher Fehler rächt sich dann zumeist später)

```
void foo(void) {
  int *ptr;
  *ptr=4711;  //Warning, ptr is uninitialized
}
```

In der Tat ist die Fehlermeldung kryptisch. Der Grund für eine Fehlermeldung ist, dass der Source-Code von der Spezifikation abweicht und der Compiler entsprechend keinen Assemblercode generieren kann. Die Fehlermeldung beschreibt somit die Syntaxverletzung. Sie ist eine Hilfestellung an den Programmierer, was genau falsch ist. Als Programmierer sollte man sich bei Fehlermeldungen überlegen, was wollte man mit dem Ausdruck erreichen und was hat man in der Tat beschrieben. Also nicht einfach den Fehler korrigieren, sondern die Fehlermeldung als Anregung verstehen, den Code zu überdenken.

In den letzten Jahren sind die Compiler so intelligent geworden, dass diese bei vielen typischen Programmiererfehlern eine Warning ausgeben. Sie meckern, wenn bspw. eine Zuweisung in einer IF-Bedingung steht oder einer IF-Anweisung kein Block folgt (beides ist syntaktisch korrekt, sind aber gleichermaßen auch typische Zeichen für mögliche Fehler). Defaultmäßig sind einige diese Warnings deaktiviert. Da sie helfen können, mögliche Fehler zu vermeiden empfiehlt es sich, den Warning Level auf den höchsten Mode zu setzen (Compilerschalter: -Wall).

1.4 Programmieren lernt man nur durch Programmieren

Lesen ist ein wichtiger Aspekt des Lernens, ausprobieren ein Weiterer. Beide ergänzen sich prima. In diesem Sinne sind im Buch viele Code-Beispiele vorhanden, ohne und zum Teil mit bewussten Fehlern.

Damit die notwendigen Include Anweisungen und ggf. Funktionsrümpfe den Beispielcode nicht unnötig aufblähen, wurde ein LUA Script entworfen (Danke an meinen Mitarbeiter), welches den im (verborgenen) im Buch enthaltenen Code in einen Compiler Explorer Link konvertiert. Über 'Öffnen im Compiler Explorer' können alle Beispiele ohne Abtippen direkt ausprobiert werden. Änderungen im Compilerexplorer sind lokale Änderungen und für den 'Rest' nicht sichtbar.

Über das LUA Script kann nicht nur Code vorgegebenen werden, sondern auch Compilerschalter und mehr gesetzt werden. Die Dokumentation zum Lua Code ist hier zu finden: Lua Doku[1]. Der Lua Code selbst ist hier zu finden: Lua Code[2].

1.5 C/C++

Das Augenmerk des Buches liegt derzeit bei der Programmiersprache C. Viele grundlegende Konzepte sind in C und C++ identisch. Da C++ eine eigene Sprache ist, sind einige syntaktischen Abweichungen vorhanden. Im derzeitigen Stand des Buches bedeutet die Ergänzung C++ im Titel des Buches, diese syntaktischen Abweichungen zu beschreiben. Zukünftig ist geplant, die ergänzenden Konzepte wie Objektorientierung, Templates, Namensraum, ... zu ergänzen.

1.6 Compiler

Die C-Spezifikation ist die Grundlage für alle Compilerhersteller. Insofern sind diese austauschbar. Die Spezifikation regelt das allgemeine Verhalten der Sprache, wie der Compiler dieses umsetzt, obliegt ihm. Über Compiler spezifische Schalter kann dem Compiler detailliertere Anweisungen für die Umsetzung gegeben werden (bei GCC über u.A. über __attribute__((attribute-list)), ab C++11 über [[attribute-list]]. An vielen Stellen gibt die C-Spezifikation ein undefiniertes Verhalten vor (z.B. bei der Rückgabe einer Adresse einer lokalen Variablen). Das Verhalten vom Compiler an diesen Stellen ist compilerabhängig. Einige fügen zusätzlichen Code ein (um z.B. zur Laufzeit einen Laufzeitfehler zu provozieren (Rückgabe der Adresse 0 im obigen Beispiel)), einige tätigen gar nichts. Ergänzend zur C-Spezifikation ergänzen einige Compiler die Spezifikation um zusätzliche Features. GCC beispielsweise ermöglicht es, in einer SWITCH-Anweisung ein CASE über einen Wertebereich zu beschreiben (Case 1 ... 3:).

Auch wenn die Sprache C sauber spezifiziert ist, wird man früher oder später compilerspezifische Eigenschaften nutzen und sich damit an einen Compiler binden. Im Falle dieses Buches ist es der GCC Compiler aus der GNU Compiler Collection.

1 https://de.wikibooks.org/wiki/Programmieren_in_C/C%2B%2B:Vorlage:CompilerExplorerLink

2 https://de.wikibooks.org/wiki/Modul:Programmieren_in_C/C%2B%2B:Vorlage:CompilerExplorerLink

1.7 Sonstiges

Die C-Spezikation ist an einigen Stellen nur sehr schwer zu vermitteln. In diesem Sinne wird an einigen Stellen im Buch von C Spezifikation abgewichen. Dies soll der besseren Lesbarkeit/Verständnis dienen.

2 Einführung/Literatur

2.1 Literatur

Kurze Literaturaufzählung, nach Relevanz sortiert:

- ©ISO/IEC ISO/IEC 9899:TC2 "Programming languages - C"
  Spezifikation der Sprache.
  C11: `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`
  C05x: `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`
  Anmerkung: Schwer verständlich
- Harbison, Samuel P (Steele, Guy L.;)
  C: A reference Manual
  ISBN: 013089592X (pbk)
  Upper Saddle River, N.J. [u.a.] Prentice Hall, 2002
  Anmerkung: Gute Spezifikation der Sprache. Kein Selbstlernbuch.
- Kernighan, Brian W (Ritchie, Dennis M.; Schreiner, Axel T.;)
  Programmieren in C mit dem C-Reference Manual in deutscher Sprache
  ISBN: 3446154973 ((pbk.)=978-3-446-15497-1*hbk)
  ISBN: 013110330X
  URL: `http://zbmath.org/?q=an:0701.68014`
  München [u.a.] Hanser [u.a.],1990
  Anmerkung: Spezifikation (nach für Kernighan Standard) auf Basis von Anwendungsbeispielen
- Wolfgang Sommergut
  Programmieren in C
  URL: `http://c-buch.sommergut.de/index.shtml`

2.2 Spezifikation der Standardbibliotheksfunktionen

Die Spezifikation der Standard Header Dateien und der Standard Library Funktionen sind Bestandteil der C-Spezifikation [C11 7.x] und beanspruchen ~⅓der Seiten. POSIX (Portable Operating System Interface) ist eine für UNIX entwickelte standardisierte Programmierschnittstelle u.A. für Betriebssystemaufrufe. Es beinhaltet und erweitert aber auch die Beschreibung der Standard-C Header Dateien und der Standard-C Library Funktionen. Da die POSIX Beschreibungen in den Manpages von UNIX und verwandten Betriebssystemen enthalten sind, empfehlen sich diese als alternative/ergänzende Beschreibung. Neben der

- detaillierten Funktionsbeschreibungen,
- der Parameterbeschreibung,
- der Beschreibung des Rückgabewertes
- der Fehlerarten

- und der zur Nutzung zu inkludierenden Header-Dateien

enthalten einige dieser Spezifikationen ergänzend Anwendungsbeispiele.

In Unix basierten Systemen ist die Spezifikation Bestandteil der Distribution und kann über die Kommandozeile wie folgt aufgerufen werden (Voraussetzung, gcc ist installiert):

```
>>man strcpy
>>man printf.3
```

Alternativ sind diese im Netz verfügbar. Hier empfiehlt es sich, das Wort 'man' der Suche voranzustellen, so dass in den ersten Suchergebnissen direkt ein Link auf die Manpages enthalten ist. Wenn nur der Funktionsname als Suchbegriff eingegeben wird, wird als Suchergebnis zumeist Anwenderinterpretationen geliefert, die oftmals nur die 'halbe' Wahrheit darstellen oder z.T. falsch sind.

2.3 Sprachenvergleich

Die Programmiersprache C gibt es schon seit 50 Jahren. Nach seiner 'Hoch'-Zeit in den 90er Jahren ist diese Sprache zwar nicht mehr führend, aber immer noch unter den Top 5 Sprachen zu finden:

- IEEE Language Ranging 2019 (Quelle: The top programming languages 2019[1])

  1. Python
  2. Java
  3. C
  4. C++
  5. R
  6. JavaScript
  7. C#
  8. Matlab
  9. Swift
  10. Go

- TIOBe-Index Stand Januar 2023 (Quelle: TIOBe-Index[2])

  1. 16,36% Python
  2. 16,26% C
  3. 12,91% C++
  4. 12,21% Java

1 https://spectrum.ieee.org/the-top-programming-languages-2019
2 https://www.tiobe.com/tiobe-index/

5. 5,73% C#
6. 4,64% Visual Basic
7. 2,87% Java Script
8. 2,50% SQL
9. 1,60% Assembly Language
10. 1,37% PHP

Gründe für die gute Positionierung:

- hohe Ausführungsgeschwindigkeit des compilierten Programms
- kein zusätzlicher Interpreter zur Ausführung notwendig (Compilersprache)
- Systemprogrammierung (Betriebssystem, Gerätetreiber sind zumeist in C/C++ geschrieben)
- eingebetteten Systemen sind in C/C++ geschrieben
- (Java) Interpreter und diverse Java 'Libraries' sind in C geschrieben
- Python Librarys sind in C/C++ geschrieben
- C Compiler sind quasi für alle Betriebssysteme und embedded Systeme verfügbar (auch aufgrund der kleinen Library)
- Energieeffizeint (siehe Heise - Grünes Programmieren in C und Rust[3])

Hinweis:

- Christoffer Lernö fragt sich in seinen Blog, ob/wie C durch eine andere Sprache ersetzt werden kann: The case against a c alternative[4]

2.4 C-Dialekte

1972 wurde C von Brian Kernighan und Dennis Ritchie mit dem Ziel entwickelt, Unix für div. Rechnersysteme (und damit nicht mehr in Assembler geschrieben) verfügbar zu machen. Dazu wurde die Vorgängersprache B unter anderem um Datentypen (einhergehend mit der Bereitstellung dieser durch die w:PDP-11[5], einem quasi Standardcomputer der 70er Jahre) und um die klare Trennung von Integer- und Pointer-Variablen erweitert (siehe Bell-Labs[6]).

Aufbauend auf dieser Grundlage wurde C stetig weiterentwickelt. (siehe w:Varianten der Programmiersprache C[7]):

- K&R C (1978)

  Kein offizieller Standard, jedoch wurde mit dem Buch von Kerningham & Ritchie 1978 (siehe Literatur) die Sprache erstmals zusammenhängend beschrieben, so dass das zugrundeliegende Buch als quasi Standard gilt.

3 https://www.heise.de/news/Gruenes-Programmieren-C-und-Rust-energieeffizient-Python-und-Perl-Schlusslicht-7259319.html
4 https://c3.handmade.network/blog/p/8486-the_case_against_a_c_alternative
5 https://de.wikipedia.org/wiki/PDP-11
6 https://www.bell-labs.com/usr/dmr/www/chist.html
7 https://de.wikipedia.org/wiki/Varianten%20der%20Programmiersprache%20C

Besonderheiten (anhand eines Beispielcodes)

```
func(str)
char * str;    /* 1)  */
{              /* 2) Zunächst Definitionsbereich */
int a;         /* 3) */
    b;         /* 4)  */
               /* 2) Ab hier dann Anweisungsbereich */
b=a+1;
               /* 2) Hier nun keine weitere Variablendefinition erlaubt */
func(4,5,6);   /* 5) */
...
//    1) Datentypdefinition bei Funktionsparameter außerhalb der ()-Klammern
//    2) am Anfang des Blockscopes ist der Definitionsbereich, in welchen
//   die Variablen definiert werden. Dieser endet mit der ersten Anweisung.
//   Ab der ersten Anweisung können keine weiteren Variablen definiert werden.
//    3) Nur wenige Datentypen vorhanden
//    4) Implizites Int, d.h. Variablen/Funktionen ohne vorangestellten
//   Datentyp sind automatisch vom Datentyp int.
//    5) kein Funktionsparametercheck, d.h. eine Funktion konnte mit anderen
//   Parameter aufgerufen werden, als in der Tat benötigt wurden (u.A.
//   aufgrund dieser Tatsache wurde das Tool lint entwickelt, welches
//   eine Prüfung vornimmt)
```

- ANSI-C oder C89/90

  Erste offizielle Spezifikation der Sprache.

  Compiler-Schalter zum Aktivieren dieser Version: -ansi oder -std=c90

  Besonderheiten:

  - Einführung weiterer Datentypen: short, long
  - Funktionsprototypen mit dem sich daraus ergebenden Funktionsparametercheck
  - Einführung weiterer Schlüsselwörter: const, volatile, unsigned
  - Einführung des Präprozessors
  - Datentypdefinition von Funktionsparameter nun in den ()-Klammern
  - uvm.

- C99

  Viele Optimierungen, die sich u.A. im Zuge mit der Einführung von C++ ergeben haben.

  Compiler-Schalter zum Aktivieren dieser Version: -std=c99

  Besonderheiten:

  - for-Schleife erzeugt einen eigenen Block, so dass innerhalb der for-Anweisung eine nur im Anweisungsbereich der for-Schleife lokale Variable definiert werden kann (for(int lauf=0;...)
  - Einführung von Zeilenkommentare (enden automatisch am Zeilenende) '//'
  - Frei platzierbare Definitionen, d.h. Variablen-Definitionen können nun auch zwischen Anweisungen stehen
  - VLA Variable Length Array; Array, dessen Dimension erst zur Laufzeit festgelegt wird
  - Inline Funktionen
  - Designated Initializers (z.B. struct point p={.x=1,.y=2};)
  - Compound Literals (z.B. var=(struct xyz){1,2,3};)

- Neue Datentypen: long long / float _complex / double _complex /...
- Verbot des impliziten int, d.h. Compiler gibt mindestens eine Warning, wenn z.B. eine Funktion ohne Rückgabewert definiert wurde, bzw. der Datentyp eines Übergabeparameters nicht gesetzt wurde

- C11 (Default bei gcc)

  Wenige Änderung zu C99. Enthält im wesentlichen Features, welche Compilerhersteller aufgrund von Anwenderforderungen vorab im Compiler implementiert hatten

  Compiler-Schalter zum Aktivieren dieser Version: -std=c11

  Besonderheiten

  - Unterstützung von Multi-Threading in Form durch Bereitstellung von "thread local storage class specifier" (Sichtbarkeit/Gültitgkeit von Variablen auf Thread Basis) und Bereitstellung von Funktionen zur Verwaltung von Threads, Mutexen, condition variable und atomic Operationen
  - Anonymous Structures und Unions (sinnvoll z.B. für einfacheren Strukturierung von Datentypen)
  - Entfernung der 'unsicheren' gets() Funktion
  - und einige weitere

- C17

  Enthält im Wesentlichen nur Fehlerkorrekturen von C11 und keine neuen Features

  Compiler-Schalter zum Aktivieren dieser Version: -std=c17

- C23

  In Entwicklung. Voraussichtliche Veröffentlichung 2024

  Änderungen: siehe bspw. C23 implications for C libraries[8]

  Besonderheiten

  - Unterstützung des Unicode Zeichensatzes
  - Testfunktionen für Addition/Subtraktion/Multiplikation auf Integerüberlauf
  - nullptr als Ersatz für NULL

In diesem Skript wird vorrangig der Standard C11 genutzt. Die Ergänzungen, welche C17 und spätere Versionen mitbringen, sind nach Ansicht des Autors nur Features. Am grundlegenden Syntax wurden hier nur geringfügige Änderungen vorgenommen.

Hinweise:

- Ergänzend zu diesen Standards erweitern einige Compilerherstellen den Standard um weitere Features. Sofern diese (oftmals nützliches) Features genutzt werden, baut man sich Compiler-Abhängigkeiten in seinen Code ein
- Viele etablierten Programme/Libraries nutzen C89/90 als Standard. So basieren bspw. cURL, SQLite, libxml2 auf diesen Standard. Auch der Linux Kernel bis Version 5.18 (Mai 2022) basiert auf C89

8 `https://htmlpreview.github.io/?https://icube-forge.unistra.fr/icps/c23-library/-/raw/main/README.html`

2.5 C Besonderheiten

2.5.1 C-Syntax

Java, JavaScript, PHP uvm. orientieren sich am Syntax der Programmiersprache C. Eine Einführung in den allgemeinen Syntax entfällt somit. Einzig werden hier einige Besonderheiten dargestellt:

- Im C-Syntax gibt es nur wenige Ausnahmeregeln, so dass quasi alles an allen Stellen erlaubt ist:

```
//Aufruf einer Funktion im Funktionsparameterbereich
int main(int argc,char *argv[]) {
  if( ({int var=1; var>argc;}))  //Hier wird eine Variable
                                 //innerhalb des IF-Konstruktes angelegt!
  switch(argc) {
        int abc;               //Variable außerhalb des Case-Bereiches
     case 1:
        int var1=8;
        hallo:                 //Sprungziel innerhalb von Switch
     case 2:
        var1*=2;               //Vorsicht, in diesem Case-Zweig ist var1 nicht
        break;                 //initialisieirt
    (void)abc;                 //Zur Unterbildung der Compilerwarning
  }
  else
    goto hallo;      //Sprung zu einer Stelle innerhalb der Switch-Anweisung

  for(int lauf1=7,lauf2=8;lauf2++,lauf1--;printf("-"));
  return 0;
}
```

- Die C-Spezifikation ist so ausgelegt, dass der C-Code in kompakten/schnellen Maschinencode übersetzt werden kann und die Prozessoreigenschaften optimal genutzt werden. Dementsprechend sind in der C-Spezfikation diverse 'undefined/unspecified Behavior' vorhanden. Beispiele sind:
  - Bitbreite des Datentyp integer ist vom Prozessor und Compiler abhängig
  - Verhalten bei Integerüberlauf/unterlauf ist von der Prozessorarchitektur abhängig
  - Auswertereihenfolge der Übergabeparameter beim Funktionsaufruf ist nicht definiert

  Bei Nichtbeachtung kann dies zu nicht portablen (auf andere Compiler oder andere Prozessoren) oder gar fehlerhaften Programmen führen (siehe auch: Undefined Behavior in C and C++[9])

- Einige Operatoren haben in Abhängigkeit der Verwendung unterschiedliche Bedeutungen

  '*' -> Ganzzahl Multiplikation, wenn beide Operanden vom Datentyp Ganzzahl sind

  '*' -> Gleitpunktzahl Multiplikation, wenn eine der beiden Operanden vom Datentyp Gleitpunktzahl und der andere von Ganzzahl ist

  '*' -> Datentyp Zeiger anlegen, wenn der linke Operand ein Datentyp ist

  '*' -> Dereferenzierung, wenn der rechte Operand vom Datentyp Zeiger ist

9 https://medium.com/@pauljlucas/undefined-behavior-in-c-and-c-f30844f20e2a

- Ausführbarer Code in der Parameterliste von Funktionen

```
int main(int argc,char *argv[printf("%*c",argc,'a')])
```

Folglich sind viele Anweisungen vom Syntax korrekt (so dass der Compiler keine Fehler ausgibt):

- führen andere Aktionen aus, als gedacht
- werden zur Verschleierung des Source-Codes genutzt (Unleserlicher Code)
- werden von Profi-Programmierer angewendet, um kleinere/schnellere Programme zu schreiben

2.5.2 C-Fallstricke

Das Ziel der Spezifikation von C war eine universelle/schnelle Programmiersprache (als Ersatz für Assembler und den komplexen Sprachen wie Algol und Fortran). Aus Geschwindigkeitsaspekten sind einige Anweisungen von der zugrundeliegenden Rechnerarchitektur abhängig, so dass C nicht portabel ist. Ebenso werden aus Geschwindigkeitsgründen keine unbedingt notwendigen Überprüfungen zur Laufzeit durchgeführt:

- Die Datentypen sind nicht klar definiert (Datentyp int ist 16- oder 32-Bit breit, abhängig von der Rechenbreite der CPU)
- C kann den gesamten vom Prozessor adressierbaren Speicherbereich für Variablen und Programm nutzen, d.h. Breite des Datentyps Zeigers ist von der Rechenbreite der CPU abhängig
- Keine Prüfung der Indices bei Arrayzugriffen

```
char arr[10];
arr[1]='a';
arr[10]='z';   //Korrekt
arr[-1]='x';   //Korrekt
```

- Keine Prüfung, ob bei der Dereferenzierung von Pointer diese auf eine gültige Speicheradresse zeigen

```
int  vari=3;
int *ptr=&vari;
vari=*++ptr;      //Zugriff auf falsche Variable
ptr=(int *)100;   //Initialisierung des Pointers mit absoluter Adresse
```

- Implizite Typkonvertierung bei Ganzzahl und Gleitkommazahlen

```
char  varc;
short vars;
int   varii;
float varf=sin(varc*vars)+varii;
//entspricht:
//float varf=(float)(sin((double)((int)varc*(int)Vars))+(double)varii);
```

- C stellt dynamischen Speicher (Heap) zur Verfügung. Der Anwender ist sowohl für die Reservierung als auch für die Freigabe des zuvor reservierten Speichers zuständig

2.5.3 Laufzeitfehler

Neben Syntax-Fehler (die durch den Compiler zur Compilezeit ausgegeben werden) gibt es insb. in C bei unsachgemäßer Programmierung viele Laufzeitfehler. Diese führen:

- zum Programmabsturz während der Laufzeit
- zu einem merkwürdigen Verhalten während der Ausführung des Programms (Variable hat auf einmal einen anderen Wert, als erwartet)

Dies liegt unter anderen an folgenden Sachverhalten:

- Keine Prüfung der Indices bei Arrayzugriffen
- Keine Prüfung, ob bei der Dereferenzierung von Pointer dieser auf eine gültige Speicheradresse zeigt.
- Keine Prüfung, ob bei der Allokation von lokalen Variablen genügend Speicher (Stack) zur Verfügung steht.
- Zahlenüberlauf bei Integer-Arithmetik ist nicht definiert (siehe 'Integer-Arithmetik')

```
         int a=2000000000;
unsigned int b=1000000000;
         int c=a+b;        //c=-1294967296
```

- Division durch 0, welche nicht durch eine Exception abgefangen wird
- C-Spezifikation an einige Stellen ein undefiniertes Verhalten beschreibt, also dem Compilerhersteller die Umsetzung der Anweisung in Code frei lässt

```
int lauf=5;
lauf=++lauf * ++lauf + ++lauf * ++lauf;
//GCC-Compiler -> lauf=130
//Clang-Compiler -> lauf=114
```

Die ersten beiden Fehler werden als Pufferüberlauf/BufferOverflow, die fehlende Stackprüfung als Stapelüberlauf/StackOverflow bezeichnet.

Laufzeitfehler sind für den ungeübten Programmierer schnell erstellt, jedoch nur schwer zu finden. **Vorsorge (also die Analyse des Codes) ist hier die bessere Variante als Nachsorge (den Fehler erst beim Auftreten zu suchen)**. Insbesondere bei Nutzung von Pointern sollte der Source-Code vorm Übersetzen auf bspw. folgende Sachverhalten geprüft werden:

- wurde der Pointer vor der Dereferenzierung mit einer gültigen Speicheradresse initialisiert
- ist der Speicherbereich, auf den der Pointer zeigt, gültig
- beinhaltet der Zeiger den Wert NULL, welcher als Fehlerfall genutzt wird
- zeigt der Zeiger nach einer Änderung (z.B. durch Zeigerarithmetik) weiterhin auf eine gültige Adresse

Ähnliche Sachverhalte gilt es bei Nutzung von Arrays zu prüfen

- liegt der Indice innerhalb der Array Grenzen
- ist sichergestellt, dass bei Übergabe eines Arrays an eine Funktion (sowohl selbstgeschrieben Funktion als auch Libraryfunktionen) diese nicht über die Grenzen des Arrays auf das Array zugreift

Ergänzend zu oben aufgeführten empfehlenswerten Codereviews empfehlen sich folgende Ansätze/Hilfsmittel zum finden/vermeiden von Laufzeitfehlern:

- Bewusste Nutzung des C Syntax. So können mit dem Schlüsselwort 'const' die Zeiger in ihrer Verwendung eingeschränkt werden (wird leider nur selten genutzt)
- Erweiterte Syntaxprüfung des Compiler nutzen, z.B. durch den Compiler-Schalter: '-Wall -Werror'
- Nutzung externer Codeanalysetools (wie z.B. lint) welche eine tiefergehende Codeanalyse als der Compiler durchführen und bei welcher einige C Funktionalitäten 'deaktiviert' werden können
- Nutzung des Code sanitizer[10] (-fsanitize=address), welcher im erzeugten Maschinencode zusätzliche Testfunktionen einbaut und Canarie zwischen den einzelnen Variablen legt, so dass Bufferüberläufe und 'Use-after-Free' Zugriffe durch Prüfung der Canarie erkannt werden
- Nutzung von Valgrind[11], welcher den auszuführenden Maschinencode in einen anderen Zwischencode übersetzt und diesen unter seiner Kontrolle in einer virtuellen Maschine ausführt (Verwendung von Valgrind siehe bspw.: Valgrind memcheck: Different Ways to lose your memory[12])

Hinweis:

- Ausgaben auf die Standardausgabe (bspw. über printf()) sind gepuffert, d.h. sie werden nicht unmittelbar zur Anzeige gebracht. Stürzt das Programm aufgrund eines Laufzeitfehlers ab, so wird dieser Puffer nicht abschließend zur Anzeige gebracht. Im Falle von Laufzeitfehler empfiehlt sich das regelmäßige flushen des Puffers mittels 'fflush(stdout)' resp. der Nutzung der Standfehlerausgabe mittels 'fprintf(stderr,”Fehler”)' oder 'perror(”Fehler”)'

2.5.4 Zusammenfassung

C wurde mit dem Ziel entwickelt, als höhere Programmiersprache zum Erstellen eines Betriebssystems zu dienen. Schnelle Programmausführung stand somit im Vordergrund. Die fehlenden Prüfungen bei bspw. Arrayzugriffen und Pointerdereferenzierung sind gewollt. C geht davon aus, dass der Programmierer weis, was er tut. Wenn der Programmierer bestimmte Sachverhalte während der Laufzeit nicht sicherstellen kann, so muss die notwendige Überprüfung durch zusätzlichen Code händisch sichergestellt werden!

Wie gesagt ist C ein offener Syntax, welcher nur wenige Einschränkungen kennt und den Syntax ohne Schnick-Schnack in Maschinencode umsetzt. Das heißt nicht, dass C eine schlechte

10 https://de.wikipedia.org/wiki/lang%3Den
11 https://de.wikipedia.org/wiki/lang%3Den
12 https://developers.redhat.com/blog/2021/04/23/valgrind-memcheck-different-ways-to-lose-your-memory#

Programmiersprache ist, sondern dass vorausgesetzt wird, dass der Programmierer weiß, was er tut:

- Er kennt die Wertebereiche der unterschiedlichen Datentypen (Plattformabhängig) und kennt die impliziten Typkonvertierungsregeln, so dass keine Typverletzungen auftreten
- Er kennt die Dimension des Arrays und stellt sicher, dass kein Zugriff auf ein Arrayelement außerhalb der Dimension erfolgt.
- Wenn er Speicher im Heap reserviert, gibt er diesen nach Verwendung wieder frei.
- ...

In vielen neueren Sprachen wurde der Ansatz "Alles ist überall erlaubt / Der Programmierer weiß was er tut" eingeschränkt, indem:

- der Syntax eingeschränkt wurde
- zur Laufzeit Überprüfung stattfinden und ggf. eine Ausnahmebehandlung im Fehlerfall ausgeführt wird
- eine komfortable Speicherverwaltung (Garbage Collector) integriert wurde

Die Programmiersprache C fordert vom Programmierer somit mehr Bewusstsein für die Sprache ab, als andere Programmiersprachen. Wer der Sprache C mächtig ist, wird sicherlich auch andere Sprachen bewusster nutzen!

Hinweis

- Im Blog vom Tom Mewet[13] sind einige wesentlichen Unterscheidungen von C zu anderen Sprachen dargestellt

2.6 Compiler Sprache

C ist eine Compiler-Sprache. Aus dem Source-Code wird durch die Toolchain, bestehend aus Compiler und Linker ein vom Prozessor direkt ausführbarer Maschinencode erzeugt.

Im Wikipedia Artikel Compiler[14] wird die Aufgabe eines Compilers wie folgt zusammengefasst: "*Ein Compiler (auch Kompilierer; von englisch compile 'zusammentragen' bzw. lateinisch compilare 'aufhäufen') ist ein Computerprogramm, das Quellcodes einer bestimmten Programmiersprache in eine Form übersetzt, die von einem Computer (direkter) ausgeführt werden kann.*"

Beispiel eines Übersetzungsvorganges:

C-Code

```
int a=7,b=8;
a=a+b+7;
```

Assemblercode (etwas vereinfacht dargestellt)

X86-64	ARM	6502	Anmerkungen

13 https://tmewett.com/c-tips/
14 https://de.wikipedia.org/wiki/lang%3Den

X86-64	ARM	6502	Anmerkungen

`MOV edx,DWORD PTR a[rip]`
`MOV eax,DWORD PTR b[rip]`
	`LDR r1,.LCPI0_0`
	`LDR r0,[r1]`
	`LDR r2,.LCP0_1`
	`LDR r2,[r2]`
		`LDA _a`
		`LDX _a+1`
		`JSR pushax`
		`LDA _b`
		`LDX _b+1`
			Lade Inhalt der Variablen in die Register

`ADD eax,edx`	`ADD r0,r0,r2`	`JSR tosaddax`	Addiere die Variablen
`ADD eax,7`	`ADD r0,r0,#7`	`JSR incax7`	Addiere die Konstante 7
`MOV DWORD PTR a[rip],eax`	`STR r0,[r1]`	`STA _a` `STX _a+1`	Speichere Register zurück in die Variablen

Der vom Compiler erzeugte Maschinencode liegt letztendlich als Zahlencode im Speicher (hier nicht dargestellt) und wird vom Prozessor Befehlsweise eingelesen und direkt ausgeführt. Der Start des erzeugten Maschinencode, die Rückkehr zum Betriebssystem und der Aufruf von Library-Funktionen (z.B. printf) bedingen ergänzende Abhängigkeiten. Für jeden Prozessor und jedes Betriebssystem ist folglich ein gesonderter Compiler notwendig.

2.6.1 Arbeitsweise eines Compilers

Der Wikipedia Artikel w:Compiler[15] beschreibt die prinzipiellen Schritte bei der Übersetzung eines Quellcodes in einen Ziel Code wie folgt:

Syntaxprüfung

es wird geprüft, ob der Quellcode ein gültiges Programm darstellt, also der Syntax der Quellsprache entspricht. Festgestellte Fehler werden protokolliert. Ergebnis ist eine Zwischendarstellung des Quellcodes

Analyse und Optimierung

Die Zwischendarstellung wird analysiert und optimiert. Dieser Schritt variiert im Umfang je nach Compiler und Benutzereinstellung stark. Er reicht von einfacheren Effizienzoptimierungen bis hin zu Programmanalyse

Codeerzeugung

Die optimierte Zwischendarstellung wird in entsprechende Befehle der Zielsprache übersetzt. Hierbei können weitere, zielsprachenspezifische Optimierungen vorgenommen werden

Diese Schritte können sowohl unabhängig voneinander durchlaufen werden (Multi-pass-Compiler), als auch in Einem (Single-pass-Compiler). Bei Letzteren wird der Quellcode nur

15 `https://de.wikipedia.org/wiki/Compiler`

einmalig von vorne bis hinten analysiert und zugleich der Maschinencode erzeugt. Trifft der Compiler auf einen Variable/Funktion/Datentyp, welche zuvor nicht definiert wurde, kann der Compiler hierfür keinen Maschinencode erzeugen. Der Code darf somit keine Vorwärtsbezüge enthalten.

Multi-pass-Compiler durchlaufen den Code mehrmals, so dass in einem am Anfang liegenden Durchlauf zunächst der Code nach Definitionen durchsucht wird. Die eigentliche Codeerzeugung erfolgt dann in einem späteren Durchlauf.

Die Sprache C entstammt aus einer Zeit, indem Speicher ein knappes Gut war. Typische Hauptspeichergrößen waren 32kByte...128kByte, Festplattenspeicher ca. 5MByte. Die Spezifikation der Sprache ging aufgrund dieser knappen Ressourcen von einem Single-pass-Compiler als Werkzeug aus. Daraus ergibt sich, dass in C alle Variablen/Funktionen/Datentypen vor der ersten Nutzung definiert oder per Prototyp/(Forward)deklaration beschrieben werden müssen.

2.6.2 Compileroptimierungen

Neben der reinen Übersetzung unterstützen moderne Compiler div. Optimierungsmöglichkeiten wie z.B.:

- Halten von Variablen in Register
- Erkennung ungenutzter Variablen
- Optimierung von Schleifen
- Erkennung und Optimierung von 'doppelten' Anweisungen

Optimierungen versuchen, das erstellte Programm hinsichtlich der Anzahl der Maschinenbefehle (kleineren Code) oder der Ausführungsgeschwindigkeit der Befehle (schnelleren Code) zu optimieren, ändern aber nichts am Source-Code. Hohe Optimierungseinstellungen können aufgrund der Zwischenspeicherung von Variablen in Registern und der Optimierung der Ausführungsreihenfolge den erzeugten Maschinencode so entfremden, dass dieser im Debugger nicht mehr nachvollziehbar angezeigt werden kann.

Daher wird in C-Projekten zwischen den Debug-Mode und dem Release-Mode unterschieden. Im ersten sind viele Optimierungen ausgeschaltet und ggf. ergänzende printf() Ausgaben enthalten, so dass im Debugger die Codeabarbeitung verfolgt werden kann. Im Release-Mode wird der Compiler auf 'max' Optimierung gestellt. Ein Debuggen ist zwar auch hier möglich, aber der Zusammenhang zwischen Maschinencode und Source-Code nur schwer nachvollziehbar.

Die Optimierungsmöglichkeiten eines Compilers sind zwar gut, der Compiler kann aber nur das optimieren, was der Programmierer 'vorgibt'. Für einen optimalen Code ist folglich auch der Programmierer verantwortlich:

- Stringvergleich dauert immer länger als ein Zahlenvergleich, d.h. bspw. keine Zustandsinformation im Datentyp String speichern (zustand=”ON”) sondern als Zahlenwert (zustand=1)
- Mehrmalige Aufrufe einer Funktion mit identischen Übergabeparametern sind zu vermeiden

```
for(index=0; index<strlen(str);  index++)
//die Funktion strlen() wird bei jedem Schleifendurchlauf aufgerufen
```

- Mathematische Ausdrücke, sofern möglich in Ganzzahlarithmetik durchführen
- IF-Abfragen vermeiden (Sprünge in Maschinensprache gehören zu den langsamsten Maschinenbefehlen)
- Schleifen von oben nach unten zählen lassen (erspart ein Vergleich und damit einen Sprung) oder alternativ zur Zählvariable einen Pointer nutzen

```
for(int lauf=0; lauf<10;lauf++)  //Ein Vergleich für < und
                                 //ein Vergleich für True/False notwendig
for(int lauf=10;lauf;lauf--)     //Nur Vergleich auf True/False notwendig
char str[]="hallo";
for(const char *ptr=str;*ptr;ptr++) //Vermeidung der Index-Variable
```

- Durchsuchen eines zweidimensionalen Arrays in der inneren Schleife über den rechten Index und nicht den linken Index (Optimierung, so dass der Cache optimal genutzt wird)

Schlecht

```
int zeile,spalte;
for(spalte=999;spalte;spalte--
)
   for(zeile=999;zeile;zeile--
)
      if(matr[zeile][spalte]>max)
           max=matr[zeile][spalte];
```

Gut

```
int zeile,spalte;
for(zeile=999;zeile;zeile--
)
   for(spalte=999;spalte;spalte--
)
      if(matr[zeile][spalte]>max)
           max=matr[zeile][spalte];
```

2.7 Zusammenspiel von C und Betriebssystem

C wurde als Sprache zum Schreiben eines Betriebssystems (Unix) entwickelt, zwecks einer effizienteren Programmierung und einer Portierbarkeit (beides hat dann Unix zum Durchbruch verholfen). Die Sprache ist somit eng mit dem Unix-Betriebssystem (und anderen Betriebssystemen, die entsprechende Eigenschaften adaptiert haben) verbunden. Dies ist unter anderen daran erkennbar, dass:

- Alles ist eine Datei, d.h.
  - Standardein-, Standardaus- und Standardfehlerausgabe erfolgt über Lesen/Schreiben in eine Datei, welche mit dem Start des Programms geöffnet sind (resp. vom startenden Programm vererbt werden). printf(), scanf() und error() nutzen diese Dateien als Arbeitsgrundlage.
  - Ein Großteil der Interaktion mit dem Betriebssystem erfolgt über Dateizugriffe (Zugriff auf Geräte, Zugriff auf Prozessorauslastung, ...)

- Start des Programms erfolgt über den Funktionsaufruf main(), welcher Übergabeparameter (als Array von Strings) übergeben werden und welche einen Fehlerstatus (als Integer) an den Aufrufer zurückgibt.
- Die Nutzung von Environment-Variablen zum Datenaustausch zwischen Betriebssystem und Anwendungsprogramm (Länder-/Spracheinstellung, Systemkonfigurationen, ...)
- Klare Trennung zwischen OS-Funktionalität und C-Funktionalität:
  - OS-Funktionen gehören nicht zur C-Standardbibliothek. D.h. OS-Funktionen werden direkt, ohne Nutzung von Wrapper, aufgerufen (z.B. wird zur Erzeugung eines Threads direkt die dazugehörige OS-Funktionalität pthread_create() genutzt)
  - Die Standard-C-Library dementsprechend nur wenige Funktionen bereitstellt
- Libraries / DLL beruhen auf den CallingConvention von C (Regelung, wie bspw. die Übergabeparameter an Funktionen übergeben werden) und sind folglich rudimentär C-Funktionen
- Linker (ein Programm zu binden mehrere Objekt-Dateien zu einer Objekt-Datei) ist eine Funktionalität des Betriebssystems (und nicht des Compilers). Das Ausgabeformat des Linkers ist wahlweise:
  - eine ausführbare Datei
  - eine statische Library (Archiv)
  - eine dynamische Library

  d.h. dynamische Librarys können mit diversen Compiler-Sprache erstellt werden und auch von diversen Programmiersprachen (inkl. Interpretersprachen wie Python) eingebunden werden
- Die Standardbibliotheksfunktionen von C sind in Librarys ausgelagert. printf() / strlen() sind bspw. in der Library libc.so und sin() / sqrt() in der Library libm.so enthalten. Werden aus einem Programm diese Funktionen genutzt, so müssen diese Librarys über Linkerschalter eingebunden werden (Ausnahme libc.so, welche per Default eingebunden wird)

2.8 Alles ist Speicher

Ein Rechnersystem besteht im Wesentlichen aus einem Prozessor/CPU (bestehend aus Arithmetische/Logischen Einheit (ALU), den Registern und dem Steuerwerk) und aus Speicher. Das Zusammenspiel beider Einheiten lässt sich wie folgt darstellen:

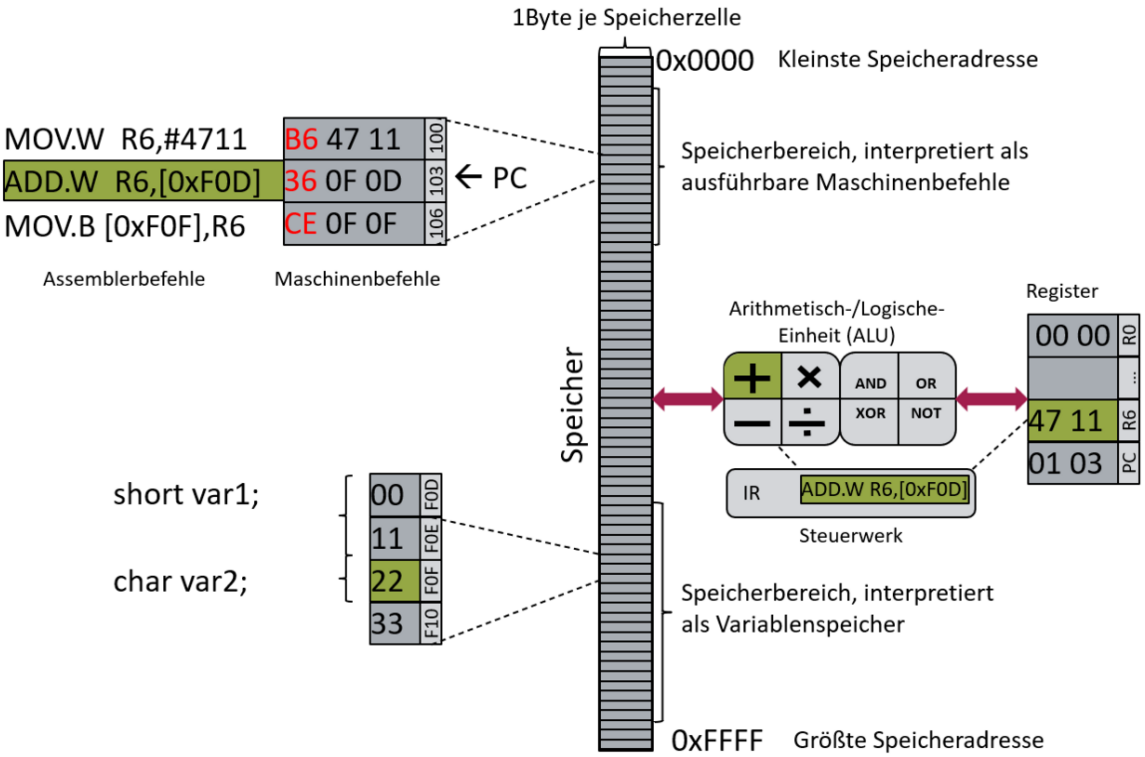

Abb. 1 Interpretation des Speicherinhaltes als Variablenspeicher und Programmspeicher

Die Aufgabe der CPU besteht darin, die Maschinenbefehle aus dem Speicher zu laden (adressiert über den ProgrammCounter PC) und diese einen nach dem anderen auszuführen. Die zu verarbeitenden Daten (Variablen) sind ebenfalls im Speicher enthalten (angesprochen über die Speicheradresse z.B. 0xf0f, 0xf0d).

Ein Speicher ist eine Einheit zur Speicherung von Informationen. Im weitesten Sinne entspricht der Hauptspeicher (nicht Festplatte, resp. sekundärer Speicher) einem Karteikarten System:

- Wobei auf einer Karteikarte nur 8-Bit/1-Byte Information gespeichert sind
  - Vorzeichenlose Zahlenwerte von 0...255
  - Vorzeichenbehaftete Zahlenwerte von -128...+127
  - ASCII Zeichen
  - Maschinenanweisungen für den CPU
- Max. Anzahl der Karteikarten hängt von der Adressierungsbreite (zumeist identisch zur Rechenbreite) des Prozessors ab:
  - Bei einem 32-Bit System sind 2^{32}=4 294 967 295 Karteikarten vorhanden
  - Bei einem 16-Bit System sind 2^{16}=65 536 Karteikarten vorhanden
  - Bei einem 64-Bit System sind 2^{64}=18 446 744 073 709 551 616 =18 Exa vorhanden
- Bindeglied zwischen Prozessor und Speicher ist das Bus-System, welches sich in einen Daten- und einen Adressbus unterteilen lässt. Der Adressbus dient dabei der Adressie-

rung einer Speicherstelle und der Datenbus zum Datentransport in die Speicherstelle (Schreiben) oder aus der Speicherstelle (Lesen).
- Es kann zu einem Zeitpunkt nur einer (zumeist die CPU) durch Vorgabe einer Adresse die Daten (Inhalt der Karteikarten) lesen oder schreiben

  - Beim Lesen wird der gesamte Inhalt der Karteikarte zurückgegeben
  - Beim Schreiben wird der gesamte Inhalt der Karteikarte neu beschrieben (Änderung einzelner Bits bedingen das vorherige Lesen des gesamten Inhaltes (Var=Var|0b0100)
  - Es gibt kein Löschen, die einzelnen Bits des Speichers sind entweder 0 oder 1

- Inhalte des Speichers hängen von der Interpretation (des Programms) ab

  - Maschinenbefehle, welches die CPU ausführt
  - Variablen als 8-Bit/16-Bit/32-Bit/Ascii/Float/Strutur/...
  - Heap, zur Darstellung von zur Laufzeit benötigten Speicher
  - Stack, zur Darstellung von lokalen Variablen

- Der Inhalt des Speichers kann

  - Flüchtig sein, d.h. er verliert nach PowerOff seinen Inhalt und ist nach PowerOn mit einem zufälligen Wert gefüllt
  - Nicht flüchtig sein, d.h. er behält auch ohne Spannungsversorgung seinen Inhalt
  - Nur lesbar sein (für Konstanten)
  - Schreib- und lesbar sein (für Variablen)
  - Nur ausführbar sein (für Funktionen)

Zusammengefasst kann gesagt werden: "Alles ist Speicher". Jede Variable belegt somit in Abhängigkeit des Datentyps Speicher, jede Funktion belegt in Abhängigkeit der dazugehörigen Anweisungen Speicher und selbst der Zugriff auf die Peripherie zur Interaktion mit der Umwelt erfolgt über Speicherzugriffe. Variablen / Funktionen / Peripherie werden folglich über Speicheradressen angesprochen, d.h. :

- Ein Zugriff auf eine Variable erfolgt über deren (Start)Speicheradresse, wobei dann in Abhängigkeit des Datentyps beginnend ab der Speicheradresse ein oder mehrere Bytes gelesen werden
- Ein Aufruf einer Funktion entspricht einem Sprung zu einer Speicheradresse, ab welcher der nächste Maschinenbefehl zur Ausführung gebracht werden soll

In C wird dieses Konzept direkt aufgegriffen, indem:

- Die Speicheradresse in einer Zeiger(variablen) gespeichert werden kann
- Über den Adressoperator zu jeder Variablen/Funktion die Speicheradresse ermittelt werden kann und z.B. in einer Zeigervariablen gespeichert werden kann
- Über Dereferenzierung auf jede Speicheradresse lesend und schreibendend zugegriffen werden kann

Alle Programmiersprachen basieren auf diesem Prinzip. Die Sprache C gibt einen Teil der Verantwortung für den Umgang mit Speicher an den Programmierer ab. Andere Sprachen versuchen, diesen Sachverhalt möglichst vor dem Programmierer zu verstecken. Dies lässt sich am Beispiel der Symboltabelle erkennen. In C werden durch den Compiler/Linker alle Variablen/Funktionssymbole direkt durch die Speicheradresse ersetzt, so dass die Symboltabelle im ausführenden Code nicht mehr enthalten ist. Dynamischer Speicher wird vom

Programmierer und nicht vom Compiler verwaltet. D.h. der Programmierer kann sich Speicher anfordern, die Verwaltung des Speichers (Zuordnung zu den Attributen/Inhalten) liegt jedoch in der Verantwortung des Programmiers.

3 Toolchain/Entwicklungsumgebung

3.1 Toolchain

Die C-Toolchain besteht im Wesentlichen aus dem Compiler, den Linker und dem Loader:

- Compiler zur Übersetzung der einzelnen C-Dateien in Objekt-Dateien
- Linker (OS-Funktionalität) zum Binden mehrerer Objekt-Dateien zu einer (ausführbaren) Executable-Datei
- Loader (OS-Funktionalität) welche die Executable-Datei vom sekundären Speichermedium in den Speicher des Prozess lädt und das Programm durch einen Sprung zu main() zur Ausführung bringt (inkl. Bereitstellung der Übergabeparameter)

→ Die C-Toolchain und das Betriebssystem sind folglich aufeinander abgestimmt

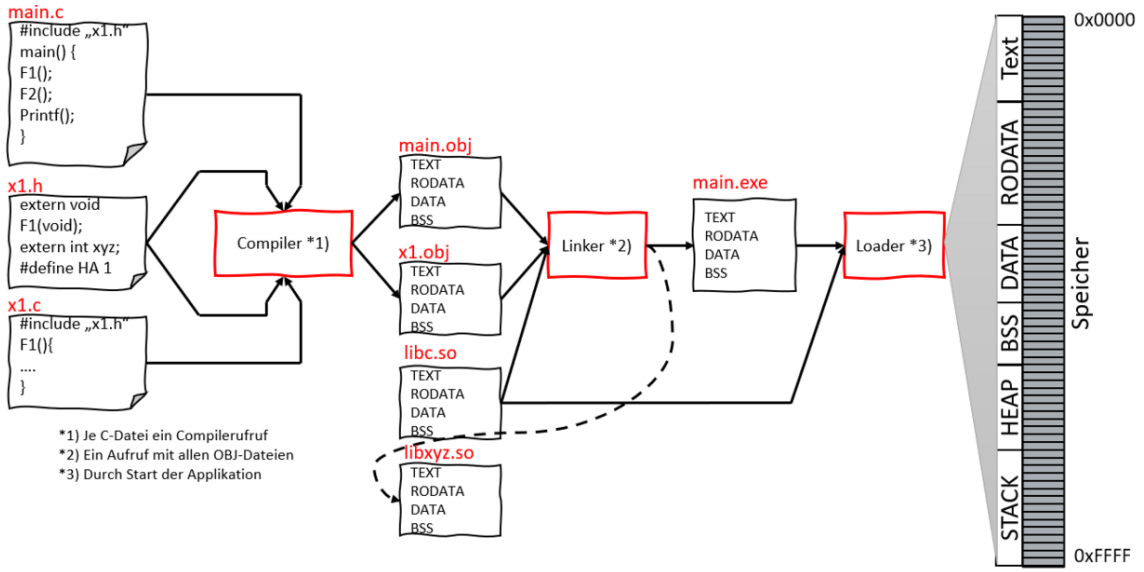

Abb. 2 Darstellung der C/C++ Toolchain bestehend aus Compiler, Linker und Loader

Die Koordination der Tools kann per Hand/Manuell, per Make oder über eine integrierte Entwicklungsumgebung (Eclipse, Visual Studio, XCODE, IntelliJ IDEA, ...) erfolgen.

3.1.1 Compiler

C-Dateien werden einzeln compiliert. Jeder Compilerdurchlauf erzeugt eine Objektdatei. Vor der eigentlichen C-Compilierung wird erst der C-Präprozessor ausgeführt, welcher unter anderen die Include-Anweisungen durch den Inhalt der dazugehörigen Datei ersetzt. Die vom Compiler erzeugte Objekt-Datei ist einzeln nicht ausführbar, da:

- Funktionen aufgerufen werden, welche in anderen C-Dateien definiert sind
- Auf globale Variablen zugegriffen werden, die in anderen C-Dateien definiert sind
- Ggf. keine main-Funktion enthalten ist, sondern nur Funktionen und globale Variablen enthalten sind, welche von anderen C-Datei genutzt werden
- Library-Funktionen verwendet werden (z.B. printf()), die nicht Bestandteil des C-Projektes sind

Compiler-Varianten

Für die Sprache C sind diverse Compiler verfügbar:

- Intel-Compiler (ICC)
- Microsoft Compiler (MSVC)
- GCC Compiler (GCC)
- Clang Compiler Frontend (CLANG) mit Compiler-System LLVM
- ...

Unter Linux ist der GCC-Compiler der Standardcompiler. Alle C-Programme und das Betriebssystem selbst wurden mit diesem Compiler erstellt. Sofern dieser nicht defaultmäßig auf dem System installiert ist, kann er mit wenigen Handgriffen schnell nachinstalliert werden. Auf Windows-Systemen gibt es keinen Standardcompiler. Oft wird hier der Microsoft Compiler genutzt. Soll besonders schneller Code für Intel Prozessoren (und nicht AMD) erzeugt werden, empfiehlt sich der Intel-Compiler. Unter macOS wird sowohl sowohl CLANG/LLVM und GCC eingesetzt.

Compiler-Parameter

Neben der Angabe der zu übersetzenden C-Datei können beim Aufruf des Compilers weitere, optionale Parameter gesetzt werden, über welchen der Compiler konfiguriert wird. Am Beispiel des GCC-Compilers sind u.A. folgende Schalter vorhanden:

- Optimierungsqualität setzen (z.B. -Ox mit x=0 → keine Optimierung x=1 → einfache Optimierung x=2 → bessere Optimierung x=3 → höchste Optimierung x=size → Optimierung hinsichtlich Programmgröße)
- Makros (Präprozessoranweisung) setzen (z.B -Dmakro=7)
- Umfang der Fehlerprüfung (Warnings) und Umgang mit diesen setzen (-Wx mit x=all → erhöhter Warning Level x=extra → zusätzlicher Warning Level, die nicht in -Wall enthalten sind x=error → Warnings als Fehler ansehen)
- einzubindende Librarys setzen (-lx mit x=c → zum Einbinden der Standard-C-Library (defaultmäßig aktiviert) x=m → zum Einbinden der Math-Library)
- Suchpfade für Header Dateien setzen (-Idir mit dir=Verzeichnis in dem ergänzend nach Header-Dateien gesucht wird)
- Suchpfade für Library Dateien setzen (-Ldir mit dir=Verzeichnis, in dem ergänzend nach Library-Dateien gesucht wird)
- Instrumentierungskommandos (Erzeugung von zusätzlichen Maschinencode zur erweiterten Fehlerprüfung) (z.B. -fsanitize=x x=address zum Instrumentieren des Executable, so dass einige illegale Speicherzugriffe erkannt werden)
- ...

3.1.2 OBJ-Datei

Eine Objektdatei speichert die vom Compiler (eigentlich die vom Assembler) übersetzten Anweisungen in einer strukturierten Form. Die Maschinenbefehle werden in der Text-Section gespeichert. Globale und statisch lokale Variablen, welche außerhalb eines Funktionskontextes existieren, werden entweder in der Data-Section (bei initialisierten Variablen) oder in der Bss-Section (bei nicht initialisierte Variablen) gespeichert. Erstere Section beinhaltet die Initialisierungswerte der Variablen. Zweitere Section wird mit Programmstart mit 0 gefüllt, so dass alle nicht initialisierten globalen Variablen den Wert 0 beinhalten.

Globale Konstanten werden in der rodata-Section gespeichert. Dies ermöglicht es dem Loader, die Zugriffsrechte auf diesen Speicherbereich auf Read-Only zu setzen.

Alle Variablen und Funktionsnamen werden mit ihren zugeordneten Speicheradressen in der Symboltabelle gespeichert.

Die absolute/tatsächliche Speicheradresse wird erst beim Erstellen der Executable erzeugt. Die Sectionen, welche die bis dahin relativen/verschiebbaren Adressen beinhaltet werden hierbei in Segmente (beinhaltet absolute Speicheradressen) gewandelt.

Typische Dateiformat für Objekt-Dateien sind Executable and Linking Format (ELF)[1] und Common Object File Format (COFF)[2].

3.1.3 Linker

Die einzelnen Objekt-Dateien werden vom Linker zu einer Objektdatei zusammengefügt, d.h. alle Funktionen und globale Variablen werden in einer Datei zusammengebunden, so dass bisher 'externe' Funktionen und 'externe' globale Variablen für alle im Zugriff stehen. Auch Libraries werden 'eingebunden', so dass z.B. printf()/sin() ebenfalls im Zugriff stehen.

Die Ausgabedatei eines Linkerdurchlaufes kann wahlweise eine **Objekt-, eine Library- oder eine Executable-Datei** sein. Erstes bedeutet, dass mit einem Linker Aufrufe nicht alle Objekt-Dateien auf einmal gebunden werden müssen, sondern dies in einen mehrstufigen Prozess (z.B. Ordnerweise, Klassenweise) erfolgen kann. Mit dem letzten Linker Aufruf im mehrstufigen Prozess wird entschieden, ob eine Library (enthält keine main()-Funktion) oder eine Executable (bei welche alle globalen Variablen, alle statisch lokalen Variablen und alle Funktionen eine absolute Speicheradresse zugewiesen bekommen) erzeugt werden soll.

1 `https://de.wikipedia.org/wiki/Executable_and_Linking_Format%20`
2 `https://de.wikipedia.org/wiki/Common_Object_File_Format`

3.1.4 Loader

Mit dem Start der Anwendung wird der w:Loader[3] aufgerufen (bestehend aus der OS-Funktion exec() (Posix)) , welcher die aktuelle laufende Anwendung durch die zu startende Anwendung ersetzt. Hierbei werden die Inhalte der Segmente aus der Objekt-Datei in die dazugehörigen Speicheradressen des Prozesses geladen. Im Anschluss wird main() (resp. C-Startup, welche dann Main startet) aufrufen. Die Übergabeparameter von exec() werden an Main() weitergereicht.

3.1.5 Make

Im Wikipedia Artikel w:make[4] wird die Aufgabe von make wie folgt zusammengefasst: " **make ist ein Build-Management-Tool, das Kommandos in Abhängigkeit von Bedingungen ausführt.**" In C/C++-Projekten wird es genutzt, die einzelnen Compiler und die anschließenden Linkeraufrufe in einem Tool zentral zu organisieren. Mit jedem Aufruf von make prüft make, in welcher C-/H-Dateien Änderungen vorgenommen wurden (C-/H-Datei neueren Datums als die dazugehörige OBJ-Datei, resp. OBJ-Dateien neueren Datums als die Executable) und übersetzt nur die geänderten Dateien. Ergänzend werden im Makefile auch Projektkonfigurationen vorgenommen (ähnlich .classpath File in Java), über welches das Projekt konfiguriert wird (Debug-Mode, Optimierung, ...)" Die Projektbeschreibung ist in der Datei 'makefile' enthalten, welche bei Start von Make ohne expliziter Angabe einer Projektbeschreibung 'ausgeführt' wird.

```
>>make         #startet Make, welches die Datei makefile einliest und ausführt
>>make -f xyz  #startet make, welches die Datei xyz einliest und ausführt
```

Mit dem Aufruf vom Make können Ziele(Targets) definiert werden, welche behandelt werden sollen. Typische Ziele sind:

```
>>make all       #Prüft das gesamte Projekt auf Änderungen und erstellt die
 Executable
>>make clean     #Löscht alle Zwischenergebnisse (also alle Objekt-Dateien)
```

3.1.6 Entwicklungsumgebung

Ein integrierte Entwicklungsumgebung IDE beinhaltet Softwarekomponenten, die ein einfaches Programmieren, die Ausführung und den Test unter einer Oberfläche ermöglichen. Zur Vermeidung von Medienbrüchen bieten IDE's eine eigene Verwaltungsoberfläche/-sprache an, aus welcher die Anweisungen für den Compiler, Linker, Debugger, Make usw. abgeleitet werden. Zur einfachen und schnellen Softwareentwicklung sind weitere Hilfstools wie Autovervollständigung oder Lösungsvorschläge für Compilerfehlermeldungen vorhanden.
Letztere Hilfstools stellen nach Meinung des Autors einen Nachteil dar, da ergänzend zum syntaktischen Fehler oftmals auch prinzipielle Fehler vorhanden sind.

3 https://de.wikipedia.org/wiki/Loader
4 https://de.wikipedia.org/wiki/make

3.2 Installation / Start beispielhafter Toolchains

In dieser Vorlesung wird der GCC-Compiler als Grundlage genutzt. Damit die Fehlermeldung des Compilers gelesen werden, wird empfohlen wahlweise den Compiler per Hand zu starten oder eine Online Version des Compilers zu nutzen. Von der Nutzung einer IDE wird abgeraten. Für Betriebssystemaufrufe wird der POSIX-Standard vorausgesetzt, wie er bspw. von LINUX, dem Windows Subsystem für Linux WSL, oder macOS unterstützt wird.

3.2.1 Händische Installation der Toolchain

Linux

Im Normalfall sollte unter Linux die benötigte Toolchain installiert sein. Bei einigen Distributionen muss jedoch die Toolchain händisch installiert werden. Dazu geben sie folgende Befehle in ein Terminalfenster ein:

```
>>sudo apt install gcc
>>sudo apt install make
```

Die Erstellung des Source-Codes kann mit einem beliebigen Editor (z.B. gedit, nano) erfolgen.

Windows 10/11 64-Bit Nutzer

Sowohl der GCC-Compiler als auch POSIX konforme OS-Aufrufe werden von Windows von Haus aus nicht unterstützt. Mit Windows Subsystem für Linux (WSL) wird ein Linux System emuliert, welche beide Voraussetzungen erfüllt:

- Installation von WSL

  Entsprechend nachfolgender Anleitung: How to install Linux on Windows with WSL[5]

- Upgrade + Vervollständigung

  Starten einer bash (bspw. durch Eingabe von 'bash' in der Suchzeile)

```
>>sudo apt-get update
>>sudo apt-get upgrade
>>sudo apt install gcc
>>sudo apt install make
```

- Datenaustausch zwischen Windows und WSL

  Für den einfachen Datenaustausch zwischen Windows und WSL empfiehlt sich, im Windows Dateisystem ein Arbeitsverzeichnis anzulegen. Das Windows Dateisystemen ist ab dem Pfad '/mnt/c/...' unter Linux gemountet:

```
>>cd /mnt/c/_Projekte/AufgabeX
```

5 https://docs.microsoft.com/en-us/windows/wsl/install-win10

- Die Erstellung des Source-Codes kann mit einem beliebigen Windows-Editor (z.B. notepad++) erfolgen.

macOS

macOS basiert auf darwin (ein freies UNIX Betriebssystem). Defaultmäßig wird macOS ohne Compiler ausgeliefert. Die Installation eines Compilers sollte identisch zur Installation unter Linux sein.

3.2.2 Händischer Start des Compilers

Beim GCC-Compiler übernimmt der Aufruf von gcc gleichermaßen den Compiler- und den Linker-Aufruf. Eine Übersetzung wird wie folgt gestartet:

```
>>gcc datei1.c datei2.c datei3.c …
```

Diese Anweisung startet zunächst für jede C-Datei den Compiler. Im Anschluss werden alle erzeugten Objekt-Files mit einem weiteren Linkeraufruf zusammengefügt. Als Executable wird die Datei a.out erzeugt. Mit dem Compiler-Parameter -o kann ein alternative Name für die Executable gesetzt werden:

```
>>gcc datei1.c datei2.c datei3.c -o prog
```

Der Aufruf kann beliebig um Compiler-Parameter ergänzt werden:

```
>>gcc datei1.c datei2.c datei3.c -o prog -fsanitize=address -Wall
```

Nach fehlerfreien Compilerdurchlauf wird die Executeable wie folgt gestartet:

```
>>./a.out   #Bei Aufruf von gcc ohne -o
>>./prog    #Bei Aufruf von gcc mit -o prog
```

Wichtige Terminal/bash-Befehle: Siehe auch: Ubuntuusers/Befehlsübersicht[6]

```
>>cd  xxx       #Change Directory, zum Wechseln des Arbeitsverzeichnisses
>>pwd           #zur Ausgabe des aktuellen Verzeichnisses
>>ls           #zur Auflistung des Inhaltes des aktuellen Verzeichnisses
>>ls -la       #zur tabellarischen Auflistung des Inhaltes des aktuellen
 Verzeichnisses
>>ll           #in vielen Installationen ist dies ein Alias auf 'ls -la'
>>cp xxx yyyy  #zum Kopieren einer Datei 'xxx' in die neue Datei 'yyy'
>>mv xxx yyy   #zum Verschieben/Umbenennen von Dateien
>>mkdir xxx       #zum Anlegen eines Unterverzeichnisses
>>rm xxx        #zum Löschen einer Datei
>>rmdir xxx       #zum Löschen eines Unterverzeichnisses (dieses muss jedoch
 leer sein)
>>cat xxx       #Eigentlich ein Befehl zum Zusammenfügen mehrerer Dateien.
               #Da ohne explizite Angabe einer Zieldatei die Ausgabe auf der
               #Standardausgabe erfolgt, kann mit diesem Befehl der Inhalt
               #einer Datei angezeigt werden.
>>less xxx       #scrollfähige Anzeige einer Datei (Beenden mit Taste 'q')
>>more xxx       #Wie less, aber ohne die Fähigkeit, rückwärts zu scrollen
>>vi main.c       #ein einfacher, aber universeller Texteditor
>>nano main.c  #ein etwas komfortabler Texteditor
```

6 https://wiki.ubuntuusers.de/Shell/Befehls%C3%BCbersicht/

3.2.3 Nutzung des Online-Compilers Compiler Explorer

Der Compiler Explorer ist eine komfortable Online-Entwicklungsumgebung, welche diverse Programmiersprachen, diverse Compiler unterstützt, eine komfortable Ausführungsinstanz für die Executable bietet und den erzeugten Assemblercode darstellen kann.

`https://www.godbolt.org`

Für den einfachen Start soll hier der prinzipielle Umgang mit dem Compilerexplorer erläutert werden:

- Eingabe des Source-Codes

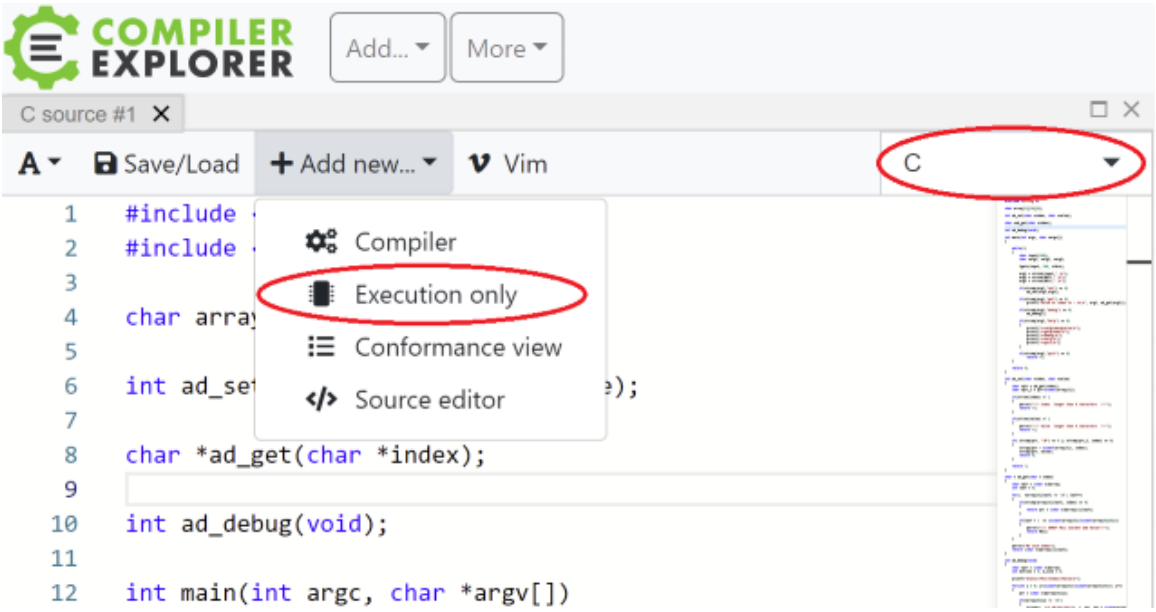

Abb. 3 Beschreibung der Editor Eingabefelder vom Compiler Explorer

Die Eingabe des Source-Codes ist selbsterklärend. Rechts oben in der Eingabebox sollte C als Programmiersprache gewählt werden. Der Compilevorgang startet automatisch, wenn für ca. 1 Sekunde keine Änderung mehr am Source-Code vorgenommen wird.

Beim ersten Start des Compilerexplorers wird defaultmäßig der compilierte Assemblercode dargestellt. Der besseren Übersicht sollte dieses Fenster geschlossen werden.

Über 'Add New-Execution only ' wird die Executable zur Ausführung gebracht. Wer den erzeugten Assemblercode oder Zwischenergebnisse des Compilers analysieren möchte, wählt Add New-Compiler.

- Executor

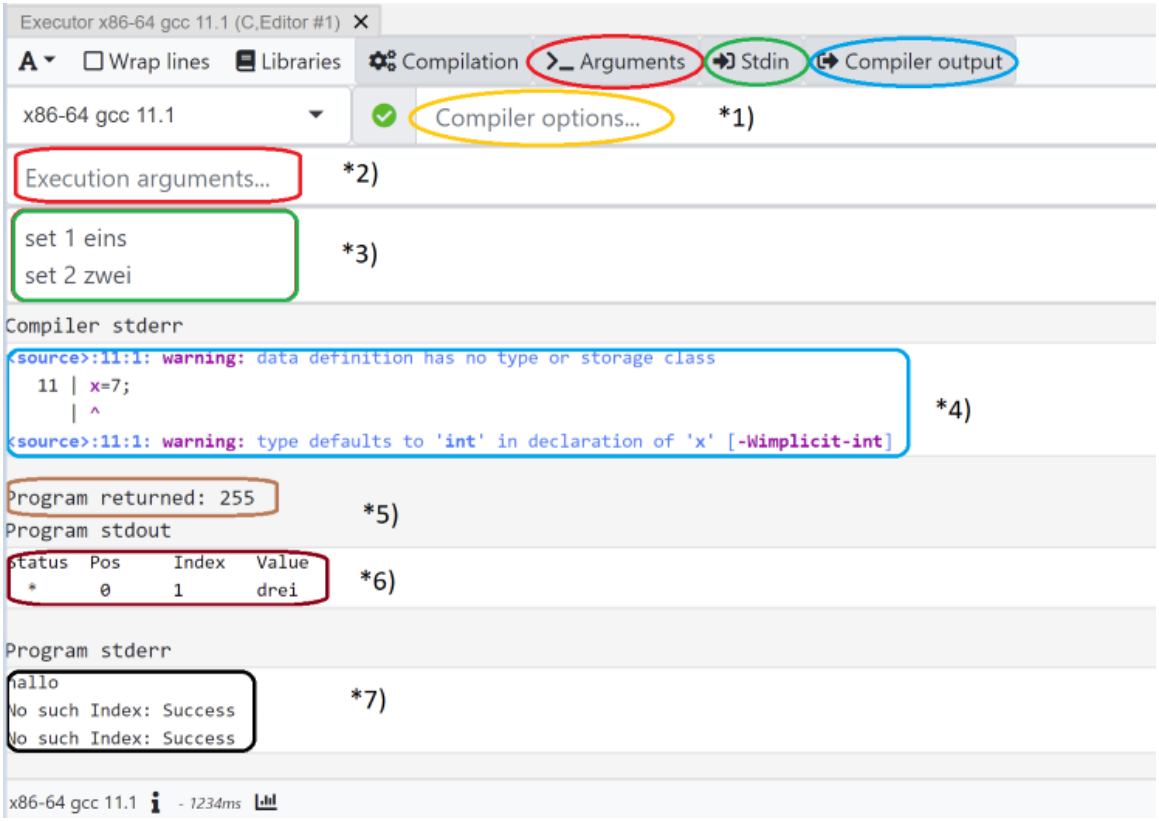

Abb. 4 Beschreibung der Executor Eingabefelder vom Compiler Explorer

Sobald der Haken im grünen Kreis erscheint, ist der Compilevorgang und die Programmausführung abgeschlossen. Änderungen bei *2) und *3) führen direkt zu einem Neustart des Programms. Mit Änderung von *1) oder Wahl eines anderen Compilers wird der Compilevorgang ergänzend neu gestartet. Über die 'Menü'-Zeile können einzelne Fenster ein- und ausgeblendet werden. Es wird empfohlen, alle Fenster einzublenden!

*1) Hier können dem Compiler-Parameter übergeben werden. Es empfiehlt sich '-fsanitize=Address -Wall -Werror' vorzugeben, so dass eine Speicherüberwachung in den Source-Code integriert wird und eine erweiterte Fehlerprüfung stattfindet. *2) Hier können der Executable Parameter übergeben werden, die dann in argc und argv von main() abrufbar sind. Entspricht dem händischen Start des Programms über './a.out para1 para2' *3) Hier wird die Standardeingabe nachgebildet. Allerdings wartet der Executor nicht auf eine aktive Eingabe durch den Nutzer, sondern gibt mit jeden Aufruf von scanf() oder fgets() die Inhalte der einzelnen Zeilen aus diesem Fenster zurück. D.h. der erste Aufruf von fgets() liefert hier 'set 1 eins' und der zweite Aufruf 'set 2 zwei' zurück. *4) Sofern mit 'Compiler Output' aktiviert, werden hier alle Compiler-Warnings angezeigt. Wichtig: Warnings sollten als Fehler angesehen werden und dementsprechend im Source-Code behoben werden *5) Hier wird der Rückgabewert von 'int main()' angezeigt. Sollte das Programm sich in einer Endlosschleife befinden (und damit die Laufzeit von ca. 10 Sekunden überschreiten), so wird das Programm nach ca. 10 Sekunden vom Compiler Explorer beendet und ein Returnwert von 143 angezeigt. Dem Wert 'Programm returned: xxx' sollte einer besonderen Aufmerksamkeit zugedacht sein. *6) In diesem Fenster wird die Standardausgabe dargestellt, also

alles, was sie bspw. mit printf() ausgeben. *7) In diesem Fenster wird die Standardfehlerausgabe dargestellt, alles, was sie bspw. Mit fprintf(stderr, ...) Oder perror(...) Ausgeben. Die Nutzung der Standardfehlerausgabe empfiehlt sich insbesondere, wenn Laufzeitfehler vorhanden sind. Aufgrund dessen, dass die Standardausgabe nicht gepuffert ist, werden alle Ausgaben direkt zur Anzeige gebracht.

3.3 Fehlersuche von Laufzeitfehlern

Ein fehlerfrei übersetzter Code heißt nicht, dass das erzeugte Programm fehlerfrei ist. Ein erfolgreicher Compilerdurchlauf bedeutet nur, dass alle C-Anweisungen in entsprechende Maschinenbefehle umgesetzt werden konnten. Fehler können auch zur Laufzeit (siehe Kap. Einführung/Literatur-Laufzeitfehler[7]) eines Programmes auftreten.

Laufzeitfehler machen sich wahlweise durch Programmabsturz oder durch ein fehlerhaftes Verhalten des Programmes bemerkbar.

Bei einem Programmabsturz beendet das Betriebssystem die Fortführung des Programmes z.B. aufgrund eines fehlerhaften Speicherzugriffes (Segmentation Fault). Beim Compiler-Explorer begrenzt der Executor die Laufzeit auf ca. 10 Sekunden und beendet dann ebenfalls das Programm. Beide 'Programmabstürze' sind dadurch gekennzeichnet, dass die main() Funktion sich nicht selbst beendet und damit der Return-Wert nicht zur Ausführung gebracht wird. Beim Compiler Explorer empfiehlt sich, den Return-Wert (und den Grünen Hacken) im Auge zu behalten.

Ein fehlerhaftes Verhalten des Programms macht sich bspw. dadurch bemerkbar, dass ein Variable einen anderen Wert beinhaltet, als erwartet:

```
int lauf = 4711;
int var  = 2;
int *ptr=&var;
*--ptr=13;
printf("%d\n",lauf);  //Erwartet 4711, Ausgabe 13
```

Ungeachtet der Art des Auftretens des Fehlers liegt der Fehlergrund zumeist nicht an der Stelle, an der sich der Fehler bemerkbar macht.

Das Mittel der Wahl zur Fehlersuche sind die in Kap. Einführung/Literatur-Laufzeitfehler[8] dargestellten Methoden, also ein Codereview durchzuführen. Ergänzend können printf()-Anweisungen helfen, den Fehler einzugrenzen. Dabei sollten beachtet werden, dass printf()-Anweisungen nicht gepuffert sind und wahlweise von einer fflush(stdout) gefolgt werden sollte oder durch fprintf(stderr,..) erfolgen sollten.

Hinweise:

- Ein Softwaretest sollte nicht nur die Gutfälle abdecken, sondern insbesondere die Schlechtfälle berücksichtigen. Dabei sollte der Kreativität freien Lauf gelassen werden. Der An-

7 Kapitel 2.5.3 auf Seite 14
8 Kapitel 2.5.3 auf Seite 14

wender der späteren Software nutzt sicherlich nicht immer den von Ihnen vorgesehenen Weg, sondern nutzt alle Wege, die nicht explizit deaktiviert sind (unwissend oder auch bewusst).

- Werden Laufzeitfehler nicht behoben, so sind dies häufig Einfallstore für Viren. Die 25 gefährlichsten Softwarefehler werden von Common Weakness Enumeration (CWE) unter `https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html` aufgelistet. Auf einer der obersten Plätze sind Arrayzugriffe außerhalb des reservierten Bereiches zu finden.

Das Open Web Application Security Project (OWASP) führt ebenfalls eine Liste der Sicherheitsbedrohungen im Web: `https://owasp.org/Top10/`

4 Grundlagen

Die Programmiersprache C/C++ beinhaltet mehrere Sprachen/ Syntaxen:

- C-Syntax
- Präprozessor-Syntax
- Printf/Scanf Formatstring Syntax
- Terminal Emulation
- Compiler/Linker Anweisungen

Letztere beiden werden nur rudimentär in dieser Vorlesung/diesem Skript behandelt. Alle anderen sind Bestandteil dieser Vorlesung.

In diesem Kapitel sollen zunächst allgemeine Eigenschaften der Sprache, der grundlegende Syntax und die Kontrollstrukturen erklärt werden. Bei vielen Erklärungen sind Code-Beispiele vorhanden. Da man aus Fehlern am meisten lernt, sind zum Teil auch negative Beispiele enthalten. Für ein besseres Verständnis empfiehlt es sich, Code-Beispiele selbst nachzuvollziehen.

Sofern sich die Eigenschaften auf eine bestimmte C-/C++- oder Compiler-Version beziehen, wird dies gesondert vermerkt.

4.1 Zeichensatz

Der Syntax von C nutzt die unteren 128 Zeichen des ASCII Zeichensatzes. Da UTF-8 in den ersten 128 Zeichen deckungsgleich zu ASCII ist, kann auch dieser zur Erstellung des Source Codes genutzt werden. Zeichen außerhalb dieses gültigen Zeichensatzes können folglich nur in Strings oder Kommentaren vorkommen. Werden Zeichen außerhalb der unteren 128 Zeichen in Strings genutzt, so gilt Folgendes zu berücksichtigen:

- Die String-Funktionalitäten der Standard-C Library gehen von dem regulären ASCII Zeichensatz aus, so dass z.B. die Suche, der Vergleich oder die Konvertierung fehlschlagen kann
- Strings werden oftmals in Verbindung mit printf() genutzt, d.h. auf der Standardausgabe ausgegeben. Der verwendete Zeichensatz sollte hier identisch zum verwendeten Zeichensatz der Terminal-Emulation sein!

4.2 Kommentarzeichen

Kommentare dienen dazu, den Source Code mit zusätzlichen Hinweisen zu versehen, so dass die Intension der Anweisungen sichtbar wird. Kommentarzeichen werden durch den

Compiler vor dem eigentlichen Übersetzungsdurchlauf entfernt und durch ein Leerzeichen ersetzt [C11 6.4.9]. Innerhalb von Strings wird nicht nach Kommentarzeichen gesucht.

`Syntax: /* */`

Blockkommentar zum Kommentieren eines Bereiches, auch über mehrere Zeilen hinweg. Dieses Kommentarzeichen kann nicht verschachtelt werden, d.h. '*/' beendet die Kommentierung, unabhängig von der Anzahl der zuvor geöffneten Kommentierungen!

`Syntax: //`

Zeilenkommentar zum Kommentieren bis zum Zeilenende (wurde in der Programmiersprache BCPL definiert, aber nicht von C, sondern erst von C++ übernommen).

Sollen größere Bereiche auskommentiert werden, so empfiehlt sich die Nutzung der Präprozessoranweisung #if 0 ... #endif. Beispiele:

```
/* Blockkommentar mit zwei öffnenden Klammern /*
   über mehrere Zeilen, nicht verschachtelt */
//Zeilenkommentar
/* Kleiner Bereich zum auskommentieren */
#if 0
Großer Bereich zum auskommentieren
#endif
```

4.3 Namenskonventionen

Folgende Regeln gelten bzgl. der Benennung von Variablen und Funktionen:

- Variablen und Funktionsnamen können aus Buchstaben, Zahlen und dem Unterstrich bestehen. Sie müssen mit einem Buchstaben oder einem Unterstrich beginnen
- Seit C95 können weitere Zeichen aus dem Universal Coded Character Set (UCS) genutzt werden [C11 D.1]
- C ist Case sensitiv, d.h. es wird zwischen Groß- und Kleinbuchstaben unterschieden
- Schlüsselwörter können nicht für Variablen/Funktionsnamen/Datentypen genutzt werden

Namenskonventionen von Libraryfunktionen:

- In C (und C++) sind Schlüsselwörter und Standardlibraryfunktionen zumeist in Kleinbuchstaben geschrieben. In der C-Standardlibrary werden oftmals verkürzte Ausdrücke (z.B. isalnum() (zum Testen ob ein Zeichen ein Buchstaben oder ein Digit ist) und in C++ der Unterstrich als Worttrenner (z.b. out_of_range) genutzt
- Makros (siehe Kap. Präprozessor[1]) werden per Konvention in GROSSBUCHSTABEN und ggf. mit Unterstrich als Worttrenner geschrieben.
- Namen beginnend mit doppelten Unterstrich oder beginnend mit einem Unterstrich gefolgt von einem Großbuchstaben (z.B. __LINE__ _Reserved) sind für den Compiler und der Standard-C-Library vorbehalten und sollten im eigenen Programm nicht benutzt werden.

1 Kapitel 9 auf Seite 203

4.4 Zeilenfortsetzung

Mit dem Backslash Operator (gefolgt von einem Zeilenende) kann eine Zeile in der nächsten Zeile fortgesetzt werden. Der Compiler löscht das \-Zeichen mit anschließendem Zeilenende und ersetzt dies durch nichts. Dies ist insbesondere bei Strings und bei Makros von Interesse, die am Ende der Zeile abgeschlossen sein müssen.

Beispiele:

```
char str1[]="Strings müssen am Ende abgeschlossen sein
             so dass dies ein Fehler ist";
char str2[]="Dies\
       ist ein Test";  //Vorsicht, führende Leerzeichen vor 'ist'
                       //bleiben erhalten!
 /*mehrzeiliges Makro*/
#define MAX(a,b)    (a>b?\
                     b:  \
                     a)
//Dies ist ein Zeilenkommentar \
welcher in dieser Zeile fortgesetzt wird
/\
* dies ist ein Blockkommentar*\
/
//hinter der Zeilenfortsetzung darf nur ein CR/LF folgen
#define MAX2(a,b)    \  //so dass hier kein Kommentar folgen darf
            a>b?a:b
```

Hinweis:

- Innerhalb von Char-Literatoren und Strings wird '\' als Escape-Operator genutzt, welche das '\' und ein oder mehrere folgende Zeichen ersetzt (siehe Kap. Grundlagen:Literale/Konstanten[2]). Daher darf hinter Backslash als Zeilenfortsetzungszeichen kein weiteres Zeichen folgen.

4.5 Gültigkeit/Sichtbarkeit von Variablen

Vorrangig in der objektorientierten Programmierung werden mit Namensräumen Objekte und deren Methoden/Attribute in einer Art Baumstruktur strukturiert. Dies ermöglicht eine eindeutige Ansprache von Variablen/Objekte, aber auch eine doppelte Verwendung von Methoden-/Attributnamen in unterschiedlichen Namensräumen. Ergänzend zu den Namensräumen kann mit public/private/proteced eine Zugriffsbeschränkung von Methoden/Attributen definiert werden.

Die Programmiersprache C unterstützt keinen Namensraum. Zugriffsmodifikatoren werden indirekt über Header-Dateien getätigt. Hinsichtlich der Gültigkeit/Sichtbarkeit unterscheidet C folgende Bereiche [C11 6.2.1]:

- Funktionen (Function Scope)
- Datei (File Scope)
- Block (Block Scope)
- Funktionsparameter in Prototypen (Function Prototype Scope)

2 Kapitel 4.10 auf Seite 53

Innerhalb eines Gültigkeitsbereiches dürfen Variablen-/Funktions-/Datentypnamen nicht doppelt genutzt werden.

In der Programmiersprache C++ sind weitere Scope wie z.B. Class Scope, Enumationation Scope und ergänzend das Konzept von Namensräumen vorhanden (Beschreibung folgt).

4.5.1 Funktionsweite Sichtbarkeit

Eine Label-Definition (als Sprungmarke für die goto-Anweisungen) erfolgt immer mit funktionsweiter Sichtbarkeit/Gültigkeit. Die genaue Beschreibung dieses Sichtbarkeitstyps erfolgt im Rahmen des Kapitels Grundlagen:Label+Goto[3].

4.5.2 Dateiweite Sichtbarkeit

Erfolgt eine Funktion-/Variablen-/Datentyp-Definition außerhalb eines Block-Scopes oder von Funktionsparameter, so sind diese innerhalb der gesamten Datei und bei Funktionen/Variablen ergänzend Projektweit (für alle Objektdateien) sichtbar/gültig (= global) (In der C-Spec unter dem Stichwort 'Linkage' beschrieben, siehe Kapitel DatentypeSpecifier:Internal/External Linkage[4]). Alle globalen Funktionen/Variablen können von allen Dateien aus genutzt/zugegriffen werden (sofern sie zuvor deklariert wurden).

Beispiel:

Datei1.c

```
//Deklaration von func(),
//welche in Datei2.c definiert wird
extern void func(void);

//Definition der Variablen global
int global=0;
int main(void) {
  func();
  global++;
  return 0;
}
```

Datei2.c

```
//Deklaration von global,
//welche in Datei1.c definiert wird
extern int global;

//Definition der Funktion func()
void func(void) {
  global++;
}
```

Wird das Schlüsselwort 'static' der Variablen/Funktionsdefinition vorangestellt, so wird die Sichtbarkeit/Gültigkeit auf Dateiweit eingeschränkt. Variablen/Funktionen können nur innerhalb der (Objekt-) Datei genutzt werden und sind für anderen (Objekt-)Dateien unsichtbar. (Siehe Kap. DatentypeSpecifier:Internal/External Linkage[5] Internal/External Linkage)

Datei1.c

Datei2.c

3 https://de.wikibooks.org/wiki/Programmieren%20in%20C%2FC%2B%2B%3A%20%20Grundlagen%20%23Label%20%2B%20Goto

4 Kapitel 6.2 auf Seite 124

5 Kapitel 6.2 auf Seite 124

```c
//Dateiweite Sichtbarkeit von var1
static int var1;

int main(void) {
  var1++;
  return 0;
}
```

```c
//Dateiweite Sichtbarkeit von var1
static int var1;

void func(void) {
  static int var2; //Vorsicht, static
  // hat hier eine andere Bedeutung
  var1++;
}
```

Projektweite Gültigkeit bedeutet insbesondere, dass keine doppelten Benennung von Variablen/Funktionen/Datentypen innerhalb des gesamten Projektes erlaubt sind, d.h. alle Variablen/Funktionennamen über alle Dateien/Librarys eindeutig sein müssen.

Beispiel 1:

Datei1.c

```c
#include <stdio.h>
int a=1;
int main(int argc,char *argv[])
{
  printf("Hello World %d",a);
  void dummy(void);  //Deklaration von Dummy
  dummy();
  return 0;
}
```

Datei.2

```c
#include <stdio.h>
int a=7;
void dummy(void)
{
  printf("Hello Again %d\n",a);
}
```

Der Linker meldet beim Zusammenfügen der Objekt-Dateien, dass die Variable a bereits woanders definiert sei (multiple definition of ‘a';)!

Beispiel 2:

```c
#include <stdio.h>   //fuer printf() size_t stderr
int printf=7;
```

Hier meldet der Compiler eine Fehlermeldung, da das Symbol printf zu einen als Variable und zum anderen als Funktion (innerhalb der inkludierten Datei stdio.h beschrieben) genutzt wird ('printf' redeclared as different kind of symbol).

4.5.3 Blockweite Sichtbarkeit

Erfolgt eine Funktion-/Variablen-/Datentyp Definition innerhalb einer Funktion, als Funktionsparameter oder eines Blockes, so sind diese nur innerhalb des Blockes sichtbar/gültig (=Lokale Variable). Blöcke können verschachtelt sein, so dass das bei identischer Namensgebung innere Definitionen Vorrang haben. Ebenso haben Blockdefinitionen Vorrang vor Datei-/Projektdefinitionen (überdecken diese):

Datei1.c | **Datei2.c**

```
int main(void) {
  int var2;    //var2 ist nur innerhalb
               //von main() sichtbar
  struct xyz   //Datentyp ist nur innerhalb
  {int x,y,z;};//von main()
               //sichtbar/gültig
  extern void func(void); //Deklaration
          //ist nur innerhalb von
          //main() sichtbar/gültig
    func();
}

void foo(void) {
 struct xyz var_xyz;  //Fehler, da
 //Datentypedefinition hier nicht mehr
 //gültig ist!

 func();  //Fehler, da Deklaration
 //hier nicht mehr gültig ist
}
```

```
#include <stdio.h>
void func(void) {
  int var2=1;  //var2 ist
  //nur innerhalb von
  //func() sichtbar

  {
    int var2=2;
    //var2 ist nur innerhalb
    //dieses Blockes sichtbar
    printf("%d\n",var2);
  }
  printf("%d\n",var2);
}
```

4.5.4 Sichtbarkeit von Funktionsparameter in Protypen

Soll eine Funktion aufgerufen werden, die zuvor nicht definiert wurde, so ist ein Prototyp/(Forward)deklaration der Funktion notwendig (siehe nachfolgendes Kapitel). Die Parameterliste dieses Prototyps hat seinen eigenen Gültigkeitsbereich.

Beispiel:

```
int a;   //a im File Scope
int func(int a, int b);  //a und b im Function Prototyp scope
```

Hinweis:

- Bei Prototypen ist die Angabe von Variablenname nicht notwendig. Die alleinige Angabe des Datentyps ist ausreichend

4.5.5 Zuordnung der Gültigkeitsbereiche zu globalen/lokalen Variablen

Zum einfacheren Verständnis wird im Skript von folgenden Begrifflichkeiten gesprochen:

- **Globale** Variablen-/Funktionen-/Datentypdefinitionen → Projektweite Sichtbarkeit/Gültigkeit
- **Statisch globale** Variable/Funktionen → Dateiweite Sichtbarkeit/Gültigkeit
- **Lokale** Variablen/Funktionen/Funktionsparameter/Datentypen → Nur innerhalb des zugehörigen Blockes sichtbare/gültig
- **Statisch lokale** Variablen → Nur innerhalb der Funktion sichtbar/gültig, mit der Besonderheit, dass die Gültigkeit über der Laufzeit der Funktion gilt (siehe Kap. DatentypeSpecifier:Static Storage Class Specifier[6])

6 Kapitel 6.9 auf Seite 135

4.5.6 Speicherzuweisung

Für die einzelnen Gültigkeitsbereiche werden im Speicher (und auch in der Objekt-Datei) unterschiedliche Speicherbereiche (Segmente) vorgehalten, d.h. der Compiler teilt den Speicher in unterschiedliche Bereiche auf und weist den Variablen/Funktionen in Abhängigkeit der Gültigkeit/Sichtbarkeit unterschiedliche Speicherbereiche zu!

```c
#include <stdio.h>
#include <stdlib.h>
        int varglobinit=33;
        int varglob;
const   int varconst=4711;

void dummy(int para1)
{
    printf("Para1=%d\n",para1);
}
int main(int argc,char *argv[])
{
    int lauf;
    int *ptr=malloc(100*sizeof(int));
    for(lauf=99;lauf>=0;lauf--) {
         ptr[lauf]=100-lauf;
         dummy(ptr[lauf]+varstat);
    }
    free(ptr);
    return 0;
}
```

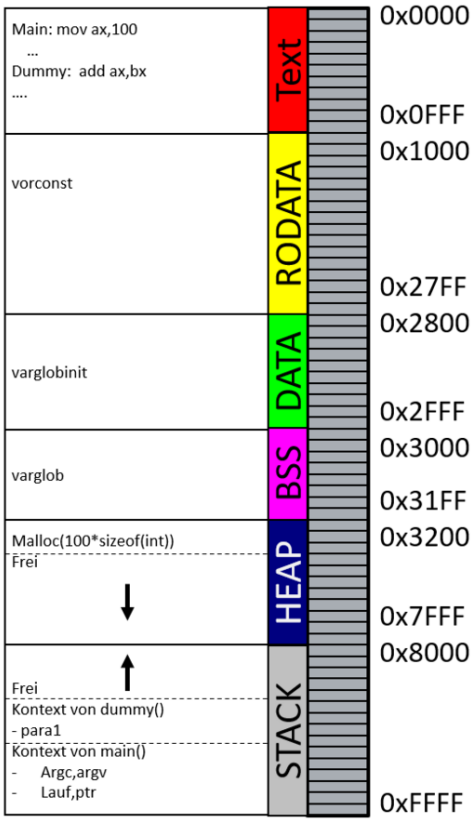

Abb. 5 Zuordnung von Inhalten eines C-Programms zu Speicherbereichen (Segmenten)

Zur Ermöglichung von Rekursion werden lokale Variable auf dem Stack gehandelt. Mit Aufruf einer Funktion oder öffnen eines neuen Block-Scopes wird Speicher auf dem Stack reserviert, der bei Beenden des Scopes wieder freigegeben wird. Die Speicheradresse von lokalen Variablen ist somit nicht Fix, sondern hängt von der Aufrufhierarchie ab!

Globale und statisch globale Variablen werden nicht auf dem Stack gehalten, sondern bekommen einen 'festen' Speicherbereich (hier DATA/BSS/RODATA) während der gesamten Laufzeit des Programms zugewiesen. Die Adresse dieser Variablen sind somit konstant. Das genutzte Speichersegment von globalen Variablen hängen von der Art der Variablen ab. Initialisierte Variablen werden im DATA-Segment gesammelt, welcher mit dem Initialisierungswert der Variablen initialisiert wird. Nicht initialisierte globale Variablen landen im BSS-Segment, so dass diese mit Programmstart durch Füllen dieses Blockes mit 0 auf 0 gesetzt werden. Konstante globale Variablen werden im RODATA-Segment gehalten. Über das Betriebssystem kann dieser Speicherbereich auf nur Lesbar gesetzt werden.

Dynamische Speicheranforderungen, resp. Speicher, dessen Gültigkeit über die Laufzeit einer Funktion hinausgeht, wird im Heap gesammelt. Für die Verwaltung dieses Speichers ist der Programmierer zuständig.

Entsprechend globalen Variablen wird für Funktionen ebenfalls ein eigener Speicherbereich (TEXT Segment) vorgesehen. Im Gegensatz zu Variablen, wo im Speicher der dazugehörige Inhalt gespeichert ist, ist der Speicherinhalt von Funktionen die Maschinenanweisungen. Die Speicheradresse der Funktionen sind konstant.

Hinweis:

- Die Speicheradresse von sowohl globalen und statisch globalen Variablen als auch von Funktionen ist konstant und ändert sich während der Laufzeit des Programms nicht. Die Adressvergabe dieser Variablen und Funktionen erfolgt somit nicht zur Laufzeit sondern durch die Toolchain (hier der Linker).
- Dieses Konzept ist allgemeingültig, gilt nicht nur für die Programmiersprache C/C++

4.6 Definition/Deklaration(Prototyp)

Der C-Compiler ist ein Single-Pass Compiler (siehe Kap Einführung/Literatur:Arbeitsweise eines Compilers [7]), d.h. für den Übersetzungsvorgang wird der Quellcode einmalig eingelesen und von vorne nach hinten abgearbeitet. Dies bedingt, dass keine Vorwärtsbezüge im Quelltext enthalten sein dürfen:

- keine Variable genutzt werden kann, die erst später definiert wird (da der Compiler in Abhängigkeit des Datentyps (den er dann noch nicht kennt) keine entsprechende Maschinensprachebefehle erzeugen kann)
- keine Funktion aufgerufen werden kann, die erst später definiert wird (da der Compiler nicht weiß, welcher Datentyp die Übergabewerte haben und folglich keine Maschinensprachebefehle zur Konvertierung und Übergabe der Variablen erzeugen kann)
- kein Datentyp genutzt werden kann (struct, union, enum, typedef), welcher erst später definiert wird (da der Compiler nicht weiß, wieviel Speicher er für die Variable reservieren muss)

Eine Lösungsvariante dieser Problematik ist, im Quellcode alle Variablen/Funktionen/Datentypen/Makros zu definieren, bevor diese erstmalig genutzt werden (Funktion main() steht dann folglich am Ende des Quellcodes):

```
void foo(void) {   //foo() wird vor dem ersten Aufruf definiert
  //Hier kein Zugriff auf var möglich!
  //Hier kein Zugriff auf Datentyp uchar möglich!
}
int var;            //var wird vor dem ersten Aufruf definiert
typedef unsigned char uchar; //Definition des Datentyps uchar
int main(void) {
  foo();     //da zuvor definiert kann diese Funktion hier aufgerufen werden
  var++;     //dito
}
```

Sollen globale Funktionen/Variablen in Datei1.c genutzt werden, die in Datei2.c definiert sind, so schlägt diese Vorgehensweise fehl. Um auch dieses zu ermöglichen, bietet C die

7 Kapitel 2.6.1 auf Seite 17

Möglichkeiten von Prototypen/Deklarationen an. Diese teilt dem Compiler mit, wie im späteren Ablauf die Variable, die Funktion definiert wird, bzw. dass es diesen Datentyp geben wird:

Datei1.c

```
//Prototyp für Variable var
extern int var;

//Prototyp für Funktion foo
void foo(void);

int main(int argc, char *argv[]) {
  foo();  //Compiler kann aufgrund
  var++;  //der Prototypen diese
            //Aufruf in Maschinen-
            //sprachebefehle umsetzen
}
```

Datei2.c

```
//Definition der Variablen var
int var;

//Definition der Funktion foo
void foo(void)  {
}
```

Innerhalb der C-Spezifikation werden für den Umgang mit dieser Problematik die Begriffe 'Declaration' und 'Definition' genutzt. Da diese Begriffe jedoch nicht klar abgegrenzt werden und auch für viele weitere Aspekte genutzt werden, sollen in diesem Skript in Anlehnung an den Wikipedia Artikel Deklaration (Programmierung)[8] (im Absatz zur Programmiersprache C/C++) die Begriffe Definition/Deklaration wie folgt genutzt werden:

Definition

entspricht der Belegung/Reservierung von Speicherplatz

- Bei der Definition einer Variablen wird hiermit Speicherplatz entsprechend dem Datentyp der Variable belegt:

```
int var1;       //entsprechend der Größe des Datentyps int wird
                //Speicherplatz für die Variable var1 reserviert
int var2=4711;  //Dito, ergänzend wird dieser Speicherplatz mit dem
                //Wert 4711 initialisiert
```

- Bei einer Funktion wird Speicherplatz zur Aufnahme der Maschinenbefehle belegt:

```
void foo(void) {   //Die Anweisungen in den geschweiften Klammern
                   //werden in Maschinenbefehle übersetzt. D.h. die
                   //Funktion foo() belegt Speicher, dessen Inhalt
                   //die Maschinensprachebefehle sind
}
```

- Ergänzend wird der Begriff Definition auch in Verbindung mit der Definition von neuen Datentypen genutzt!

Deklaration/Prototyp

ist die Mitteilung an den Compiler, wie diese Variable/Funktion an einer späteren Stelle im Source-Code oder in einer anderen Source-Datei definiert wird. Auf Grundlage dieser 'vorab' Information kann der Compiler die passenden Maschinensprachebefehle für den Zugriff/Aufruf auf die Variablen/Funktion einsetzen:

8 https://de.wikipedia.org/wiki/Deklaration%20%28Programmierung%29

```
extern int var1;        //Deklaration von var1
extern void foo(void);  //Deklaration von foo()
       void foo(void);  //Deklaration von foo()
```

Insbesondere bei dem Umgang mit der Deklaration gilt es Folgendes zu berücksichtigen:

- Deklarationen werden entweder:
  - am Anfang einer C-Datei geschrieben, so dass Funktionen/Variablen im Zugriff stehen, die erst später in der C-Datei definiert werden (entspricht dann der Private Anweisung in anderen Sprachen)
  - in Header-Dateien geschrieben, so dass diese Variable/Funktionen für alle sichtbar sind, welche diese Header-Datei einbinden (entspricht der Public Anweisung in anderen Sprachen)
- Deklarationen von Funktionen werden in der C-Spec als (function) Prototyp bezeichnet und sind dadurch gekennzeichnet, dass diese mit einem Semikolon abgeschlossen sind. Der Specifier 'extern' ist hier optional und sagt eigentlich aus, dass diese Funktion external Linkage besitzt (siehe DatentypeSpecifier:Internal/External Linkage[9]). Die Benennung der Parametervariablen sind gleichermaßen optional:

```
extern int foo(int   var1, float var2);  //Deklaration
extern int foo(int       , float     );  //Deklaration
       int foo(int   var1, float var2);  //Deklaration + external Linkage
       int foo(int       , float     );  //Deklaration + external Linkage
```

- Für globale Variablen gibt es keine (Variablen) Prototype. Stattdessen muss zur Bekanntgabe des Datentyps der Specifier 'extern' vorangestellt werden, mit welchen ergänzend zum Ausdruck gebracht wird, das diese Variable external Linkage und damit globale Gültigkeit hat (siehe Kap DatentypeSpecifier:Extern - Storage Class Specifier[10]). Da die Deklaration einzig zur Bestimmung der Zugriffsmethoden auf Maschinenspracheebene genutzt wird, können diese keine Initialisierungswerte enthalten:

```
extern int var1;
extern int var2=7; //KO Deklarationen können keine
                   //Initialisierungswerte haben
```

- Deklarationen von statisch globalen, lokalen und statisch lokalen Variablen nicht möglich sind:

```
extern static int glob; //keine Deklaration von statisch globalen
                        //Variablen möglich
int main(int argc, char *argv[]) {
```

9 Kapitel 6.2 auf Seite 124
10 Kapitel 6.9 auf Seite 135

```
  extern int var;  //Deklaration bezieht sich auf eine globale
                   //Variable, die nicht definiert ist
  extern static int lok;
  printf("%d %d\n",var,lok);
  int var;
  static int lok;
}
```

- Deklarationen für identische Variablen/Funktionen können beliebig oft erfolgen, sofern der Datentyp identisch ist:

```
extern int var;
extern int var;           //OK, doppelt Deklaration mit identischen Datentyp
extern unsigned int var;  //KO, Doppelte Deklaration mit unterschiedlichen
                          //    Datentyp
```

- Sofern eine Deklaration und eine Definition in der identischen Datei erfolgen überprüft der Compiler, ob Deklaration und Definition identisch sind. Es empfiehlt sich, die eigene Header-Datei (in welcher die Deklaration normalerweise enthalten ist) zu inkludieren:

Datei1.c

```
//Include der eigenen Header-
Datei
#include "datei1.h"
int var1;   //Definition von var1
short var2; //Definition von var2

int var3;  //Private Variable
           //da kein Prototyp
           //im Header vorhanden
void func(void) {
}
```

Datei1.h

```
extern int var1;  //Deklaration
extern char var2; //KO, da Deklaration
                  //von Definition
                  //abweicht
void func(int);   //KO, da Deklaration
                  //von Definition
                  //abweicht
```

- Innerhalb einer Datei kann eine Variable identischen Datentyps mehrmals definiert werden (bei identischen Initialisierungswert). Diese Definitionen werden vom Compiler als eine Definition angesehen. Wenn jedoch eine identische Variable in unterschiedlichen Dateien definiert wird, so legt jede Datei eine eigene unabhängige Variable an, welche dann beim Zusammenführen der Objektdateien durch den Linker zu Fehler führt. Da Header-Datei zumeist in mehrere C-Dateien inkludiert werden, sind Definitionen in Header-Dateien 'verboten'.
- Deklarationen innerhalb eines Block-Scopes oder eines Function Prototyp Scopes sind nur in diesen Bereich gültig.

4.7 Grundlegende Datentypen und Typsicherheit

Die Programmiersprache C kennt folgende grundlegende Datentypen (siehe Kap. Datentypen[11]). Von jedem Datentyp kann ein Zeiger angelegt werden:

Datentyp	**Bsp. für Datentyp**	**Bsp. für Zeiger auf Datentyp**

Ganzzahl	`char var;`
	`int var;`
	`short int var;`
	`long int var;`
	`long long int var;`
		`char *ptr;`
		`int *ptr;`
		`short int *ptr;`
		`long int *ptr;`
		`long long int *ptr;`

Gleitkommazahl	`double var;`
	`float var;`
	`long double var;`
		`double *ptr;`
		`float *ptr;`
		`long double *ptr;`

Funktionen	`void func(void) {}`	`void (*pfunc)(void);`

Arrays	`int arr[10];`
	`float arr[3][3];`
		`int (*parr);`
		`float (*parr)[3];`
		`void (*pfunc[5])(void);`

Strukturen/Unions	`struct xyz {int x,y,z};` `union abc {int a,b,c};`	`struct xyz *ptr;` `union abc *ptr;`
Aufzählungstyp	`enum abc {A,B,C};`	`enum abc *ptr;`
Boolescher Datentyp	`_Bool var;`	`_Bool *ptr;`

11 Kapitel 5 auf Seite 91

Datentyp	**Bsp. für Datentyp**	**Bsp. für Zeiger auf Datentyp**
Komplexe Zahlen	`_Complex var;`	`_Complex *ptr;`

Mit der Defintion (dem Anlegen) einer Variablen eines Datentyps wird Speicher entsprechend der Größe des Datentyps reserviert. Optional kann dieser Speicher initialisiert werden. Über den Datentyp wird ausgesagt, wie der Inhalt des Speichers zu interpretieren ist:

```
char   var1;        //Speicherplatzreservierung von 1 Byte
                    //Inhalt der Speicherstelle wird als Zahl interpretiert
                    //(Hinweis: ASCII-Zeichen sind Zahlen)
double var2=47.11;  //Speicherplatzreservierung von 8 Byte
                    //Inhalt der Speicherstellen wird als Gleitkommazahl
                    //interpretiert (bestehend aus je einer Ganzzahl für
                    //Mantisse und Exponent)
short arr[10];      //Speicherplatzreservierung von 10*2 Byte
                    //Inhalt der 10 Speicherstellen werden als Ganzzahl
                    //interpretiert
```

Die Funktionalität der Operatoren hängt vom Datentyp der Operanden ab. Der Plus-Operator angewendet auf ganzzahlige Operanden führt eine Ganzzahl Addition aus, angewendet auf gleitkomma Operanden eine Gleitkomma Addition. Ist einer der Operanden ein Zeiger, so wird eine Zeigeraddition angewendet. Für andere Datentypen ist der Plus-Operator nicht definiert, so dass der Compiler einen Fehler wirft. **Bei der Betrachtung eines Ausdruckes sollte dem genutzten Datentypen folglich eine besondere Aufmerksamkeit zuteilwerden**:

```
int vari=47;
vari+12;           //(int) + (int) → Ganzzahladdition
float varf=12.3;
3.1415F+varf;      //(float) + (float) → Gleitkommaaddition
int *ptr=&vari;
ptr+3;             //(int *) + (int) → Zeigeraddition
int arr[3];
arr+3              //(int (*)) + (int) → Nicht definiert
```

Das Ergebnis einer Operation hat ebenfalls einen Datentyp. Eine Ganzzahl Addition gibt als Ergebnis einen Ganzahl Datentyp zurück, eine Gleitkomma Addtition einen Gleitkomma Datentyp, usw.:

```
int vari1=47;
int vari2=11;
vari1 + 12 + vari2;     //((int) + (int))→(int) + (int)
40 < vari1 < 50;        //((int) < (int))→(int) < (int)
```

Wie Operatoren haben auch Funktionen Operanden (in Form von Übergabeparameter). Das Ergebnis eines Funktionsaufrufes wird ebenfalls in Form eines Datentyps zurückgegeben:

```
double add(int par1,par2) {return par1+par2;}
int vari1=47;
int vari2=11;
vari1 + add(var2,12);   //(int) + ((int),(int))→(int)
```

Sind die Datentypen der Operanden nicht 'kompatibel', so 'passt' der Compiler die Datentypen im Falle von Ganzahl und Gleitkomadatentypen per impliziter Regel an (siehe Kap.

Datentypen:Implizite Typumwandlung[12]). Über explizite Typumwandlung (siehe Kap.Datentypen:Explizite Typumwandlung[13]) kann ein Datentyp z.T. in einen anderen Datentyp gewandelt werden. In allen anderen Fällen ist C Typsicher, d.h. der Compiler gibt zumindest eine Warning aus, wenn die Datentypen nicht identisch sind und der Compiler keine implizite Regel anwenden kann.

Tipp:

- Compiler gibt bei Fehlermeldung oftmals die erwarteten und die vorhandenen Datentyp an. Wenn solch eine Fehlermeldung kommt, sollten sie sich a) überlegen, was wollten sie syntaktisch geschrieben haben und b) was haben sie in der Tat geschrieben. Ein Google nach einer Lösung des Compilerproblems löst zumeist nicht das Problem, sondern umgeht es!

Hinweis:

- Ein guter Programmierer sollte die Datentypen so wählen, dass:
  - die Datentypen zum Inhalt und zum Wertebereiches passen
  - keine Datentypkonvertierungen notwendig sind

  andernfalls besteht die Gefahr, dass hiermit Laufzeitfehler einprogrammiert werden.
- Wenn Datentypkonvertierungen notwendig sind, sollte die explizite Konvertierung genutzt werden

4.8 Abhängigkeiten bzgl. Rechnerarchitektur

Kenngrößen von Prozessoren sind unter anderem die Anzahl der Bits für den Datentransport und für Arithmetisch-/Logische-Operationen (→ Datenbreite) und die Anzahl der Bits für die Adressierung des Speichers (→ Adressierungsbreite). Die Datenbreite sagt dabei aus, welche max. Datenbreite ein Maschinenbefehl (zumeist in einem Taktzyklus) transportieren/berechnen kann. Werden programmtechnisch größere Datenbreiten benötigt, bedingt dies, dass hierzu mehrere Maschinenbefehle notwendig sind. Wenn kleinere Datenbreiten notwendig sind, so wird oftmals das Ergebnis abgeschnitten (sofern der Prozessor keine Befehle für den Umgang mit kleineren Datentyp beherrscht):

32-Bit Datentransport/Operation bei 32-Bit Datenbreite	64-Bit Datentransport/Operation bei 32-Bit Datenbreite	16-Bit Datentransport/Operation bei 32-Bit Datenbreite
`int32_t a32,b32;`	`int64_t a64,b64;`	`int16_t a16,b16;`

`// a32=b32;`
`mov  eax,b32`
`mov  a32,eax`
	`// a64=b64;`
	`mov  eax  ,b64`
	`mov  edx  ,b64+4`
	`mov  a64  ,eax`
	`mov  a64+4,edx`
		`//a16=b16;`
		`movz  eax,b16`
		`mov   a16,ax`

12 Kapitel 5.6.1 auf Seite 101
13 Kapitel 5.5 auf Seite 99

32-Bit Datentransport/Operation bei 32-Bit Datenbreite	64-Bit Datentransport/Operation bei 32-Bit Datenbreite	16-Bit Datentransport/Operation bei 32-Bit Datenbreite

`// a32=a32+b32;`
`mov edx, a32`
`mov eax, b32`
`add eax, edx`
`mov a32, eax`
	`// a64=a64+b64;`
	`mov ecx , a64`
	`mov ebx , a64+4`
	`mov eax , b64`
	`mov edx , b64+4`
	`add eax , ecx`
	`adc edx , ebx`
	`mov a64 , eax`
	`mov a64+4, edx`
		`// a16=16+b16;`
		`movzx eax, a16`
		`mov edx, eax`
		`movzx eax, b16`
		`add eax, edx`
		`mov a16, ax`

Die Adressierungsbreite gibt an, mit welcher Bitbreite der Speicher adressiert werden kann:

- Bei einem 32-Bit System können 2^{32}=4.294.967.295=4Gi Bytes/Speicherstellen adressiert werden
- Bei einem 16-Bit System können 2^{16}=65.536=64Ki Bytes/Speicherstellen adressiert werden
- Bei einem 64-Bit System können 2^{64}=18.446.744.073.709.551.616 =16Ei ~18Exa Bytes/Speicherstellen adressiert werden

Waren die Datenbreite und die Adressierungsbreite bei den ersten Prozessoren noch unterschiedlich, so sind diese heutzutage zumeist identisch. Bei einem 32-Bit Prozessor bedeutet dies, dass dieser max. 4Gi Bytes Speicherstellenstellen adressieren kann und Maschinenbefehle für den Datentransport und die Berechnung von 32-Bit Datenworten beinhaltet.

Die Datenbreite und auch die Adressierungsbereite haben direkten Einfluss auf die Programmiersprache:

- Wird defaultmäßig mit größerer Datenbreite gerechnet, als der Prozessor 'von Hause' aus unterstützt, so werden die erzeugten Maschinenprogramme größer und langsamer (Umgedreht bedeutet dies jedoch nicht, dass die Maschinenprogramme kleiner und schneller werden)
- Die max. Größe von Verbunddatentypen und Arrays ergibt sich aus der Adressierungsbreite des Prozessors

Die Intention der Programmiersprache C/C++ ist, die Prozessorarchitektur optimal zu nutzen. Dementsprechend sind diverse Eigenschaften der Sprache nicht fest spezifiziert, sondern von der Rechnerarchitektur abhängig.

4.8.1 Integer Datentyp

Intention von C/C++ ist, dass der Umgang mit dem Grunddatentyp 'int' möglichst direkt in einen Maschinensprachebefehl umgesetzt werden kann.

Bei einem 16-Bit Prozessor hat der Datentyp int daher eine Breite von 16-Bit und bei einem 32-Bit Prozessor eine Breite von 32-Bit (siehe Kap Datentypen:Ganzahl Datentypen[14]). Über vorangestellte Qualifier kann dieser Grunddatentyp verkleinert/vergrößert werden.

14 https://de.wikibooks.org/wiki/Programmieren%20in%20C%2FC%2B%2B%3A%20Datentypen%20%23%20Ganzzahlige%20Datentypen

4.8.2 Integer Arithmetik

Ganzzahlen haben nur einen begrenzten Wertebereich. Wird dieser bspw. durch eine Addition überschritten (d.h. das Rechenergebnis benötigt zur Darstellung mehr Stellen, als der Datentyp 'hergibt'), so gibt es zwei Vorgehensweisen:

- Arithmetischer Überlauf (siehe w:arithmetischer Überlauf[15]): Die nicht speicherfähigen Stellen werden verworfen, so dass i.d.R. ein falsches Ergebnis erzeugt wird!

Beispiel:

```
unsigned char a=254,b=3,c;
c=a+b;   //c ist (254+3)%256=1
c=a*b;   //c ist (254*3)%256=250
```

- Sättigungsarithmetik (siehe w:Sättigungsarithmetik[16]): Alle Operationen laufen innerhalb eines festen Intervalls ab, so dass Überläufe und Unterläufe auf dieses Intervall begrenzt werden:

Beispiel:

```
unsigned char a=254,b=3,c;
c=a+b;  //c ist MAX((254+3),255)=255
c=a*b;  //c ist MAX((254*3),255)=255
```

Die Spezifikation von C macht bezüglich der zu nutzenden Methodik keine Vorgaben, so dass das Handling von der ALU des Prozessors abhängt. Aus Komplexitäts- und Geschwindigkeitsgründen ist dies zumeist der arithmetische Überlauf. Nur bei Spezialprozessoren (z.B. DSP Digital Signal Processor) kommt z.T. die Sättigungsarithmetik zum Einsatz.

4.8.3 Gleitkomma Datentyp und Arithmetik

Für den Umgang mit Gleitkommazahlen bieten einige Prozessoren spezielle Recheneinheiten (w:Gleitkommaeinheit[17] Floating Point Units FPU) an. Sowohl die Darstellung von Gleitkommazahlen als auch das Verhalten bei Überlauf/Unterlauf ist damit Herstellerabhängig. Um auch diese Einheiten effektiv nutzen zu können, ist die C/C++ Spezifikation in diesem Bereich ebenfalls sehr offen.

4.8.4 Max. Speicherbedarf von Variablen

Die maximale Größe von Arrays/Strukturen/Unions ist in C/C++ von der zugrundeliegenden Rechnerarchitektur abhängig:

15 https://de.wikipedia.org/wiki/arithmetischer%20%C3%9Cberlauf
16 https://de.wikipedia.org/wiki/S%C3%A4ttigungsarithmetik
17 https://de.wikipedia.org/wiki/Gleitkommaeinheit

```
char a[100000];  //Speicherbedarf=100.000*1Byte=100.000 Bytes
                 //Bei 16-Bit Architektur KO
                 //Ab  32-Bit Architektur OK
struct size_16 {
    char a,b,c,d,e,f,g,h,
         i,j,k,l,m,n,o,p;
};
struct size_256 {
    struct size_16 a,b,c,d,e,f,g,h,
                   i,j,k,l,m,n,o,p;
};
struct size_4096 {
    struct size_256 a,b,c,d,e,f,g,h,
                    i,j,k,l,m,n,o,p;
};
struct size_65536 {
    struct size_4096 a,b,c,d,e,f,g,h,
                     i,j,k,l,m,n,o,p;
};
struct size_65536 var65536; //Speicherbedarf=65536 Bytes
                            //Bei 16-Architektur KO
                            //Ab  32-Architektur OK
```

Im Falle von globalen Variablen erzeugt der Compiler/Linker eine Fehlermeldung. Werden von diesen Datentypen Variablen als lokale Variablen angelegt, so erzeugt dies zur Laufzeit einen Stackoverflow.

4.8.5 Zeigervariable

Eine Zeigervariable ist eine Variable, die eine Speicheradresse speichert. Die Größe dieses Datentyps hängt von der Adressierungsbreite des Prozessors ab. Unabhängig von dem Datentyp, auf den die Zeigervariable zeigt, ergibt sich folgende Breite:

- 4-Byte bei einer 32-Bit Rechnerarchitektur
- 8-Byte bei einer 64-Bit Rechnerarchitektur
- 2-Byte bei einer 16-Bit Rechnerarchitektur

4.8.6 Datentyp size_t

Die maximale Größe von Arrays/Strukturen/Unions ist in C von der zugrundeliegenden Rechnerarchitektur abhängig. Sie ergibt sich aus der Adressierungsbreite. Da die Datenbreite des Datentyps int zumeist kleiner als die Adressierungsbreite ist, existiert in C/C++ der vorzeichenlose Datentyp size_t.

Alle Funktionen, welche den Speicherbedarf übergeben bekommen, bzw. zurückgeben (bspw. strlen(), memcpy(), sizeof()) , nutzen den Datentyp size_t:

```
//Prototypen
size_t strlen(char *str); //Die max. Stringlänge ist durch den
                          //adressierbaren Speicher vorgegeben
void *malloc(size_t size);
void *memcpy(void *dest, const void *src, size_t n);
```

Der Datentyp size_t ist in der Header-Datei stddef.h der Standard-C-Library per typedef gesetzt. Im Normalfall muss diese Header-Datei nicht explizit inkludiert werden, da bei Nutzung von z.B. strlen() diese bereits durch die Header-Datei string.h inkludiert wird:

```
typedef unsigned long long size_t;  //Bei einer 64-Bit Rechnerarchitektur
```

Hinweis:

- Der Datentyp size_t ist immer ein vorzeichenloser Datentyp (es gibt keine negativen Speicheradressen).
- Mit dem Datentype ssize_t wird eine vorzeichenbehaftete Variante zur Verfügung gestellt, welche hilfreich ist, wenn Speicheradresse subtrahiert werden sollen.
- Soll der Wert vom Datentyp size_t mittels printf() ausgegeben werden, so ist der 'Length Modifier' z und der 'Conversion Specifier' u zu nutzen:

```
size_t var=strlen("String");
printf("%zu",var);
```

4.9 Boolescher-Datentype/Operatoren

Ursprünglich gab es den booleschen Datentyp in der Programmiersprache C nicht. Mit C99 wurde zwar der Datentyp _Bool eingeführt, dies änderte jedoch nicht den Umgang mit Anweisungen und Operatoren, die einen booleschen Datentyp erwarten/zurückgeben!

An allen Stellen, an denen ein boolescher Datentyp erwartet wird (z.B. IF-Abfrage, bei booleschen Operationen) wird der Datentyp Ganzzahl, Gleitkommazahl oder Zeiger erwartet (= Scalar-Type) und ausgewertet. Der hierin gespeicherte Zahlenwert sagt aus, ob diese Zahl als True oder False zu interpretieren ist:

- False, wenn Ganzzahl gleich 0 oder Gleitkommazahl gleich 0.0 oder Zeiger gleich NULL
- True, wenn Ganzzahl ungleich 0 oder Gleitkommazahl ungleich 0.0 oder Zeiger ungleich NULL

Beispiel:

```
int    a=3;
double b=0.0;
int   *ptr=&a;
while(1);
if(a) {}
for(  ; b && ptr ; );
```

Operatoren, welchen einen booleschen Wert zurückliefern (Vergleichs-Operatoren und Boolesche-Operatoren), liefern ein Integer-Datentyp zurück:

- (int)0 für False
- (int)1 für True

Beispiel:

```
int a=3;
int b= (a==3);  //b=1
```

In den if()/for()/while() Anweisungen ist folglich kein explizites Konvertieren eines Ganzzahl-/Gleitkomma-/Zeigerwertes mittels des Vergleichs-Operators in einen Booleschen Wert notwendig:

```
int a;
double b;
if(a)         //entspricht if(a!=0)
if(!b)        //entspricht if(b==0.0)
while(1)      //entspricht while(True)
for(int lauf=1; lauf ; lauf--);
```

Umgedreht kann ein boolescher Wert oder auch das Ergebnis einer booleschen Operation direkt mit einer Ganzzahl- oder Gleitkommazahl verknüpft werden:

```
int a=3,b=2;
a=a+1 && 7 << 3  & a==3;
if( a==1  &  b==2)
if(3 < a < 9)    //Vorsicht, die Schreibweise lässt ein
                 //anderes Verhalten erwarten!
```

Mit C99 wurde der Datentyp `_Bool` eingeführt. Dieser entspricht weiterhin einem Integerdatentyp mit der Besonderheit, dass beim Zuweisen eines Wertes dieses auf true oder false geprüft wird und Variablen dieses Datentyps nur die Werte 0 oder 1 annehmen können:

```
_Bool  var1=12;
printf("var1=%d\n",var1);  //→ 1
float  var2=13.2;
var1=var2;
printf("var1=%d\n",var1);  //→ 1
var1--;  //Post/Pre Inkrement/Dekrement in C++ nicht auf Datentyp bool! möglich
printf("var1=%s\n",var1?"true(1)":"false(0)");  //→ false
```

Für den einfacheren Umgang mit dem 'neuen' Datentyp wurde in der Header-Datei stdbool.h die Makros `true` (als Austauschwert für die Ganzzahl 1) und `false` (als Austauschwert für die Ganzahl 0) definiert. Ergänzend wird hier der Datentyp `bool` als Alias auf `_Bool` gesetzt.

```
#include <stdbool.h>
bool foo(int par,int min,int max) {
if((par >= min) && (par <= max))
  return true;
else
  return false;
}
```

Hinweis:

- In C++ gibt es anstatt des Datentypes `_Bool` den Datentyp `bool`. Auch sind die beiden Konstanten `true` und `false` ohne zusätzliches Einbinden der Header-Datei vorhanden. Der weitere Umgang ist jedoch identisch zu C.

4.10 Literale/Konstanten

Zahlenwerte und Strings im Source-Code werden vom Compiler in binäre Zahlen umgerechnet. Über der Schreibweise/Syntax der Literale/Konstanten wird dem Compiler gesagt, in welchem Zahlenformat die Zahl auf Source-Code Ebene beschrieben ist und welchem Datentyp sie entspricht. Unabhängig wie die Konstanten im Source-Code beschrieben sind, werden die Zahlen intern als binäre Zahlen gespeichert. Beispiele, wie die Zahlen intern gespeichert werden:

```
int   var=17;   //0000  0000  0000  0000  0000  0000  0001  0001
short var=0x307;//                      0000  0011  0000  0111
char  var=017;  //                                  0000  1111
int   var=-1;   //1111  1111  1111  1111  1111  1111  1111  1111
float var=0.0;  //0000  0000  0000  0000  0000  0000  0000  0000
float var=1.0;  //0011  1111  1000  0000  0000  0000  0000  0000
```

4.10.1 Ganzzahl-Konstanten

Im Allgemeinen ordnet der Compiler einer numerischen, ganzzahligen Konstanten den kleinstmöglichen Datentyp beginnend vom Datentyp integer zu. Sobald der Wert der Konstanten den Wertebereich von integer übersteigt, wird der nächst größere passende Datentyp gewählt. Ergänzend gilt, dass alle Ganzzahl-Konstanten als vorzeichenbehaftete Ganzzahl-Konstanten angesehen werden!

Beispiele für integer Konstanten:

```
123
+123
-123          → Formal wird das Minus als Operator angesehen
                und die positive Zahl 123 negiert
0x123         → Konstante ist in hexadezimaler Schreibweise notiert
0123          → Konstante ist in oktaler Schreibweise notiert
0b011         → Konstante ist in binärer Schreibweise notiert
      nicht offizieller C-Syntax, wird aber von vielen Compiler unterstützt
      ab C23-Spezifikation offiziell im C-Syntax enthalten!
6'432'108     → Tausendertrenner
    Nur in C++ Spezifikation enthalten
    ab C23-Spezifikation auch im C-Syntax enthalten.
5123123123    → Wertebereich größer als 2^31 so dass automatisch
                zum nächst größeren Datentyp konvertiert wird
```

Mit den Suffixen L/l und LL/ll kann der Datentyp long und long long erzwungen werden:

```
12L           → Konstante wird in eine long Integer Konstante konvertiert
4711LL        → Konstante wird in eine Long Long Konstante konvertiert
```

Mit dem Suffix U/u kann die Konstante explizit auf unsigned gesetzt werden:

```
12U           → Konstante wird als vorzeichenlose Zahl gesepeichert
4711ULL       → Konstante wird in eine vorzeichenlose
                  Long Long Konstante konvertiert
```

4.10.2 Char-Konstanten

In einfachen Anführungsstrichen geschriebene ASCII Zeichen werden entsprechend der ASCII-Tabelle[18] als Zahlenwert (Datentyp int) dargestellt:

18 https://de.wikipedia.org/wiki/American%20Standard%20Code%20for%20Information%20Interchange%20

```
'A' → Integerkonstante 65
'a' → Integerkonstante 97
…
```

Über den Escape-Operator[19] (Zeichen Backslash '\') wird das/die dem ESCAPE Operator folgende Zeichen gesondert interpretiert. Die im Source-Code erhaltene Zeichenfolge wird im Computerprogramm als ein Zahlenwert (Datentyp int) gespeichert. Dies ist u.A. zur Darstellung von ' und " aber auch zur Darstellung der unteren 32 Zeichen der ASCII Tabelle notwendig (Siehe auch w:Steuerzeichen[20])

```
ESCAPE   Integer-  → Bedeutung / (Terminal)funktionalität
Sequence konstante
'\0'    =  0 → 0 Character = Stringendezeichen
'\a'    =  7 → Bell (erzeugt auf dem Terminal einen Ton)
'\b'    =  8 → Backspace (Bewegt den Cursor um eine Position nach links)
'\t'    =  9 → Horizontal Tab (Bewegt den Curso auf die nächste horizontale
               Tabulatorposition, Abstand ist Terminalspezifisch)
'\n'    = 10 → NewLine (Zeilenvorschub)
'\v'    = 11 → Vertial Tab
'\r'    = 13 → Carriage Return (Wagenrücklauf, Bewegt den Cursor an den
               Zeilenanfang)
'\"'    = 34 → Double Quote
'\''    = 39 → Single Quote
'\?'    = 63 → Question mark
'\\'    = 92 → Backslash
'\f'    = 12 → FormFeed (Erzeugt auf dem Terminal/Drucker
                         einen Seitenvorschub)
'\ooo'  = 0ooo → Oktal, wobei o Digits von 0..7 sein müssen
'\xdd'  = 0xdd → Hexadezimal, wobei d Digits von 0..9 A..F sein müssen
'\udddd'       Unicode mit 4 Digits
'\Uddddddddd'  Unicode mit 8 Digits
```

Beispiel

```
'A' == 65 == '\101' == '\x41'   == '\u0041'  == '\U00000041'
```

Über Prefixe können alternative Zieldatentypen/Konvertierungen erzwungen werden:

```
u'b'      → unicode 16    (nur C++)
U'b'      → unicode 32    (nur C++)
L'b'      → wchar_t
```

Hinweis

- Die ASCII-Tabelle basiert auf einer 7-Bit Zeichenkodierung, so dass bspw. die Umlaute 'ä', 'ö' und 'ü' dort nicht kodiert sind. Sollen auch diese Zeichen genutzt werden, so empfiehlt sich der Datentyp wchar_t

4.10.3 Gleitkommakonstanten

Alle Gleitkommakonstanten sind ohne weitere Angaben vom Datentyp double!

19 https://de.wikipedia.org/wiki/Escape-Sequenz
20 https://de.wikipedia.org/wiki/Steuerzeichen

```
555.12
555.        → 555.0
.1          → 0.1
4e2         → 400.0
0x22.11p13  → Konstante in Hexadezimalen Schreibweise (ab C99)
```

Über Suffixe kann ein alternativer Datentyp erzwungen werden:

```
3.0F        → Datentyp float
3.0l        → Datentyp long double
```

Hinweis:

- Aufgrund der unterschiedlichen Basis zwischen dem menschlichen Zehner-System und dem computerinternen Binär-System entstehen insbesondere bei den Nachkommastellen Rundungsfehler. Die Konstante 0.1f wird durch die Wandlung intern als 1.00000001490116119384765625e-1 dargestellt
- Die C-Spec verlangt, dass die binäre Repräsentation von Gleitkommazahlen zur Laufzeit (z.B. durch die Funktion atof() strtod() scanf(”%f”)) identisch zur der zur Compilezeit ist. Beispiel:

```
if(3.14==atof("3.14"))   printf("muss erfüllt sein");
//3.14 wird vom Compiler in ein Binärwert gewandelt
//Der String "3.14" wird zur Laufzeit mittels atof in ein Binärwert gewandelt
//Beide Wandlungsergebnisse müssen identisch sein
```

4.10.4 Konstanten Ausdrücke

Ergänzend zu den Zahlenwerten werden auch Ausdrücke durch den Compiler ausgewertet. Als Voraussetzung für die Ausdrücke gilt, dass alle Operatoren Literale/Konstante sein müssen, so dass der Compiler die Berechnung durchführen kann. Als Operatoren können alle im normalem Syntax erlaubten Operatoren (z.B.: +,-,*,/,&,&&,|,||,»,«,==,!=) genutzt werden.

Beispiel:

```
int var1=7*3+5;           → wird vom Compiler durch 26 ersetzt
int var2=(4710+(8-1))&-8; → wird vom Compiler durch 4712 ersetzt
var1=4*4*var1+77*(77%3);  → wird vom Compiler durch 16 * var1 + 154 ersetzt
```

Hinweis:

- Konform zur C-Spezifikation bieten einige Compiler weitere Operatoren aus der math.h Library an (z.B. sin(), cos(), ...). Bestehen die Aufrufparameter aus Konstanten, so ersetzt der Compiler den Ausdruck durch die 'Ergebnis'-Konstante:

```
int var3=sin(1.0)*10;   →wird vom Compiler durch 8 ersetzt
```

- Konstanten Ausdrücke werden durch den Präprozessor ausgewertet, so dass dieser die Ergebnisse ebenfalls nutzen kann:

```
#define BUF_SIZE 512
#if (BUF_SIZE & (BUF_SIZE -1)) != 0
#endif
```

4.10.5 String Konstanten

Stringkonstanten erzeugen ein Array von Characters. Mit dem abschließenden Hochkommata wird das Array um ein Stringendezeichen ergänzt:

```
"abc"       →  (char []) {'a','b','c','\0'}
```

Adjacent Strings Konstanten werden vom Compiler zu einer String-Kontante zusammengefasst, so dass nur ein Stringendezeichen vorhanden ist:

```
"xxx" "yyy"  →  (char []) {'x','x','x','y','y','y','\0'}
```

Über Prefixe können alternative Zieldatentypen/Konvertierungen erzwungen werden:

```
L"string"    →   (wchar_t[7]){L's',L't',L'r' …)
```

Über Postfixes können in C++ Objekte erzeugt werden:

```
"string"s    → erzeugt einen String vom Datentyp std::string
"string"sw   → Erzeugt einen String vom Datentyp std::string_view
```

String Konstanten werden Compilerintern über unnamed Variablen realisiert, welche als globale Variable angelegt werden und per default const sind:

```
printf("Wert=%d",4);
//entspricht
const char unnamed1[]="Wert=%d";
printf(unnamed1,4);

//String Konstanten sind per Default Konstant
char *str="hallo";  //Typkonflikt!, da "hallo" vom Datentyp const char[] ist
str[0]='H';         //Laufzeitfehler, da nun schreibender Zugriff auf im
                    //Readonly Speicher angelegt Variable.
```

Für doppelte String Konstanten wird nur einmal Speicher reserviert:

```
const char *str1="hallo";
const char *str2="hallo";
if(str1==str2) printf("Identische Startadressen");
```

4.11 Initialisierungsliste / Compound Literal

Arrays, Strukturen und Unions bestehen zumeist aus mehreren Elementen. Zur Initialisierung solcher Variablen werden die Initialisierungswerte in geschweiften Klammern als Initialisierungsliste zusammengefasst.

`Syntax: {Initialisierungsliste}`

Für eine Initialisierungsliste wird kein Speicher vorgehalten. Vielmehr wird die Variable direkt mit dem Inhalt der Liste 'gefüllt'. Der Datentyp der Initialisierungsliste ergibt sich aus dem Datentyp der Variable:

```
int arr[5] = { 1,2,3,4,5 };
struct xyz { int x,y,z;};  //Definition einer Struktur
struct xyz var={1,2,3};
```

Wird der Initialisierungsliste ein Datentyp in runden Klammern vorangestellt, so wird hiermit eine 'unnamend' Variable/Objekt, ein Compound Literal erstellt:

`Syntax: (type) {Initialisierungsliste}`

Compound Literale können überall dort genutzt werden, wo andernfalls vorab initialisierte Variablen des gleichen Typs eingesetzt würden. Bei einmaligen Gebrauch solcher Variablen erspart man sich hierüber die Definition der Variablen:

```
int *p=(int []){2,4};
//entspricht
int unnamed0[]={2,4};
int *p=unnamed0;
drawline((struct point){1,2},(struct point){2,3});
drawlinep( &(struct point){1,2},&(struct point){2,3});
//entsprechen
struct point unnamed1={1,2},unnamed2={2,3};
drawline(unnamed1,unnamed2);
drawlinep(&unnamed1,&unnamed2);
```

Der Gültigkeitsbereich eines Compound Literals entspricht der von Variablen. Wird ein Compound Literal außerhalb eines Block-Sopes angelegt, so entspricht sie einer globalen Variablen, andernfalls einer lokalen Variable.

Hinweis:

- Compound-Literal sind per Default keine Konstanten, können also zur Laufzeit geändert werden (siehe auch 7.8 Const Type Qualifier):

```
float *arr=(float []){1.1,2.2,3.3};  //Kein Typkonflikt
arr[0]=47.11;   //Änderung des 'Konstanten' arr[0] ist möglich
```

4.12 Variableninitialisierung

Mit der Definition einer Variable kann diese optional initialisiert werden. Der tatsächliche Zeitpunkt der Variableninitialisierung (zur Laufzeit oder zur Compilezeit) und damit die

Art des Initialisierungswertes / der Inhalt der Initialisierungsliste hängt von der Gültigkeit/Sichtbarkeit der Variablen ab (siehe Gültigkeit/Sichtbarkeit von Variablen[21]).

4.12.1 Globale + Static lokale Variablen

Globale und statisch globale Variablen werden in einem während der Laufzeit des Programms festen Speicherbereich gehalten. Zum Startzeitpunkt des Programms ist der Speicherplatz solch einer Variable mit dem Initialisierungswert gefüllt, so das bei erstmaliger Nutzung dieser Variablen diese ihre Initialisierungswerte enthalten. Die Belegung des Speicherplatzes erfolgt durch den Compiler, so dass als Initialisierungswerte /Inhalt der Initialisierungsliste nur Konstanten oder Konstanten-Ausdrücke (siehe Literale/Konstanten[22]) erlaubt sind:

```
int g=1;                          //Speicherstelle der Variable g wird
                                  //mit 1 belegt
short array[3]={1,1+1, sin(0.0)}; //Speicherstellen des Arrays wird
                                  //mit 1, 2 und 0 belegt werden
```

Nicht initialisierte globale und static lokale Variablen werden mit 0 initialisiert! D.h. die Speicherstellen dieser Variablen werden mit dem Wert 0 vorbelegt:

```
int f;          //f hat den Wert 0
double arr[10]; //Alle Elemente von arr werden mit 0.0 initialisiert!
```

4.12.2 Lokale Variable

Der Speicherplatz für lokale Variable wird auf dem Stack zur Verfügung gestellt. Bei jeder Ausführung der Definition wird Speicher vom Stack nach dem LIFO Prinzip[23] reserviert und im Anschluss Maschinencode zur Initialisierung der Speicherstellen ausgeführt (sofern Variable einen Initialisierungswert hat). Zum besseren Verständnis kann man sich die Variableninitialisierung als zwei Anweisungen vorstellen: a) der Definition der Variablen und b) der Zuweisung des Initialisierungswerter zur Variablen:

```
int var=sin(b);
//entspricht
int var;     //a)Variablendefinition/Speicherplatzreservierung
var=sin(b);  //b) Initialisierung über separate Zuweisungs-Operation
             //d.h. es werden Maschinensprachebefehle erzeugt, welche
             //den Inhalt der Variablen b holt, anschließend die sin()
             //Funktion aufruft und dessen Rückgabewert der Variablen
             //var zuweist.
```

Folglich können die Initialisierungswerte/Inhalte der Initialisierungsliste aus beliebigen Ausdrücken stehen und müssen keine Konstanten oder Konstanten Ausdrücke sein:

```
void foo(float f,float g) {
  float summe=f+g;
  float arr[3] = {f+f,g-f,sin(f)}; //Initialisierungswerte
                                   //werden zur Laufzeit berechnet
}
```

21 Kapitel 4.5 auf Seite 37

22 Kapitel 4.10 auf Seite 53

23 https://de.wikipedia.org/wiki/Last%20In%20%E2%80%93%20First%20Out

Aufgrund der Ausführung von Maschinencode bedingt die Initialisierung Rechenzeit.

Wird eine Variableninitialisierung mit einer switch oder goto Anweisung übersprungen, so wird zwar die Variable angelegt (Speicherplatz reserviert), jedoch die Maschinenbefehle zur Initialisierung dieser Variablen nicht ausgeführt (siehe auch Switch-Anweisung[24] und Label + Goto[25]). Die Variable enthält dann einen Zufallswert:

```
switch(value) {
  case 0:
    int lok=7;
    printf("%d",lok);  //OK, lok beinhaltet den Wert 7
    break;
default:
    printf("%d",lok);  //KO, Variable lok vorhanden,
                       //aber nicht initialisiert. D.h. Ausgabe einer
    break;             //Zufallszahl
}
```

Nicht initialisierte lokale Variablen werden nicht initialisiert, d.h. keine zusätzlichen Maschinenbefehle zum Initialisieren erzeugt. Der Variablenwert entspricht dann einem 'Zufallswert' oder korrekter gesagt dem Inhalt der Variablen, welcher zuvor der Speicherstelle zugewiesen war:

```
int foo(void) {
  int var_foo;    //Variable var_foo bekommt den identischen
  return var_foo; //Speicherbereich wie var_far zugwiesen
}
int far(void) {
  int var_far=7;
  return var_far;
}
printf("%d\n",foo());   //Zufallswert (0, da Stack mit 0 initialisiert ist)
printf("%d\n",far());   //Stack belegen
printf("%d\n",foo());   //Zufallswert (7, da Speicherstelle zuvor mit 7 belegt
 wurde
```

Hinweis:

- Bei erstmaligem Funktionsaufruf kann eine nicht initialisierten Variablen den Wert 0 beinhalten (Stackframe wird mit Programmstart auf 0 gesetzt). Beim nächsten Aufruf der Funktion ist zumeist die Speicherstelle der lokalen Variablen mit anderen Werten gefüllt worden, so dass nun ein anderer 'Zufallswert' als 'Initialisierungswert' enthalten ist.

4.12.3 C++

In C++ ist ergänzend die Initialisierung einer Variablen über () möglich. Bei Objekten wird hierüber der zugehörige Konstruktor aufgerufen. Bei primitiven Datentypen entspricht der Wert in den Klammern dem Initialisierungswert:

```
int variable(10);       //Bei primitiven Datentype entspricht dies
                        //int variable=10;
struct test {
  int a;
  test(int par) {a=par;}  //Konstruktor der Struktur test
```

24 Kapitel 4.16.6 auf Seite 79
25 Kapitel 4.16.6 auf Seite 79

```
};
struct test test(4);   //Konstruktor test(int par) wird aufgerufen
struct test test(4,5); //KO, kein passender Konstruktor vorhanden
```

4.12.4 Sonstiges

Folgende weitere Sachverhalten gilt es bzgl. Variableninitialisierung zu berücksichtigen:

- Nach C-Spezifikation ist eine Initialisierung von Variablen nicht notwendig. Es liegt folglich kein Fehler vor, wenn auf eine nicht initialisierte Variable lesend zugegriffen wird. Ungeachtet der C-Spezifikation überprüft der Compiler mit dem Schalter '-Wall' diesen Sachverhalt und gibt eine Warning aus.
- Deklarationen/Prototypen können keinen Initialisierungswert enthalten

```
extern int var=7;    //Fehlerhaft
```

- Const und lokale Static Variablen sollten initialisiert werden, auch wenn C in diesem Fall keinen Fehler meldet!

```
const int konstante;  //Syntaktisch OK, Initialisierungswert fraglich!
static int statisch;
```

4.13 Anweisung (Statement) & Expression

Eine Anweisung (Statement) stellt entsprechend dem Wikipedia Artikel Anweisung[26] *"ein in der Syntax einer Programmiersprache formulierte einzelne Vorschrift dar, die im Rahmen der Abarbeitung des Programms auszuführen ist"*. Entsprechend der C-Spezifikation [C11 6.8] kann eine Anweisung aus:

labled-statement: Eine Sprungmarke für die Switch- und die Goto-Anweisung

compound-statement: siehe nachfolgendes Kapitel

expression-statement: siehe nachfolgendes Kapitel

selection-statement: Eine if() oder switch() Anweisung

iteration-statement: Eine while(), do while() oder eine for() Anweisung

jump-statement: Eine goto, break, continue oder return Anweisung

Die einzelnen Statements können dabei aus weiteren Statements bestehen.

4.13.1 Ausdruck (Expression)

Entsprechend der C-Spezifikation [C11 6.5] ist ein Ausdruck eine Folge von Operatoren und Operanden, die die Berechnung eines Werts angibt, ein Objekt oder eine Funktion

26 https://de.wikipedia.org/wiki/Anweisung_%28Programmierung%29%20

bezeichnet, Nebenwirkungen erzeugt oder eine Kombination davon ausführt. Die Wertberechnungen der Operanden eines Operators werden vor der Wertberechnung des Ergebnisses des Operators sequenziert (siehe auch Expression[27]).

Beispiele:

```
a=sqrt(a*a+b*b);
//Ausdruck mit Nebenwirkungen
int func(int par) { a++; return par*2;}
b=func(1)*func(2);
//Ausdruck mit undefinierten Verhalten
a=++a + 1;

a=for(lauf=10;lauf>0;lauf--);  //for-Anweisung ist kein Ausdruck
                               //d.h. liefert keinen Wert zurück
```

4.13.2 Discarded Value

Discarded Value sind Ausdrücke, dessen Ergebnisse nicht verwendet werden (also keiner Variablen zugewiesen werden). Ergebnisse von Ausdrücken, die nicht verwendet werden, führen nicht zu einer Fehlermeldung:

```
int a=7;
a;          //Variable a wird gelesen, der Wert jedoch verworfen
7;          //Konstante wird verworfen
a+7;        //Ergebnis wird Berechnet und verworfen
printf("hallo"); //Printf hat einen Rückgabewert, dessen Wert verworfen wird
```

Hinweise:

- Soll bewusst der Rückgabewert verworfen werden, so kann/sollte mit dem explizten Cast auf (void) der Rückgabewert explizit 'verworfen' werden. Dies ist u.A. hilfreich bei Variablen, die (noch) nicht benutzt werden und der Compiler die Warning ”unused Variable” ausgibt:

```
(void)a;        //Zur Vermeidung von Compiler-Warnings "Unused Variable"
(void)(a+7);    //hier eine explizte 'Ansage' der Verwerfung des Wertes
(void)func();
```

- Mit der Anweisung [[nodiscard]] (ab C++17, ab C23) wird der Compiler angewiesen, dass der Aufrufer einer Funktion mit Rückgabewert der Rückgabewert entgegenzunehmen ist. Wird der Rückgabewert verworfen, so gibt der Compiler eine Fehlermeldung aus:

```
    [[nodiscard]] int  Client(int socket);
    Client(5);     ////Fehlermeldung, da Rückgabewert nicht verwendet wird
```

- Java kennt Dicarded Values nicht. Ausnahme, es handelt sich um einen Funktionsaufruf, dessen Rückgabewert nicht genutzt wird.

27 https://de.wikipedia.org/wiki/lang%3Den

4.13.3 Sequence Point

Ein Sequenzpunkt (u.A. das abschließende Semikolon am Ende der Anweisung) ist nach dem Wikipedia Artikel Sequence Point[28] definiert als der Punkt in der Ausführung eines Computerprogramms, an dem garantiert ist, dass alle Nebenwirkungen früherer Bewertungen durchgeführt wurden und noch keine Nebenwirkungen von nachfolgenden Bewertungen durchgeführt wurden.

Sequenzpunkte sind beispielsweise dann von Interesse, wenn dieselbe Variable innerhalb eines einzelnen Ausdrucks mehr als einmal geändert wird und die Ausführungsreihenfolge der Operationen nicht sauber spezifiziert ist.

Ein oft zitiertes Beispiel ist der C-Ausdruck 'i=i++;'. Der Sequence Point garantiert, dass alle Operationen (hier Zuweisung und Postinkrement) ausgeführt sind. Jedoch ist die Ausführungsreihenfolge dieser beiden Operationen nicht spezifiziert. In Abhängigkeit des Zeitpunktes der Ausführung des Postinkrementes (vor oder nach der Zuweisung) kann i unterschiedliche Werte annehmen. In C und C++ führt die Auswertung eines solchen Ausdrucks zu undefiniertem Verhalten. Andere Sprachen, wie z. B. C#, definieren den Vorrang des Zuweisungs- und des Inkrementoperators so, dass das Ergebnis des Ausdrucks i=i++ garantiert ist.

Weitere Sequence Points:

- Beim ?-Operator stellt sowohl das ? als auch der : ein Sequenz Point dar. Zunächst muss der linke Ausdruck vollständig bearbeitet werden, bevor einer der beiden Bedingungen bearbeitet wird
- Bei der logischen UND / ODER Funktion stellen der Operator ein Sequenz Point dar. Erst wird der linke Operand vollständig ausgeführt und ausgewertet, bevor (ggf.) der rechte Operand ausgewertet wird (siehe Logische Verknüpfungen[29])
- Beim Komma-Operator stellen der Komma ein Sequence-Point dar (siehe Komma-Operator[30])
- und weitere (siehe Appendix der C-Spec)

Operationen, bei denen die Ausführungsreihenfolge nicht spezifiziert ist:

- Bei Funktionsaufrufen ist die Abarbeitungsreihenfolge der Parameter nicht spezifiziert, so dass bspw.:

```
//Aufrufruf einer Funktion, deren Parameter sich aus Funktionsaufrufen
//ergeben
(*pf[f1()]) (f2(), f3() + f4());   //pf -> Array von Funktionszeigern
//Die Funktionen f1,f2,f3 und f4 können in einer beliebigen Reihenfolge
//aufgerufen werden
```

28 https://de.wikipedia.org/wiki/lang%3Den
29 Kapitel 4.16.4 auf Seite 77
30 Kapitel 4.16.2 auf Seite 75

- Per Postinkrement/-dekrement Operator sagt aus, dass für die eigentliche Operation der Wert der Variablen genutzt wird und im Anschluss die Variable erhöht / erniedrigt werden soll. Nach C-Spezifikation [C11 6.5.2.4] muss dies spätestens vor dem nächsten Sequence Point erfolgt sein. Der genaue Zeitpunkt innerhalb des Sequence Points ist jedoch nicht vorgegeben:

```
int a=7;
int b=a++ + a;   //Ergebnis Abhängig, ob das Postinkrement vor der Addition
                 //oder als letztes ausgeführt wird.
a=a++ + 1;      //dito
```

4.13.4 Block / Compound-Statement

Ein Block oder Compound-Statement fasst Vereinbarungen und Anweisungen zu 'einer' Anweisung/Statement zusammen. Es kann überall dort eingesetzt werden, wo nur eine Anweisung erlaubt ist.

Syntax: {Declaration-or-statement-list/textsuperscript$_{\mathrm{OPT}}$}

Ein Block wird nicht wie bei einer Anweisung mit einem Semikolon abgeschlossen (Das angehängte Semikolon wäre dann eine separate leere Anweisung!).

Ein Block hat blockweite Sichtbarkeit (siehe Blockweite Sichtbarkeit[31]), so dass hier definierte Variablen nur hier gültig sind. Deklarationen sind ebenfalls nur in dem Block gültig.

```
for(..) Anweisung;   //For-Schleife führt nur diese eine Anweisung aus
for(…) {Block}       //For-Schleife führt Block (als eine Anweisung) aus

if(..) Anweisung; else Anweisung;  //IF-Anweisung führt nur 'eine'
                                   //Anweisung aus
if(..) {Block} else {Block}        //IF-Anweisung führt Block aus
if(  ) {Block}; else {Block};      //Semikolon ist eine separate
                                   //Leer-Anweisung, so dass else
                                   //nicht zur IF-Anweisung zugeordnet
                                   //werden kann (2 Anweisungen vor Else)
void foo(void) {
  int var1;
  {  //Block innerhalb von Block als Strukturierungswerkzeug
    int var2;
    struct xyz{int x,y,z};  //Defintion eines Datentyps im Block
    struct xyz var3;
  }
  struct xyz var4; //KO, Datentyp hier nicht mehr bekannt
}
```

Die GNU-C Reference (nicht C++) erlaubt es, Funktionen innerhalb von Blöcken (und damit auch innerhalb von Funktionen) zu definieren. Diese innere Funktion steht dann nur in diesem Block zur Verfügung und kann ergänzend auf Variablen des umgebenden Blockes zugreifen:

```
void foo(void) {
  {
    int label=1;
```

31 Kapitel 4.5.3 auf Seite 39

```
    //Funktion innerhalb einer Funktion
    void dummy(void){
        label++;  //Zugriff auf Variablen des umschließenden Blockes
    }
    dummy();      //Aufruf der nur in diesem Block gültigen Funktion
  }
  dummy();        //KO, Funktion hier nicht mehr gültig
}
```

Hinweise:

- **Blöcke können/sollten als Strukturierungswerk genutzt werden. Dem Leser von Source Code kann auf dieser Weise vermittelt werden, dass diese Anweisungen 'zusammen' zu lesen sind.**
- Im Gegensatz zu Java, in welchem innerhalb eines Blockes kein Variablenname genutzt werden darf, die außerhalb des Blockes existiert, ist dies in C erlaubt. Die äußere Variable steht damit innerhalb des Blockes nicht mehr im Zugriff.

```
int a=3;
int main(void){
  int a=7;  //Globale Variable ist ab hier nicht mehr sichtbar/zugreifbar
  {
    int a=9;   //Innere Variable überdeckt äußere Variable
  }
}
```

4.13.5 Embedded Statement

Embedded Statements sind eine GNU-C Erweiterung (siehe GNU-C-Manual 3.18[32]). Da sie vom Autor als sehr nützlich angesehen wird und auch von diversen Compilern und auch in C++ unterstützt wird, wird diese hier beschrieben.

Ein Block fasst mehrere Anweisungen zu einer Anweisung zusammen, hat jedoch kein Rückgabewert. Ein Embedded-Statement entspricht einem Block mit der Ergänzung, dass dieser einen Rückgabewert hat, so dass ein Embedded-Statement überall dort eingesetzt werden kann, wo eine Expression erwartet wird.

`Syntax: ({Declaration-or-statement-list`$_{\mathrm{OPT}}$`})`

Der Rückgabewert wird nicht über eine return-Anweisung angegeben, sondern entstammt aus der letzten Anweisung innerhalb der Liste:

```
int var;
var={int par=7;}                //KO Block hat keine Rückgabewert.
var=({int par=7; par++; par;}); //OK Embedded-Statement hat einen
                                //Rückgabewert, der sich aus der
                                //letzten Anweisung 'Par;' ergibt
                                //Hier 8.
```

Ein Embedded-Statement ähnelt folglich einem Funktionsaufruf mit dem Unterschied, dass a) in der Tat kein Funktionsaufruf stattfindet und b) der Rückgabewert nicht mit return, sondern über die letzte Anweisung zurückgegeben wird. Der Anwendungsfall ist dort zu

32 https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

finden, wo eine Expression erwartet wird und mehrere Anweisungen zum Ermitteln der Expression notwendig sind:

```
int var1=8;
int b=({int d=var1++;d>10?10:d;});
if(  ({int d=b;b=var1;b=d;}) > 1)
```

Hinweise:

- Ein Embedded-Statement ist insbesondere bei Nutzung von Makros (siehe Kap Präprozessor[33]) hilfreich
- Mittels eines Embedded-Statement kann eine anonyme Funktion/Lambdafunktion dargestellt werden:

```
void test(int (*fptr)(int),int arr[10]) {
  for(int lauf=0;lauf<10;lauf++)
    arr[lauf]=fptr(arr[lauf]);
}
int main(int argc, char *argv[]) {
  int arr[]={1,2,3,4,5,6,7,8,9,10};
  //Aufruf mit einer anonymen Funktion
  test( ({int func(int a) {return a%2?a:0;} ; func;})
        ,arr);
}
```

4.14 Funktionen / Prozeduren

Funktionen/Prozeduren sind ein Strukturierungsmerkmal, so dass Teile der Funktionalität des Programmes wiederverwendbar sind. C/C++ unterscheidet nicht zwischen Funktionen (haben einen Rückgabewert, können damit in Ausdrücken verwendet werden) und Prozeduren (haben keine Rückgabewert, resp. Rückgabetyp void).

Mit jedem Aufruf einer Funktion wird für den zugehörigen Block ein eigener Stack-Kontext angelegt, so dass Funktionen sich selbst aufrufen können (Rekursion). Jeder Funktionsinstanz hat somit seinen eigenen lokalen Variablenbereich, teilen sich jedoch die globalen und die statisch lokalen Variablen.

Mit dem Aufruf einer Funktion können Parameter übergeben werden, wobei die Übergabeparameter vom Aufrufer (Caller) durch den Aufruf in die lokalen Variablen des Aufgerufenen (Callee) kopiert/zugewiesen werden (Call by Value). Wird ein Zeiger übergeben, hat der Callee über die Dereferenzierung des Zeigers die Möglichkeit, die Variablen des Caller zu verändern (Call by Reference) (siehe Kap. Zeiger:Zeiger bei Funktionsaufrufen).[34]).

Hinweise:

- Die Klammerung hinter dem Funktionsnamen (`foo()`) zeigt an, dass es sich um einen Funktionsaufruf (Verzweigung der Programmausführung zu der Startadresse der Funktion) handelt

33 Kapitel 9 auf Seite 203

34 https://de.wikibooks.org/wiki/Programmieren%20in%20C%2FC%2B%2B%3A%20Zeiger%20%23Zeiger%20bei%20Funktionsaufrufen%20

```
void foo(void) {printf("foo called");};
foo();  //Klammern hinter dem Funktionsnamen bedeuten Aufruf der Funktion
```

- Die Nutzung des Funktionsnamens ohne Klammerung (`foo`) gibt die Startadresse der Funktion zurück. Die Startadresse kann in einem Zeiger auf Funktionen gespeichert werden (siehe Kap. Zeiger:Zeiger auf Funktionen[35]).

```
void foo(void) {printf("foo called");};
void (*fptr)(void)=foo;
```

- Deklaration/Definition vor dem ersten Aufruf der Funktion 'notwendig', damit Compiler die Übergabeparameter entsprechend bereitstellt und ggf. eine implizite Typkonvertierung durchführt (dito Rückgabewert). Die Angabe der Variablennamen können bei der Deklaration entfallen
- Die fehlende Angabe der Übergabeparameter bedeutet in C (nicht C++), dass diese Funktion mit beliebigen Übergabeparametern aufgerufen werden kann und der Compiler keine Typprüfung durchführt:

```
int func() {...}
//kann wie folgt aufgerufen werden
func();
func(4);
func("hallo Welt",4.7);
```

→ unbedingt vermeiden

Hat die Funktion keinen Übergabewert, so wird dies in C mit void gekennzeichnet

- In C++ bedeutet die Schreibweise ohne Datentypbeschreibung, dass keine Parameter erwartet werden
- In der Parameterliste von Funktionen dient in der GNU-C Spezifikation das Semikolon zum Trennen zwischen Deklaration von Übergabeparameter und den Übergabeparametern. Alle Übergabewerte links vom Semikolon sind (Forward)deklarationen:

```
void test1(int a; int b,int a) {
//      Forward;  Parameter
  printf("a=%d b=%d\n",a,b);
}
//Aufruf
test1(3,7);    //a=7 b=3

//Sinnvoll im Umgang mit Variable Length Arrays
void test2(int dim; const char str[dim], int dim);
//Aufruf
test2("hallo",6);
```

35 Kapitel 7.14 auf Seite 168

4.14.1 Return-Anweisung

Mit der Return-Anweisung wird eine Funktion (vorzeitig) beendet. Der optionale Parameter dient der Angabe des Rückgabewertes der Funktion an den Aufrufer. Der Datentyp des Parameters ergibt sich aus dem Rückgabedatentyp der Funktion.

`Syntax: return expression`$_{\text{OPT}}$

Der Rückgabewert expresion sagt aus, dass hier nicht nur Konstanten oder Variablen stehen können, sondern beliebige Ausdrücke erlaubt sind. Der resultierende Datentyp des Ausdruckes muss letztendlich dem Rückgabedatentyp der Funktion entsprechen:

```
int foo(double a, double b) {
  return (int)(a*a+b*b);
}
```

Hinweise:

- Bei nicht void-Funktionen ist eine return-Anweisung mit einer Expression entsprechend dem Datentyp der Funktion zwingend notwendig:

```
int func(void) {
  return 5.1;     //Datentyp wird zuvor int konvertiert
}
double a=func();  //int wird in double konvertiert
```

- Bei einer void-Funktion ist die return-Anweisung ohne Expression optional. Hier entspricht der schließende Block der Return-Anweisung

```
void func(char *str) {
  if(str==NULL)
    return;
  printf("%s",str);
}
```

Sollen mehr als ein Rückgabewert an den Aufrufer zurückgegeben werden so empfiehlt sich:

- die Rückgabewerte in einer Struktur zu packen (siehe Datentypen:Struktur/Verbundtyp[36])
- die Rückgabewerte über die Aufrufparameter als Call-by-Reference zu übergeben (siehe Zeiger:Zeiger bei Funktionsaufrufen[37])

Rückgabewert aus Datentypsicht

Aus Datentypsicht kann der Funktionsname inkl. den Parameter(n) in einem Ausdruck mit dem Datentyp des Rückgabewertes der Funktion ausgetauscht werden:

36 Kapitel 5.7 auf Seite 105
37 Kapitel 7.12 auf Seite 163

```
int foo(void) {return 4711;}
int var;
var=1+foo();  //Aus Datentypsicht (int)=(int)+(int)
```

Daraus ergibt sich, dass bei Arrays/Strukturen/Unions direkt auf einzelne Komponenten zugegriffen werden kann, ohne diese zuvor einer Variablen zuzuweisen:

```
struct xyz{int x,y,z;};
struct xyz foo(int par) {
  struct xyz var;
  var.x=var.y=var.z=par;
  return var;
}
if(foo(12).x == 12) printf("Identisch\n");
```

Fehlerrückgabe

Viele Funktionen können unter gewissen Umständen nicht korrekt ausgeführt werden. Z.B. aufgrund dessen, dass:

- der benötigter Speicher vom Heap nicht bereitgestellt werden kann
- Übergabeparameter außerhalb des zulässigen Bereiches liegen
- angeforderte Ressource (Datei, Geräte, ...) nicht vorhanden/verfügbar sind

Solche Fälle gilt es dem Aufrufer mitzuteilen, da in diesen Fällen die vom Aufrufer geforderte Funktionalität nicht erfüllt ist und der Aufrufer ggf. nicht weiterarbeiten kann. Zur Mitteilung des Fehlers sollte vorrangig der Rückgabewerte genutzt werden. (auch wenn in C++ mit Exception eine Alternative hierzu bereitsteht)

```
char *strdup(char *src) {
  char *dst;
  if((dst=malloc(strlen(src)+1))==NULL)
    return NULL;
  strcpy(dst,src);
  return dst;
}
```

Die Standard-C Library-Funktionen geben in diesem Fall zumeist ein -1 (Rückgabewert Ganzzahl) oder einen NULL-Zeiger (Rückgabewert Zeiger) zurück. Der genaue Fehlergrund ist dann ergänzend in der globalen Variablen errno gespeichert. Mittels der Standard-C Funktion strerror() kann die Zahl in einen für den Anwender/Programmierer lesbaren String umgewandelt werden (die GNU-C Library Funktion printf() bietet ergänzend den Conversion Spezifier %m als alternative Ausgabemöglichkeit an):

```
#include <errno.h>
int var;
if(-1==scanf("%d",&var) )  printf("Es wurde keine Zahl erkannt\n");
int ret=open("/etc/ptmp",O_WRONLY|O_CREAT|O_EXCL,
                         S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
if(-1==ret)
  printf("open failed() with %s\n",strerror(errno));
if(-1==ret)
  printf("open failed() with %m (nur glibc)\n");
```

Für eine sauber Programmierung sollte diese Methodik generell angewendet werden. Jede Funktion, welche Fehlschlagen kann, teilt dies per Rückgabewert dem Aufrufer mit. Der Rückgabewert einer jeden Funktion wird vom Aufrufer kontrolliert.

4.14.2 Main-Funktion

Mit Start der Anwendung übergibt das Betriebssystem die Kontrolle an die Anwendung und führt dort die Funktion main aus. Als Übergabe- und Rückgabeparameter sind folgende Varianten erlaubt:

```
int main(void) {  }
int main(int argc, char *argv[]) {  }
```

Mit dem Start der Anwendung kann, wie bei jedem Funktionsaufruf, Parameter an den startenden Prozess übergeben werden. Als Parameter können Strings übergeben werden, welche in Form eines Arrays von Strings übergeben werden. Im Parameter argc sind die Anzahl der Strings enthalten. Im Parameter argv[x] sind die einzelnen Strings enthalten. Sofern vorhanden ist im ersten String der Programmname/Aufrufname der Anwendung enthalten:

```
int main(int argc, char *argv[]) {
  //Variante 1: Darstellung des Inhaltes des Arrays von Strings
  for(int lauf=0;lauf < argc; lauf++)
    printf("%d.Parameter: '%s'\n",lauf,argv[lauf]);
  //Variante 2: Darstellung des Inhaltes des Arrays von Strings
  (void)argv;  //Wird hier nicht benötigt
  for(char **str=argv;*str!=NULL;str++)
    printf("Parameter: '%s'\n",*str);
  return 0;
}
```

Hinweis:

- Im Compiler-Explorer können die Parameter im Executor unter 'Execution Arguments', getrennt mit Leerzeichen eingegeben werden.

Wird die Funktion main beendet, entweder über die return Anweisung oder den Aufruf der exit() Funktion, wird die Kontrolle zurück an den startenden Prozess (zumeist Shell) gegeben. Über den Rückgabewert wird im Allgemeinen dem Aufrufer der Fehlerstatus mitgeteilt (0 bedeutet fehlerfreie , ungleich 0 fehlerhafte Ausführung). Der Start der Anwendung kann als Funktionsaufruf angesehen werden, welchen einen Wert an den Aufrufer zurückliefert.

Hinweise:

- Der Return-Wert der main-Funktion kann mit »echo $? in der Bash abgefragt werden
- Mittels `exit(status)` kann das Programm an jeder Stelle innerhalb des Programms beendet werden. Der Übergabewert von Exit() entspricht dann der expression der return Anweisung von main

4.14.3 Inline Funktion

Wie der Name inline andeutet, ist die Intention von Inline, dass die aufgerufen Funktion Bestandteil der aufrufenden Funktion ist. Die aufgerufene Funktion soll somit im Kontext des Aufrufers ausgeführt werden, so dass kein eigener Stack bereitgestellt werden muss. Der Start der Funktion kann somit schneller erfolgen:

```
inline int foo(void) { }
```

Das Inline erfolgt durch den Compiler (und nicht durch den Linker). Damit das inline funktioniert, muss die zu inlinende Funktion vor dem Aufruf definiert sein, d.h. sie muss in derselben C-Datei enthalten sein [C11 6.7.4].

Hinweis:

- Das 'inlinen' von Funktionen über die eigene Objektdatei hinweg erfolgt in C/C++ über Link Time Optimization (LTO). Hierzu fügt der Compiler den Zwischencode mit in die Objektdatei, so dass der Linker das 'inlinen' vollziehen kann (siehe Interprocedural Optimization[38])

4.14.4 Variadic Function / Ellipsis Punctuator

Mit dem Ellipsis Puncturator '...' kann in Funktionsdefinitionen angegebenen werden, dass anstatt der ... Anweisung, beliebig viele Parameter/Argumente übergeben werden können. Dies wird unter anderen in der printf() Anweisung genutzt:

```
int printf(const char *fmt, ... );
```

Die wie folgt aufgerufen werden kann:

```
printf("Hello World");   //es folgt kein Parameter
printf("%d",100);        //es folgt ein Parameter vom Datentyp int
printf("%f",1.1);        //es folgt ein Parameter vom Datentyp double
printf(conststr,1,2,3,"xyz")  //Ohne Worte
```

Da der Compiler für die optionalen Parameter nicht den Zieldatentyp kennt, erlaubt die C-Spezifikation als Übergabedatentypen hier nur die Grunddatentypen `signed int`, `unsigned int`, `long`, `long long`, `double` und `Zeiger`. Entspricht der übergebene Parameter nicht einen dieser Datentypen, so wird er mittels implizit Cast entsprechend nachfolgender Tabelle gewandelt [C11 6.5.2.2 Default Argument Promotions]:

Ausgangstyp	**Umwandlungstyp**
char, signed char, short	int
unsigned char, unsigned short	unsigned int
float	double
Alle anderen Datentypen	bleiben erhalten

Bspw.:

```
signed char varc=-1;
float  varf     =0.1;
printf("%d %f",varc, varf);  //varc wird beim Aufruf nach int konvertiert!
                             //varf wird beim Aufruf nach double konvertiert!
```

Zum Umgang mit dem variadischen Parameter stellt C die Funktionen `va_start()`, `va_arg()`, `va_copy()` und `va_end()` bereit. Auf Grundlage des dazugehörigen Datentyps `va_list` können variadische Parameter auch an Unterfunktionen weitergereicht werden (bspw vprintf()). Dieses Thema bedingt jedoch genaue Kenntnisse vom Umgang mit dem Stack und ist nicht Bestandteil dieser Vorlesung.

38 https://de.wikipedia.org/wiki/lang%3Den

4.15 Variable Length Array (VLA)

Im Normalfall ergibt sich die Anzahl der Elemente eines Arrays aus einer Konstanten, so dass der Compiler zur Compilezeit passend Speicher reserviert (siehe Kap. Array[39]):

```
    int arr[4];    //Compiler reserviert für Variable arr
                   //4*sizeof(int) Speicherplatz
```

Bei Variable Lenght Array wird die Arraydimension nicht über eine Konstante, sondern über eine Variable oder einen Ausdruck angegeben. D.h. die tatsächliche Dimension des Arrays ergibt sich erst zum Ausführungszeitpunkt der Definition:

```
void foo(size_t n) {
  int arr[n*2+1];   //Arraydimension ergibt sich zum Ausführungszeitpunkt
  n++;              //Nachträgliches Ändern von n ändert nicht
                    //die Größe des Arrays!
}
```

Ein Variable Length Array kann nur als lokale Variable oder als Übergabeparameter definiert werden (wobei im letzteren Fall kein Speicher hiermit reserviert wird). Eine Initialisierung eines VLA's mit der Definition ist nicht möglich, d.h. nach der Definition beinhaltet das Array 'zufällige' Werte. Es empfiehlt sich daher, diese bspw. mit memset händisch zu initialisieren (siehe auch Kap Array:Initialisierung[40]):

```
void foo(size_t n) {
  char str[n];
  memset(str,0,n*sizeof(char));  //Array mit 0 füllen
```

Mittels sizeof kann weiterhin der Speicherbedarf ermittelt werden, wobei sizeof mit einem VLA als Argument kein Konstantenausdruck ist. Stattdessen wird die Größe zur Laufzeit ermittelt:

```
void foo(size_t n) {
  int arr[n*2+1];    //VLA
  size_t size;
  size=sizeof(arr);  //sizeof(arr) entspricht einem Funktionsaufruf
                     //welcher die Speichergröße des VLA's ermittelt
```

VLA wurden erstmals in C mit C99 eingeführt. In C11 wurden VLA's als optional eingestuft und mit C23 zurück auf notwendig gesetzt. In C++ sind VLA's nicht definiert, werden jedoch vom GCC Compiler unterstützt. Als alternativen Datentyp wird hier std::array empfohlen.

Hinweis:

- VLA sind nicht direkt in einem Switch-Case Block definierbar. Wird hier ein VLA benötigt, so muss hierzu ein Block erzeugt werden:

```
 switch(value) {
   case 1:
     char arr1[value];  //Compilerfehler, da bei value!=1 kein Speicherplatz
                        //für das VLA reserviert wird
     break;
   case 2: {
       char arr2[value];   //OK
```

39 Kapitel 8 auf Seite 179
40 Kapitel 8.8 auf Seite 192

```
    }
    break;
}
```

4.16 Operatoren

In der Programmiersprache C sind folgende Punctuators [C11 6.4.6] definiert:

```
   [ ]  ( )  { }  .  ->
   ++   --    &   *  +   -   ~   !
   /    %    <<  >>  <   >  <=   >=  ==  !=  ^  |  &&  ||
   ?    :    ;  ...
   =          *=       /=       %=   +=     -=     <<=     >>=    &=    ^=
|=     ,     #   ##
   <:   :>   <%   %>   %:   %:%:
```

Die Funktionalitäten sind identisch zu Java. Aufgrund des Datentyps Zeiger gibt es in C ergänzend den Punctuator `->` (siehe Kap Datentypen:Struktur/Verbundtyp[41]). Der Präprozessor bedingt die beiden Punctuatoren `#` und `##` (siehe Kap Präprozessor[42]). Die letzten 6 Punctuators `<: :> <% %> %: %:%:` werden zu `[ ] { } # ##` ersetzt. Sie sind für Computersysteme gedacht, welche keine ALTGR-Taste besitzen.

Java kennt ergänzend den `>>>` Operatoren, welcher in C über den normalen Schiebebefehl und den zugrundeliegenden Datentyp gesteuert wird.

Wie in Java sind einige Punctuators vom Kontext abhängig und haben damit unterschiedliche Funktionalitäten:

```
   ++   → (Postfix) ++       vs.    ++(Präfix)
   --   → (Postfix) --       vs.    --(Präfix)
   +/-  → Vorzeichen         vs. Addition/Subtraktion
   ()   → Funktionsaufruf()  vs. Befehlsbestandteil if()  vs. Priorisierung
```

Für den Umgang mit dem Datentyp Zeiger besitzen einige Punctuatoren ergänzende Funktionalitäten (siehe Kap Zeiger[43]):

```
& →  Bitweises UND:
     Anwendung: Ganzahlausdruck & Ganzahlausdruck
& →  Adressoperator:
     Anwendung: &Variable
* →  Multiplikation
     Anwendung: Ganzahl-/Gleitkommaausdruck * Ganzahl-/Gleitkommaausdruck
* →  Datentyp Zeiger
     Anwendung: Datentyp * Variablenname;
* →  Dereferenzierung:
     Anwendung: * Zeiger
```

41 Kapitel 5.5 auf Seite 99
42 Kapitel 9 auf Seite 203
43 Kapitel 7 auf Seite 139

Punctuatoren, welche von Java abweichende / ergänzende Funktionalitäten haben, werden nachfolgend gesondert beschrieben.

Sind in einer Anweisung mehrere Operatoren enthalten, so hängt die Abarbeitungsreihenfolge von der Rangfolge/Priorität der Operatoren ab (siehe C-Programmierung:Liste der Operatoren nach Priorität[44]).

Die Rangfolge ist identisch zu Java, wobei C mehr Operatoren kennt. Generell empfiehlt sich, lieber eine Klammer zu viel als zu wenig zusetzen. Klammern haben die höchste Priorität.

4.16.1 Zuweisungs-Operator

Die Zuweisung (=-Operator, assignment expression) weist dem L-Value den R-Value zu. Ergänzend dient der L-Value als Rückgabewert des Operators, so dass das Ergebnis der Zuweisung weiter genutzt werden kann. Die Abarbeitungsreihenfolge ist von rechts nach links (R-L Assoziativität). Wird der 'letzte' Rückgabewert nicht verwendet, so wird dieser verworfen (siehe Kap. Discarded Value[45]).

Beispiel:

```
a=7;        //Zuweisung
a=b=7;      //Erste Zuweisung (b=7) hat einen Rückgabewert, welcher im
            //Anschluss a zugewiesen wird
if(a=7)     //Variablen a wird der Wert 7 zugewiesen. Der Rückgabewert
            //der Zuweisung wird im Anschluss auf True oder False geprüft
if(7=a)     //Besser, da aufgrund der fehlerhafter Zuweisung ein
            //Compilerfehler geworfen wird
x*=y=z      //entspricht  x=x*(y=z)
a=b=d+7;    //entspricht  a=(b=(d+7))
a=b==c;     //a bekommt das boolesche Ergebnis der Vergleichsoperation
            //zugewiesen
```

Hinweise:

- Auf beiden Seiten der Zuweisung muss der identische Datentyp stehen. Wenn andernfalls eine implizite Typkonvertierung nicht möglich ist, gibt der Compiler einen Fehler aus:

```
int a = 7.3;    //implizite Typkonvertierung
double b=a+3;   //Implizite Typkonvertierung der Intergeraddition nach double
float c=(float)a+(float)3;  //Explizite Typkonvertierung
char *d=a;      //Compilerfehler, da Integerwert nicht implizit in einen
                //Zeiger konvertiert werden kann
```

- Bei der Parameterübergabe beim Funktionsaufruf wird ebenfalls indirekt der Zuweisungsoperator angewendet. Die Parameter des Aufrufers werden per Zuweisung den lokalen Variablen der aufgerufenen Funktion zugewiesen

44 https://de.wikibooks.org/wiki/C-Programmierung%3A%20_Liste_der_Operatoren_nach_Priorit%C3%A4t

45 Kapitel 4.13.2 auf Seite 62

```
void func(int a, float b) {
  printf("a=%d b=%f",a,b);
}
int   x=4;
float y=5.0;
func(x,y);   //der Inhalt der Variablen x wird der lokalen Variable a
             //(a=x) und der Inhalt der Variablen y der lokalen Variablen b
             //(b=y) zugewiesen.
func(y,x);   //Dito, mit der Ergänzung, dass eine implizite Typkonvertierung
             //stattfindet (a=(int)y) und (b=(float)x).
```

4.16.2 Komma-Operator

Der Komma-Operator erlaubt es, zwei Ausdrücke auszuführen, wo nur einer erlaubt wäre. Die Ergebnisse aller durch diesen Operator verknüpften Ausdrücke, außer dem letzten werden verworfen. Der letzte Ausdruck dient als Rückgabewert.

Folgendes gilt es bei der Nutzung des Komma-Operators zu beachten:

- Der Komma-Operator wird von links nach rechts ausgewertet
- Das Komma selbst stellt eine Sequence Point dar (siehe Kap Sequence Point[46])
- An Stellen, wo das Komma zum Syntax gehört (z.B. Trennung der Übergabeparameter bei der Funktionsdefinition und beim Funktionsaufruf, Trennung von Variablen bei der Variablendefinition/-deklaration, Trennung der der enum Konstanten) muss der Ausdruck ergänzend geklammert werden, wenn der Komma-Operator gefordert ist
- Der Komma-Operator hat von allen Operatoren die niedrigste Priorität/Rangfolge, so dass dieser Operator als letztes 'ausgewertet' wird

Beispiel:

```
int a=(5,3);       //5 wird verworfen, 3 dient als Rückgabewert, so dass
                   //a den Wert 3 erhält.
int b=5,3;         //KO, da Komma Bestandteil der Definition ist
int c=1,d=2,e=3;
int f=(c+=2,a+a);  //c um zwei erhöht, a+a in f
int g= c+=2,a+a;   //KO, da Komma Bestandteil der Defintion ist
void func(int value){}
func(1);           //Funktion wird mit dem Wert 1 aufgerufen
func(1,2);         //KO, da Komma Bestandteil des Funktionsaufrufes ist
func((1,2));       //Funktion wird mit dem Wert 2 aufgerufen
int arr[12];
arr[3,4]=1;        //3 wird verworfen, 4 dient als Index für das Array
```

'Sinnvolle' Anwendungsbeispiele:

- Bei For-Schleifen, so dass mehrere Variablen initialisiert und mehrere Variablen mit jedem Schleifendurchlauf geändert werden können:

```
int sum,lauf;
int arr[]={5,4,3,2,1};
for(sum=100,lauf=5 ; lauf ; lauf--,sum+=arr[lauf]);
```

46 Kapitel 4.13.3 auf Seite 63

- Bei der Return-Anweisung, bei welcher zur Darstellung der Zusammengehörigkeit mehrere Aktionen ausgeführt werden sollen:

```
return errno=EINVAL,-1;
return printf("Fehler"),-1;
```

- An Stellen, wo nur eine Anweisung erlaubt ist (und man zu faul ist, einen Block zu öffnen)

```
if(…) y=2,z=3;
```

Hinweis:

- In vielen Ländern wird das Komma zur Darstellung von Gleitkommazahlen genutzt. Wird versehentlicher Weise dies in C/C++ übernommen, so führt dies zu einem unerwarteten Verhalten:

```
double var;
var=5.3;      //Variable wird mit 5.3 initialisiert
var=5,3;      //KO, da Komma Bestandteil der Defintion ist
var=(5,3);    //Variable wird mit 3 initialisiert
```

4.16.3 ?-Operator / Conditional-Operator

Operator mit 3 Operanden, welcher in Abhängigkeit des ersten Operanden den zweiten oder dritten Operanden zurückgibt.

`Syntax: Logical-OR-expression ? expression : condition-expression`

Entspricht der if()-Anweisung, wobei der ? Operator als solches einen Wert zurückgibt! Expression wird nur ausgewertet, wenn der erste Operand true ist. Andernfalls wird nur condition-expression ausgeführt. Der zurückgegebene Datentyp ergibt sich aus der Expression resp. Condition-expression. Beide sollten somit vom identischen Datentyp sein.

Folgendes gilt es bei der Nutzung des ?-Operators zu beachten

- Der Datentyp Expression und condition-Expression muss identisch sein

```
int a=5;
a=a > 10 ? "String" : 4;   //KO, "String" ist vom Datentyp (char*) und
                           //       4     ist vom Datentyp (int).
```

- Nach der GNU-C Reference kann der mittlere Operand fehlen, in diesem Fall wird stattdessen das Ergebnis des linken Operanden dort eingesetzt (siehe `https://gcc.gnu.org/onlinedocs/gcc/Conditionals.html#Conditionals`)

```
int a=13;
int b=a>10 ? : 0;    //wenn a > 10 wird true, andernfalls false zurückgegeben
```

- Sollen innerhalb von condition-expression mehrere (durch Komma getrennte) Anweisungen enthalten sein, so sind diese zu Klammern! Andernfalls bedingt die niedrige Priorität des Komma-Operators ein anders Verhalten, als erwartet.

```
int a=7,max=5;
max=a>max ? a : max;         //Alles bestens

int var0,var1,var2;
var0=a>5 ? a++,5:6,a=1;      //das zweite Komma wird als separate Anweisung
                             //ausgeführt und kommt somit immer zur
                             //Ausführung
var1=(a>5 ? a++,5:6),a=1;    //durch Klammerung klare Abgrenzung
var2= a>5 ? a++,5 : (6,a=1); //durch Klammerung klare Abgrenzung
```

Die Nutzung des ?-Operators empfiehlt sich an diversen Stellen:

- Bei Makros

```
#define MAX(a,b) (a>b?a:b)
```

- Beim Fehlerhandling

```
char *str=strstr("hallo123","lox");
//Ausgabe von str nur, wenn Suche erfolgreich (str!=NULL) war
printf("%s",str!=NULL?str : "(not found)"  );
```

- Prüfen, ob eine Funktion erfolgreich ausgeführt wurde, bevor eine weitere Funktion ausgeführt wird

```
int ret;
ret=init1();       //im Fehlerfreien gibt init1() 0 zurück
ret=ret?:init2();  //Aufruf von init2() nur wenn ret==0 ist
                   //andernfalls wird ret zurückgegeben
```

4.16.4 Logische Verknüpfungen

Alle logischen Operationen nutzen als Operanden Ganzzahlen ,Gleitkommazahlen oder Zeiger. Der Operand wird als false angesehen, wenn der Operand 0, 0.0 oder NULL ist. Andernfalls als true. Das Ergebnis ist immer vom Datentyp int und hat den Wert von 0 oder 1.

```
Syntax: Expression && Expression
Syntax: Expression || Expression
Syntax: ! Expression
```

Bei der logischen UND und ODER Verknüpfung wird zuerst der linke Ausdruck ausgewertet (Operator ist ein Sequence Point). Der rechte Ausdruck wird nur ausgeführt, wenn:

- bei der logischen UND Verknüpfung der linke Ausdruck TRUE ergab

- bei der logische OR Verknüpfung der linke Ausdruck FALSE ergab

Andernfalls steht das Ergebnis schon mit der Auswertung des linken Operanden fest.

In der theoretischen Informatik wird dies 'Non-strict-evaluation' genannt, bei welcher der Rückgabewert feststeht, bevor alle Ausdrücke bearbeitet worden sind.

Beispiel für das Verhalten des Ausbleibens der Auswertung des zweiten Operators:

```
int a=0,b=0;
//a=-1; //Als alternativer Startwert
if(++a || ++a)
  b=1;
```

Diese Funktionalität ist insbesondere beim Umgang mit Fehlern vorteilhaft:

- Prüfen, ob eine Variable einen gültigen Wert hat, bevor sie weiterverwendet wird

```
char *str = strstr("hallo123","la");
if(str!=NULL  && strlen(str)  )
  printf("Len = %ld",strlen(str));
```

- Prüfen, ob eine Funktion erfolgreich ausgeführt wurde, bevor eine weitere Funktion aufgerufen wird:

```
int ret;
ret=init1();  //Bei erfolgreicher Initialisierung wird 1 zurückgegeben
ret=ret && init2();
```

4.16.5 Bitweise Verknüpfungen

Alle bitweisen Operationen nutzen als Operanden Ganzzahlen. Bei vorzeichenbehafteten Zahlen ist das Ergebnis nach [C11 6.5] Implementierungsabhängig, wobei das Vorzeichenbit zumeist als eigenständiges Bit angesehen wird und bei UND/ODER/EXOR als normales BIT betrachtet wird.

```
Syntax: Ganzzahl & Ganzzahl → Bitweise UND Operation
Syntax: Ganzzahl | Ganzzahl → Bitweise ODER Operation
Syntax: Ganzzahl ^ Ganzzahl → Bitweise EXOR Operation
Syntax: ~ Ganzzahl → Bitweise Negierung
Syntax: Ganzzahl >> additive-expression
Syntax: Ganzzahl << additive-expression
```

Mittels des Schiebeoperators können Ganzzahlen um additive-expression bitweise nach Links << oder Rechts >> verschoben werden. Der resultierende Datentyp ergibt sich aus dem linken Operanden. Der rechte Operand muss positiv und kleiner gleich der Bitbreite des linken Operanden sein. Andernfalls ist das Ergebnis implementierungsspezifisch.

Nach [C11 6.5.7] ist das Ergebnis der Schiebeoperation implementierungsspezifisch, wenn der linke Operand eine negative vorzeichenbehaftete Zahl ist. In der Regel wendet der Compiler jedoch folgende Schiebeoperation an:

- Logisches Schieben (zu füllende Bit-Stellen werden mit 0 aufgefüllt), wenn der linke Operand eine vorzeichenlose Zahl ist
- Arithmetisches Schieben (zu füllende Bit-Stelle beim Rechtsschieben wird mit dem Vorzeichenbit aufgefüllt), wenn der linke Operand eine vorzeichenbehaftete Zahl ist. Hiermit wird das Vorzeichen beibehalten, so dass eine negative Zahl nach dem Schieben weiterhin negativ ist (In Java erfolgt dies durch den »> - Operator)

Hinweis:

- Ein Schieben um eine Stelle nach rechts entspricht einer Division durch 2
- Ein Schieben um eine Stelle nach links entspricht einer Multiplikation mit 2

4.16.6 sizeof-Operator

Der sizeof-Operator dient zur Ermittlung des Speicherbedarfs (in Bytes) einer Variablen/eines Datentyps.

```
Syntax: sizeof (type-name)
Syntax: sizeof unary-expression
```

Als Argument für den Sizeof-Operator kann wahlweise ein Datentyp, eine Variable oder ein Ausdruck genutzt werden. Sizeof gilt als Konstantenausdruck, d.h. seine Berechnung erfolgt durch den Compiler, sodass:

- sizeof zur Initialisierung von globalen Variablen genutzt werden

```
struct xyz{int x,y,z;};
size_t size=sizeof(struct xyz);
```

- bei einem Ausdruck dieser nicht ausgeführt wird, sondern nur der resultierende Datentyp bestimmt wird und dessen Größe zurückgegeben

```
short var=4711;
size_t size=sizeof var++;   //var=4711, size=2
```

Der resultierende Datentyp des Rückgabewertes ist size_t.

Es empfiehlt sich, den sizeof-Operator mit Klammern zu nutzen, andernfalls wird im Falle eines Ausdruckes der Speicherbedarf des linken Operanden angegeben:

```
size_t size;
short var;
size=sizeof var+0;   //size=2, da sizeof auf var angewendet wird
size=sizeof(var+0);  //size=4, da sizoeof auf den resultierenden datentyp
                     //angewendet wird
```

Hinweise:

- sizeof angewendet auf char gibt immer den Wert 1 zurück.

- sizeof ist nicht auf Operanden vom Typ Funktion, void, Felder ohne Größenangabe und Bitfelder anwendbar
- wird sizeof auf ein VLA angewendet, so entspricht sizeof nicht mehr einen Konstantenausdruck, sondern einem Funktionsaufruf

```
void foo(int n) {
  int arr[n];
  size_t size=sizeof(arr);  //Speicherbedarf ist kein Konstantenausdruck
                            //sondern wird zur Laufzeit berechnet
```

- Sizeof entspricht nicht strlen(), da sizeof den für die Variable reservierten Speicherbereich zurückgibt und nicht wie strlen() den Inhalt des Speicher hinsichtlich des Stringendezeichens auswertet.

```
char string[100]="Test";
printf("%zd\n",sizeof(string));  //100
printf("%zd\n",strlen(string));  //4
```

4.17 Anweisungen

4.17.1 If-Anweisung

`Syntax: if (expression) statement`
`Syntax: if (expression) statement else statement`

Das Ergebnis des Ausdruckes (Ganzzahl, Gleitkommazahl oder Zeiger) wird auf ungleich 0, 0.0 oder NULL getestet. Trifft dies zu, so wird statement ausgeführt. Andernfalls wird das else statement ausgeführt, sofern vorhanden. Sollen mehrere Anweisungen ausgeführt werden, so sind diese mittels Block zusammenzufassen (diesen dann bitte nicht mit ; abschließen).

Beispiel:

```
if(a==1)
if(a=1)    //Erst Zuweisung, dann prüfen der Rückgabe der Zuweisung
if(a)      //entspricht a!=0
if(printf("hallo")<0)  //prüfen, ob printf() fehlschlägt
if( ({int dummy=a;a=b;b=dummy;})==6)
if(a=7)    //Zuweisung!
if(7=a)    //KO, Eine Konstante kann kein L-Value sein
if(7==a)
  {printf("a ist 7");};       //Semikolon hinter Blocksope ist eine zweite
else                          //Anweisung, so dass die ELSE Anweisung nicht
  {printf("Compilerfehler");};//zur IF-Anwesigung zugeordnet werden kann.
```

Hinweis:

- Bei der Anweisung `if(a==7)` finden aus syntaktischer Sicht zwei Aktionen statt. Zunächst wird geprüft, ob a gleich 7 ist. Dieser Vergleich liefert true(=1) oder false(=0) zurück. Die If-Anweisung prüft im Anschluss, ob das Argument true oder false ist.

4.17.2 For-Anweisung

`Syntax: for(Initialisierungsteil`$_{\text{OPT}}$; Test$_{\text{OPT}}$; Fortsetzung$_{\text{OPT}}$) Statement

Die For-Anweisung wird vom Compiler wie folgt umgesetzt:

```
    Initialisierungsteil_OPT
  Goto Next
Goon:
   //Continue-Anweisung entspricht goto Continue
   //Break-Anweisung entspricht goto Break;
   Statement
Continue:
   Fortsetzung_OPT
Next:
   if(Test_OPT) goto Goon
Break:
```

Die einzelne Elemente haben folgende Bedeutung/Besonderheiten:

Initialisierungsteil

Zum Initialisieren von Schleifenvariablen, wird einmalig mit Beginn der FOR-Anweisung ausgeführt.

- kann Leer sein, dann erfolgt keine Initialisierung
- Ab C99: Hier kann eine lokale Variable definiert werden, welche nur innerhalb dieser For-Anweisung gültig ist

Testteil

Hierin wird vor jedem Schleifendurchlauf geprüft, ob die Schleifenbedingung erfüllt ist. Wenn Bedingung nicht erfüllt ist (Ganzzahl, Gleitkommazahl oder Zeiger gleich 0,0.0 oder NULL), wird die For-Schleife beendet.

- Kann leer sein, dann wird dieser Teil als true angesehen

Fortsetzungsteil

Wird nach jeden Schleifendurchlauf ausgeführt. In der Regel sollte hier die Schleifenvariable aktualisiert werden

- Kann leer sein

Statement

In der Schleife auszuführender Ausdruck

- Sollen mehrere Ausdrücke ausgeführt werden, so sind diese in einem Block zusammen zu fassen.

Ergänzende Anweisungen innerhalb des Statements/Block

- 'continue' zum Fortsetzen des Schleifendurchlaufes, d.h. Sprung zum Fortsetzungs-Teil
- 'break', zum Beenden des Schleifendurchlaufes

Beispiele:

- For-Schleife mit lokaler Schleifenvariable

```
char string[]="hallo";
for(size_t lauf=0; lauf < strlen(string); lauf++)
//Schleife ist Suboptimal, da in jedem Durchlauf die Funktion
//strlen() aufgerufen wird!
```

- Schleife mit Break-Anweisung

```
char dest[5];
char src[10]="HalloWelt";
for(size_t lauf=0; src[lauf]!=0; lauf++) {
  if(lauf >= sizeof(dest))
    break;
  dest[lauf]=src[lauf];
}
```

- Schleife, bei welcher die gesamte Aktion innerhalb der For-Anweisung ausgeführt wird

```
char dest[5];
char src[10]="HalloWelt";
for(size_t l=0;
    src[l]!=0 && l <sizeof(dest);
    printf("l=%zd\n",l),dest[l]=src[l],l++);
```

- Schleifenvariable muss keine Zählvariable sein

```
int  zahl=0b100010;
int bits=0;
for(int mask=0b10000000;  mask!=0; mask>>=1 )
  bits+= zahl & mask ? 1 : 0;
```

- Zeiger als Schleifenvariable

```
char src[]="hallo";
char dst[strlen(src)+1];
char *s,*d;
for(s=src,d=dst; *s; )
  *d++=*s++;
*d=0;
```

Hinweis:

- Zur Geschwindigkeitsoptimierung sollten im Test- und Fortsetzungsteil keine Funktionsaufrufe mit Werten enthalten sein, die sich innerhalb der Schleife nicht ändern:

```
for(int lauf=0;lauf<strlen("hallo");lauf++)
```

- Zur Geschwindigkeitssteigerung sollte im Testteil auf 0 abgefragt werden, d.h. die Schleifenvariable dekrementiert werden

```
for(int lauf=700000;lauf;lauf--)
```

- Alternativ zu einer Laufvariablen empfiehlt sich die Nutzung von Pointer als Schleifenvariable;

```
for(char *ptr=str; *ptr;  ptr++);
```

4.17.3 While-Anweisung

`Syntax: while(expression) Statement`
`Syntax: do Statement while(expression);`

Die While-Anweisungen werden vom Compiler wie folgt umgesetzt:

while()-Anweisung

```
continue:
  if(!expression)
      goto break
  //Break-Anweisung entspricht
  //  goto break;
  //Continue-Anweisung entspr.
  //   goto continue
  statement
  goto continue
break:
```

do while()-Anweisung

```
continue:
  //Break-Anweisung entspr.
  //  goto break
  //Continue-Anweisung ent.
  //  goto continue
  statement:
  if(expression)
     goto continue
break:
```

Folgendes gilt es bei der Nutzung der While-Anweisung zu beachten:

- Soll mehr als ein Statement innerhalb der While-Schleife ausgeführt werden, so sind diese in einem Block zu schreiben
- Statement kann eine Leeranweisung (gekennzeichnet durch ein Semikolon) sein

```
while(1);
do ; while(1);
```

- Bei der do-While-Schleife ist das abschließende Semikolon Bestandteil des Befehls

- expression muss eine Ganzzahl, Gleitkommazahl oder Zeiger zurückgeben, der bei False (0, 0.0 oder NULL) die Schleife beendet.

Beispiele:

- Endlosschleife

```
while(1) {}
```

- Schleife zum Durchlaufen eines String

```
char str[]="string";
char *ptr=str;
while(*ptr++);
size_t len=ptr-str-1;
```

4.17.4 Switch-Anweisung

```
Syntax: Switch(expression) statement
Syntax: { case constant-expression: statementopt default: statementopt}
```

Die Switch-Anweisung entspricht einer verschachtelten If-Anweisung, mit der Besonderheit, dass der auszuführende Code nicht im If-Zweig stattfindet, sondern dort ein goto enthalten ist:

Switch-Anweisung

```
switch(a) {
   case 1:
     int a=7;
      //Code
    break;
   case 2:
     printf("%d",a);

   case 3:  {
     int a;
     //Code
      }
    case  4  ...  8:
    default:
      //Code
}
```

Compilerumsetzung

```
if(a==1)
  goto label1;
else if(a==2)
  goto label2;
else
  goto label_default;
//--------------------
label1:
   //Code
   goto label_break;
label2:
  //Code
label_default:
   goto label_break;

label_break:
```

Folgendes gilt es bei der Nutzung der Switch-Anweisung zu beachten:

- Expression muss zu einer Ganzzahl ausgewertet werden (int, short, char, long, long long)
- Constant-expression muss eine Integer-Konstante. Sie darf nicht doppelt im Block vorkommen. Der genaue Datentyp ergibt sich aus dem Datentyp der Expression

- Ohne den folgenden Block kann nur ein case oder default Fall enthalten sein

```
int var=1;
switch(var) case 1: printf("hallo");
```

- Die geschweifte Klammer der Switch-Anweisung erzeugt einen Block, so dass hierin definierte Variablen innerhalb des gesamten Block zur Verfügung stehen (sind also nicht CASE abhängig). Die CASE Anweisungen sind nichts anderes als Sprungmarken, so dass der Initialisierungsteil der lokalen Variablen übersprungen / nicht ausgeführt werden kann:

```
switch(a) {
    int a=3;     //Variable in gesamten Block gültig
  case 1:
    int b=3;  //Variabe in gesamten Block gültig
    printf("%d %d",a,b);  //a undefiniert, b=3
    break;
  case 2:
    printf("%d %d",a,b);  //a und b undefiniert
    break;
  case 3: {        //Block, so dass Variable
    int a=4711;    //nur in diesem CASE gültig ist
    break;
    }
}
```

- GNU-C (siehe Switch-Statement[47]) erlaubt ergänzend zur Angabe einer Konstante die Angabe eines Wertebereiches.

```
switch(a) {
  case 1 ... 8:   //Leerzeichen zwischen den Konstanten zwingend notwendig
                  //andernfalls wird der Punkt zur Konstante zugehörig
                  //interpretiert (erzeugt dann eine Gleitpunktkonstante)
```

- Break ist optional. Wenn break weggelassen wird, geben Compiler ggf. eine Warning aus. Mittels der GCC-Compiler-Anweisung __attribute__((fallthrough)); kann diese Warning unterdrückt werden.

```
switch(a) {
  case 0:  //ggf. Compiler-Warning
  __attribute__((fallthrough)); case 1:
  case 2:
    break;
}
```

- Case - Anweisungen sind nichts anderes als Sprungmarken (Labels), so dass CASE-Anweisungen überall stehen können:

47 https://www.gnu.org/software/c-intro-and-ref/manual/html_node/switch-Statement.html

```
int prime(int value) {return value+1;}
void hallo(int var)  {
  switch(var) {
    default:
      if(prime(var))
        case 2: case 4:
        printf("%d",var);
      else {
        case 3: case 6:
        printf("%x",var);
      }
    case 7:
      printf("xyz");
  }
}
```

- Variable Length Arrays sind als Variablen im Block der Switch Anweisung nicht erlaubt. Ist dies notwendig, so sind diese in einen inneren Block anzulegen

```
switch(var) {
  case 1:
    int arr[var];  //KO
    break;
  case 2: {
    int arr[var];  //OK
   }
```

Hinweis:

- Eine Switch-Case Anweisung über Strings sind in C/C++ nicht möglich:

```
    switch(str) {
      case "hallo":
        return 1;
      case "du da":
        return 2;
      case "xyz":
        return 3;
      default:
       rReturn -1;
    }
```

Ein direkte Lösungsansatz ist die Nutzung von strcmp() zum Vergleichen von String. Da strcmp zeichenweise den String vergleicht, ist dieser Ansatz langsam. Alternativ bietet sich an, von allen String einen Hash-Wert zu berechnen und zunächst diesen Hash-Wert zu kontrollieren. Da der Hash-Wert nicht eindeutig ist, muss ergänzend ein strcmp() erfolgen:

```
char *str="hallo";
switch(HASH_LAUFZEIT(str)) {
  case HASH_COMPILEZEIT('d','u'):
    if(!strcmp(str,"du"))
       printf("Case \"du\":\n");
    break;
  case HASH_COMPILEZEIT('h','a','l','l','o'):
```

```
    if(!strcmp(str,"hallo"))
      printf("Case \"hallo\"\n");
    else if(!strcmp(str,"xyz"))
      printf("String='xyz'\n");
    break;
default:
  return -1;
}
```

Siehe:Hash Table in C[48]

4.18 Label + Goto

`Syntax: name:`

Labels dienen zur Kennzeichnung einer Anweisung, so dass die Programmausführung an dieser Stelle fortgesetzt werden kann, resp. mit einer GOTO-Anweisung angesprungen werden kann. Ein Label ist durch den folgenden Doppelpunkt gekennzeichnet. Dem Label muss eine Anweisung folgen, welches im Falle einer folgenden Variablendefinition durch eine Leeranweisung (Semikolon) erzeugt werden kann.

Folgendes gilt es bei der Nutzung von Labels zu beachten:

- Ein Label entspricht der Speicheradresse des der Label-Anweisung folgenden Anweisung
- Bei nicht genutzten Labels (mit goto nicht angesprungene Labels) gibt der Compiler eine Warning aus. Diese kann beim GCC-Compiler mit __attribute__((unused)) unterdrückt werden
- die Case- und die Default-Anweisung aus der Switch-Anweisung entspricht einem Label

`Syntax: goto label;`

Mit der Goto-Anweisung wird die Programmausführung an der mit `label` gekennzeichneten Stelle fortgesetzt.

Folgendes gilt es bei der Nutzung von goto zu beachten:

- Mit einer Goto Anweisung kann an eine beliebige Steller innerhalb der **identischen** Funktion gesprungen werden. Es kann nicht zu einem Label einer anderen Funktion gesprungen werden (dies erfolgt über `setjmp()` und `longjmp()`)
- Mit der Goto Anweisung kann gleichermaßen in einen Block hinein, als auch herausgesprungen werden:

```
goto first_time;
for(;;) {
  if(a>10)
    goto last_time:
```

48 https://benhoyt.com/writings/hash-table-in-c/

```
    first_time:
    a++;
  }
  last_time: ;
```

- Wird mit der Goto Anweisung der Initialisierungsteil einer Variablen übersprungen, so ist zwar die Variable existent, beinhaltet aber nicht ihren Initialisierungswert

Das Label hat eine funktionsweite Sichtbarkeit (siehe Kap Gültigkeit/Sichtbarkeit von Variablen[49]). Dies bedeutet, dass das:

- Labels nur innerhalb einer Funktion gültig ist
- Labels einen von Variablen/Funktionen/Datentypen unabhängigen Namensraum haben
- Kein Prototyp für ein Label notwendig ist, wenn es bei Nutzung über goto noch nicht gesetzt worden ist, also nach vorne gesprungen wird

Beispiel:

```
goto label1;
if(a==3) {
  label2:
    printf("True\n");
}
else
  label1:
{
  printf("False\n");
}
if(a==3) goto label2;
```

Hinweis:

- In Java gibt es das Schlüsselwort goto, welches aber ohne Funktion ist. Es gibt jedoch Labels, die mit ”break label” angesprungen werden können.

Prinzipiell sind Goto-Anweisungen zu vermeiden. In einigen Anwendungsfällen erweist sich goto als nützlich:

- Konstruktor mit mehreren Initialisierungsteilen

```
int main(int argc,char *argv[]) {
  enum {INIT_2,INIT_1,INIT_START} fortschritt=INIT_START;
  int *ptr1;
  int *ptr2;
  if(!(ptr1=malloc(10)))
    goto end;
  fortschritt=INIT_1;
  if(!(ptr2=malloc(20)))
    goto end;
  fortschritt=INIT_2;
  //    ...
  end:
  switch(fortschritt) {
    case INIT_2: free(ptr2); __attribute__((fallthrough));
    case INIT_1: free(ptr1); __attribute__((fallthrough));
```

49 Kapitel 4.5 auf Seite 37

```
    case INIT_START:
  }
  return (int)fortschritt;
}
```

- Abbruch einer verschachtelten Schleife

```
int arr[10][10];
size_t zeile;
size_t spalte;
for(zeile=0;zeile<10;zeile++)
  for(spalte=0;spalte<0;spalte++)
    if(arr[zeile][spalte]==0)
      goto label_end;
label_end:
if((zeile==0) && (spalte==0))
  printf("Nichts gefunden");
else
  printf("Gefunden an %zd %zd",zeile,spalte);
```

4.19 Sonstiges

4.19.1 ??-Trigraphs

Zeichenfolge bestehend aus 3 Zeichen beginnend mit 2 Fragezeichen. Die beiden Fragezeichen entsprechen einem ESCAPE-Operator (siehe Kap Char-Konstanten[50]), welcher das nachfolgende Zeichen durch ein anderes ASCII Zeichen ersetzt. Die Ersetzung findet auch innerhalb von Strings statt.

Trigraphs wurden früher genutzt, als die ALTGR Taste noch nicht üblich war.

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

Bei neuer C Varianten muss zur Nutzung von Trigraphs der Compiler mittels -trigraphs aktiviert werden.

50 Kapitel 4.10.2 auf Seite 54

5 Datentypen

Die Programmiersprache C kennt nachfolgende grundlegende Datentypen und Zeiger auf diese Datentypen (siehe auch Grundlagen:Grundlegende Datentypen und Typsicherheit[1])

Datentyp	Bsp. für Datentyp	Bsp. für Zeiger auf Datentyp

Ganzzahl[2]	`char var;`
	`int var;`
	`short int var;`
	`long int var;`
	`long long int var;`
		`char *ptr;`
		`int *ptr;`
		`short int *ptr;`
		`long int *ptr;`
		`long long int *ptr;`

Gleitkommazahl[3]	`double var;`
	`float var;`
	`long double var;`
		`double *ptr;`
		`float *ptr;`
		`long double *ptr;`

Funktionen	`void func(void) {}`	`void (*pfunc)(void);`

Arrays[4]	`int arr[10];`
	`float arr[3][3];`
		`int (*parr);`
		`float (*parr)[3];`
		`void (*pfunc[5])(void);`

Strukturen[5] /Unions[6]	`struct xyz {int x,y,z};` `union abc {int a,b,c};`	`struct xyz *ptr;` `union abc *ptr;`
Aufzählungstyp[7]	`enum abc {A,B,C};`	`enum abc *ptr;`
Boolescher Datentyp[8]	`_Bool var;`	`_Bool *ptr;`

1 Kapitel 4.16.6 auf Seite 79
2 Kapitel 5.1 auf Seite 92
3 Kapitel 5.2 auf Seite 95
4 Kapitel 8 auf Seite 179
5 Kapitel 5.7 auf Seite 105
6 Kapitel 5.8 auf Seite 115
7 Kapitel 5.9 auf Seite 118
8 https://de.wikibooks.org/wiki/Programmieren%20in%20C%2FC%2B%2B%3A%20Grundlagen%20%23%20Boolescher-Datentype%2FOperatoren

Datentyp	Bsp. für Datentyp	Bsp. für Zeiger auf Datentyp
Komplexe Zahlen[9]	`_Complex var;`	`_Complex *ptr;`

Bis auf die Datentypen Funktionen, Array und die Zeiger sollen die wesentlichen Aspekte dieser Datentypen im nachfolgenden beschrieben werden.

Der grundlegende Datentyp ist 'int'!

- Es wird mit nichts Kleinerem gerechnet als mit dem Datentyp 'int' (siehe Implizite Typumwandlung[10]) :

```
char var1=10,var2=20,var3;
var3=var1+var2; //Addition findet auf Basis des Datentyps int statt
```

- Es gibt zwar den Booleschen Datentyp, dieser entspricht jedoch einem Ganzzahldatentyp und `true` und `false` entsprechen den Integerwerten 0 und 1
- Der Aufzählungsdatentyp und seine Aufzählungselemente sind vom Datentyp 'int'
- Bis einschl. C90 wurde bei der Definition von Variablen ohne Angabe des Datentyps, bei Definition einer Funktion ohne Angabe des Datentyps der Übergabeparameter, des Rückgabedatentyps der Datentyp implizit auf int gesetzt (implizit int)

```
static var2=2;
foo(par1,par2){
  auto var1;
  var1=par1+par2;
  return var1;
}
```

5.1 Ganzzahl Datentypen

Grundlegend kennt die Programmiersprache C nur zwei Ganzzahldatentypen:

- Datentyp: `int`

  Vorzeichenbehafteter (signed) Datentyp, dessen Bitbreite von der Rechnerarchitektur und des Compilers abhängig ist

- Datentyp `char`

  Datentyp mit einer Mindestbreite von 8-Bit u.A. zur Speicherung von Zeichen typischerweise im ASCII Format. Die C-Spezifikation [C11 6.2.5 Types] lässt es dem Compiler frei, ob char als vorzeichenbehafteter oder vorzeichenloser Datentyp implementiert wird.

9 Kapitel 5.3 auf Seite 97
10 Kapitel 5.6.1 auf Seite 101

Ergänzend kann der Datentyp char auch zur Speicherung von ganzen Zahlen genutzt werden. In diesem Fall empfiehlt sich, diese explizit als signed und unsiged zu definieren. Wenn nur ASCII Zeichen gespeichert werden, so ist diese Angabe nicht nötig!

Optional können Qualifier dem Datentyp vorangestellt werden:

- `unsigned` / `signed` zur Angabe, ob der Inhalt als vorzeichenlose Zahl oder als vorzeichenbehaftete Zahl zu interpretieren ist. Ohne Angabe sind Variablen vom Datentyp int vorzeichenbehaftet
- `short` / `long` / `long long` (letzteres seit C99) als vorangestellten Qualifier zu `int` zur Vergrößerung/Verkleinerung der Bitbreite. Bei Angabe des Qualifiers ist das Schlüsselwort `int` optional. Die Beschreibung `long long int` und `long long` sind gleichbedeutend!

Als Bitbreite für die Datentypen ist im C-Standard nur ein Mindestbreite definiert. Die genaue Bitbreite hängt von der Rechenbreite des Systems und vom Compiler ab (siehe [C11 6.2.5 Types]):

Datentype	**Mindest breite**	**Typisch bei 32-Bit Architektur**	**Typisch bei 64-Bit Architektur**	**printf Format Anweisung**
`char`	8	8	8	`%c`
`signed char` `unsigned char`	8	8	8	`%hhd` `%hhu`
`short` `short int`	16	16	16	`%hd` / `%hu`
`int`	16	16/32	32	`%d` / `%u`
`long` `long int`	32	32	32/64	`%ld` / `%lu`
`long long` `long long int`	64	64	64	`%lld` / `%llu`

In der Header-Datei `limits.h` sind die tatsächlichen Grenzwerte der Datentypen gespeichert:

Konstante	**Beschreibung**	**Typ. bei 64-Bit Architektur**
`CHAR_BIT`	Anzahl der Bits in einem Char	8
`SCHAR_MIN`	min. Wert, den der Typ bei signed char aufnehmen kann	-128
`SCHAR_MAX`	max. Wert, den der Typ bei signed char aufnehmen kann	127
`UCHAR_MAX`	max. Wert, den der Typ bei unsigned char aufnehmen kann	255
`CHAR_MIN`	min. Wert, den der Typ char aufnehmen kann	0 oder SCHAR_MIN
`CHAR_MAX`	max. Wert, den der Typ char aufnehmen kann	SCHAR_MAX oder UCHAR_MAX
`SHRT_MIN`	min. Wert, den der Typ short int annehmen kann	-32.768

Konstante	Beschreibung	Typ. bei 64-Bit Architektur
SHRT_MAX	max. Wert, den der Typ short int annehmen kann	32.767
USHRT_MAX	max. Wert, den der Typ unsigned short int annehmen kann	65.535
INT_MIN	min. Wert, den der Typ int annehmen kann	-2.147.483.648
INT_MAX	max. Wert, den der Typ int annehmen kann	2.147.483.647
UINT_MAX	max. Wert, den der Typ unsigned int aufnehmen kann	4.294.967.296
LONG_MIN	min. Wert, den der Typ long int annehmen kann	-2.147.483.648 oder -9.223.372.036.854.775.808
LONG_MAX	max. Wert, den der Typ long int annehmen kann	2.147.483.647 oder 9.223.372.036.854.775.807
ULONG_MAX	max. Wert, den der Typ unsigned long int annehmen kann	4.294.967.296 oder 18.446.744.073.709.551.616
LLONG_MIN	min. Wert, den der Typ long long int annehmen kann	-9.223.372.036.854.775.808
LLONG_MAX	max. Wert, den der Typ long long int annehmen kann	9.223.372.036.854.775.807
ULLONG_MAX	max. Wert, den der Typ unsigned long long int annehmen kann	18.446.744.073.709.551.615

Aufgrund dessen, dass die Breite der Datentypen von der Rechnerarchitektur und vom Compiler abhängig ist, sind in der Header-Datei `stdint.h` Aliase für Datentypen enthalten, welche in ihrem Alias die tatsächliche Breite und das Vorzeichens beinhalten:

```
uint8_t    -->  unsigned  8-Bit
 int8_t    -->    signed  8-Bit
uint16_t   -->  unsigned 16-Bit
 int16_t   -->    signed 16-Bit
    ...
```

Für C++ sehen diese Datentypen wie folgt aus:

```
std::intptr_t
std::int8_t    std::uint8_t
std::int16_t   std::uint16_t
std::int32_t   std::uint32_t
Std::int64_t   std::uint64_t
```

Zur Erzeugung von portablen/rechnerunabhängigen Programmen empfiehlt sich, diese Datentypen zu nutzen. Die Rechenbreite ist dann unabhängig vom Compiler und Betriebssystem.

5.2 Gleitkomma Datentypen

Gleitkommazahlen werden im Computer auf Basis folgender Schreibweise dargestellt (siehe auch: w:Gleitkommazahl[11]):

```
              Dezimalsystem                 Dualsystem
12,3456710 = 1234567 * 10^-5       0,110 = 1,1001100110011 * 2^-4
             -------      --                ---------------     --
             Mantisse  Exponent             Mantisse         Exponent
```

Mantisse

Vorzeichenlose ganze Zahl (ggf. mit einer festen Position des Dezimalpunktes=Festkommazahl)

Exponent

Vorzeichenbehaftete ganze Zahl, welche um einen Bias verschoben ist

Vorzeichen

Vorzeichen der (vorzeichenlosen) Mantisse

Zur Speicherung im Computer werden das Vorzeichen, die Mantisse und der Exponent getrennt im Binärformat gespeichert, für den Programmierer aber als eine (zusammenhängende) Zahl dargestellt:

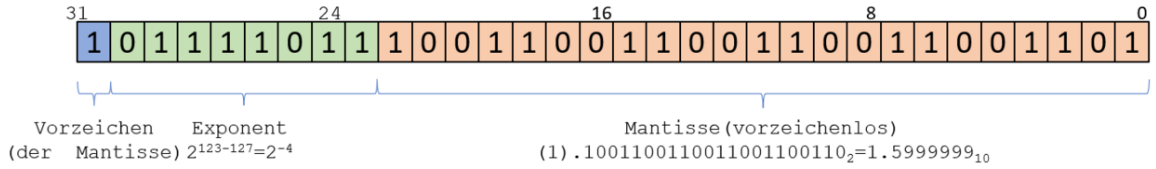

Abb. 6 Bitweise Darstellung einer Gleitkommazahl, aufgeteilt in Vorzeichen, Exponent und Mantisse

Gleitkommazahlen haben genauso wie ganze Zahlen einen beschränkten Wertebereich:

- Die Anzahl der Bits der Mantisse bestimmen die Anzahl der Nachkommastellen:

```
Bei 23-Bit Mantisse hat das niederwertigste Bit die Wertigkeit:
2^-22=1/2^22=1/(2^10*2^10*2^2)=1/(1024*1024*4)≅1/(4.000.000)=0,000.000.25
→ Es könnten also nur 7..8 dezimale Nachkommastellen gespeichert werden
```

- Mit dem Exponent kann 'quasi' der kleinste und der größte Wertebereich angegeben werden:

```
Bei 8-Bit Exponent mit einem Bias von 127 liegt der Wertebereich des
Exponenten im Bereich von -126 … +127 (Exponent -127 und +127 wird zur
Darstellung weitere Zahlen benötigt).
Die kleinste darstellbare Zahl (unter Vernachlässigung der Nachkommastellen der
```

11 https://de.wikipedia.org/wiki/Gleitkommazahl

```
 Mantisse ist:
2^-126=1/2^126≅1,175*10^-38
Die größte darstellbare Zahl ist:
2^127       ≅1,7*10^38
```

Die genaue Darstellung (Anzahl der Bits für Mantisse und Exponent, Darstellung von NAN/INF, ...) ist in C nicht spezifiziert und somit compiler- und rechnerabhängig. Zumeist wird mittlerweile die w:IEEE 754[12] Standardisierung genutzt (Norm wurde von Intel mit der Entwicklung der 8087 FPU entworfen).

Grunddatentypen und der Wertebereich (nach w:IEEE 754[13])

Datentype	Speicherplatz	Exponent	Mantisse	Größte Zahl	Kleinste Zahl	Genauigkeit
`float`	4 Byte	8 bit	23 bit	$+$-3,4*10^{38}	1,2*10^{-38}	6 Stellen
`double`	8 Byte	11 bit	52 bit	$+$-1,7*10^{308}	2,3*10^{-308}	12 Stellen
`long double`	10 Byte	≥8 bit	≥63 bit	$+$-1,1*10^{4932}	3,4*10^{-4932}	18 Stellen

Aufgrund des nicht standardisierten Formates sollte ein Gleitkommadatentyp nicht im Binärformat für den Datenaustausch (mit Netzwerk, über Dateien, ..) genutzt werden. Andere Rechnersysteme/Programmiersprachen würde aufgrund der anderen Interpretation andere Zahlenwerte aus den übertragenen/gespeicherten Binärdaten auslesen!

```
float var1;
write(file_hdl,&var1,sizeof var1);   //Schreiben des Binärwertes
                                     //in eine Datei
double var2;
send(socket_hdl,&var2,sizeof var2,0);//Senden des Binärwertes
                                     //über ein Netzwerk
```

Die Standard-C-Library bietet einige mathematische Funktionen wie `sin() / cos() / tan() / atan2() / sqrt() / pow() / exp()/…` an. Die Prototypen dieser Funktion sind in der Header-Datei `math.h` beschrieben, so dass diese bei Nutzung dieser Funktion zu inkludieren ist. Ergänzend ist die Shared Library libm.so über den Compilerschalter '-lm' einzubinden:

```
#include <math.h>
//Math-Library libm.so mittels Linker-Command '-lm' einbinden
int main(int argc,char *argv[]) {
  double var1=47.11;
  double var2=sqrt(var1);
```

Die Funktionen basieren auf den Datentyp `double`, d.h. sowohl der Übergabewert als auch der Rückgabewert ist vom Datentyp `double`. Die Prototypen sehen wie folgt aus:

```
double sin(double);
double cos(double);
double tan(double);
double asin(double);
double acos(double);
double atan(double);          //Wertebereich der Rückgabe von -PI/2..+PI/2
double atan2(double,double);  //Wertebereich der Rückgabe von -Pi .. +PI
double sqrt(double);
double log(double);
```

12 https://de.wikipedia.org/wiki/IEEE%20754
13 https://de.wikipedia.org/wiki/IEEE%20754

Über den Suffix f oder l im Funktionsnamen kann der zugrundeliegende Datentyp auf float (Suffix f) oder long double (Suffix l) geändert werden:

```
float       sinf(float);
long double sinl(long double);
```

Ergänzend zu den Prototypen sind die gebräuchlichen Naturkonstanten in `math.h` definiert (siehe auch: math.h[14]):

```
M_E         Value of e
M_LOG2E     Value of log_2 e
M_LOG10E    Value of log_10 e
M_LN2       Value of log_e 2
M_LN10      Value of log_e 10
M_PI        Value of π
M_PI_2      Value of π/2
M_PI_4      Value of π/4
M_1_PI      Value of 1/π
M_2_PI      Value of 2/π
M_2_SQRTPI  Value of 2/√π
M_SQRT2     Value of √2
M_SQRT1_2   Value of 1/√2
```

Gleitkommazahlen können nicht nur Zahlenwerte, sondern auch Sonderwerte annehmen. Diese Sonderwerte sind ebenfalls in `math.h` beschrieben (siehe auch: math.h[15]):

```
INFINITY  A constant expression of type float representing
          positive or unsigned infinity, if available; else a
          positive constant of type float that overflows at
          translation time.
NAN       A constant expression of type float representing a
          quiet NaN. This macro is only defined if the
          implementation supports quiet NaNs for the float type.
```

Mit `fesetround()/fegetround()` kann gesetzt/gelesen werden, wie mit Ergebnissen umzugehen sind, die nicht exakt darstellbar sind. Mögliche Werte sind 'round to nearest' (default), 'round up', 'round down' und 'round toward zero'.

5.3 Komplexe Zahlen

Nach dem Wikipedia Artikel w:komplexe Zahlen[16] *stellen komplexe Zahlen eine Erweiterung der reellen Zahlen dar. Ziel der Erweiterung ist es, algebraische Gleichungen wie* $x^2+1=0$ bzw. $x^2=-1$ lösbar zu machen. ... Da die Quadrate aller reellen Zahlen größer oder gleich 0 sind, kann die Lösung der Gleichung $x^2=-1$ keine reelle Zahl sein. Man braucht eine ganz neue Zahl, die man üblicherweise i nennt, mit der Eigenschaft $i^2=-1$. Diese Zahl i wird als imaginäre Einheit bezeichnet.Komplexe Zahlen werden nun als Summe a+b*i definiert, wobei a und b reelle Zahlen sind und i die oben definierte imaginäre Einheit ist.
Der Datentyp für komplexe Zahlen (erst ab C99 enthalten) beinhaltet folglich zwei Gleitkommazahlen, eine für den reelen Teil und eine für den imaginären Teil. Entsprechend den Gleitkommazahlen steht der komplexe Datentyp ebenfalls in drei unterschiedlichen Genauigkeiten zur Verfügung:

14 https://www.man7.org/linux/man-pages/man0/math.h.0p.html
15 https://www.man7.org/linux/man-pages/man0/math.h.0p.html
16 https://de.wikipedia.org/wiki/komplexe%20Zahlen

Datentype	Speicherplatz
`float _Complex`	2x4Byte
`double _Complex` `_Complex`	2x8Byte
`long double _Complex`	2x10Byte

Wie auch beim Booleschen Datentyp ist in der Header-Datei `complex.h` für den einfacheren Umgang mit diesem Datentyp das Makro `complex` als Textersetzung für `_Complex` gesetzt.

Zur Darstellung von komplexen Konstanten wurde die Gleitkommakonstante um den Suffix i ergänzt. Durch Anhängen von i wird aus dem ansonsten rellen Teil der imaginäre Teil und damit aus der Gleitkommakonstante eine komplexe Gleitkommakonstante:

```
_Complex varc1=2i;
_Complex varc2=1+2i;
printf("%f %f\n",creal(varc1),cimag(varc1));
printf("%f %f\n",creal(varc2),cimag(varc2));
```

Normale Gleitkommavariablen werden als reellen Teil einer komplexen Zahl angesehen. Bei Operationen von Gleitkommazahlen und komplexen Zahlen wird der imaginäre Teil der Gleitkommazahl auf 0 gesetzt. Beim Zuweisen einer komplexen Zahl an eine Gleitkommazahl wird nur der reele Teil 'gelesen'.

Soll eine Gleitkommavariablen zu einem imaginären Teil gewandelt werden, so muss diese bspw mit der Konstanten 1.0i multipliziert werden:

```
double vard1=2i;       //Vorsicht, es wird nur der reele Teil
printf("%f\n",vard1);  //der komplexen Zahl in vard1 gespeichert!
_Complex varc1=1+2i;
         vard1=1;
varc1=varc1+vard1;
printf("%f %f\n",creal(varc1),cimag(varc1));
varc1=varc1+vard1*1.0i;
printf("%f %f\n",creal(varc1),cimag(varc1));
```

Die Standard-C-Library bietet diverse mathematische Funktionen wie csin(), ccos(), csqrt(), cpow() und ergänzend Umrechnungsfunktionen von Gleitkommazahlen in komplexe Zahlen (und umgedreht) wie creal(),cimag() ,cabs() für den Umgang mit komplexen Zahlen an:

```
double _Complex csin(double _Complex);
double _Complex csqrt(double _Complex);
double          creal(double _Complex);
double          cimag(double _Complex);
```

Zur Nutzung dieser Funktionen muss die Header-Datei `complex.h` inkludiert und ergänzend die Shared Library libm.so über den Compilerschalter '-lm' eingebunden werden.

```
#include <complex.h>
//Math-Library mittels Linker-Command '-lm' einbinden
int main(int argc, char *argv[]) {
  double _Complex xyz=6+2i;  //Reeler-Teil=6  Imaginärer Teil=2
  printf("Reeller Teil: %f Imaginärer Teil:%f",creal(xyz),cimag(xyz));
```

Die Funktionen beruhen auf den Datentyp `double _Complex`, d.h. sowohl der Übergabewert als auch der Rückgabewert ist `double _Complex` resp. `double` (bei den Umrechnungsfunktionen). Über den Suffix f und l im Funktionsnamen kann der zugrundeliegende Datentyp geändert werden:

```
float       crealf(float _Complex z);
long double creall(long double _Complex z);
```

Hinweis:

- Der Datentyp `_Complex` und die dazugehörigen mathematischen Funktionen sind in der C-Spezifikation als optional gekennzeichnet.

5.4 Boolschescher Datentyp

Siehe Grundlagen: Boolescher-Datentype/Operatoren[17]

5.5 void / unvollständiger Datentype

Der Datentyp `void` dient vorrangig dazu, nicht vorhandene Über- und Rückgabewerte von Funktionen anzuzeigen. Eine Variable vom Datentyp `void` kann nicht angelegt werden. Ein Zeiger vom Datentyp `void` ist möglich, kann aber nicht dereferenziert werden (siehe Kap. Zeiger:Void-Zeiger[18])

```
void func(void); //Void zur Darstellung der nicht vorhandenen Parameter und
                 //des nicht vorhandenen Rückgabewertes
void var;        //Eine Variable vom Datentyp void kann nicht angelegt werden
void *ptr;       //Void-Zeiger
```

Hinweise:

- `sizeof(void)` ist nach der C-Spezifikation nicht erlaubt. Dennoch geben viele Compiler hier den Wert 1 zurück!
- Mit einem expliziten Cast auf den Datentyp void wird dem Compiler mitgeteilt, dass diese Variable in Gebrauch ist, der Wert in dieser Operation aber nicht genutzt wird. Hierüber kann die Compilerwarning 'unused Variable' unterbunden werden:

```
void func(int par1, int par2) {
  int var1=12;
  int var2=13;
  (void)par2;     //Angabe, dass diese Variable genutzt wird.
  (void)var2;     //Angabe, dass diese Variable genutzt wird.
  if(var1==par1) //var1 und par1 werden verwendet
```

17 Kapitel 4.16.6 auf Seite 79

18 https://de.wikibooks.org/wiki/Programmieren%20in%20C%2FC%2B%2B%3A%20Zeiger%20%23Void-Zeiger

5.6 Datentypkonvertierung

Die Abarbeitungsreihenfolge der Operatoren wird durch die Prioritätenliste/Rangfolge (siehe C-Programmierung:Liste der Operatoren nach Priorität[19]) vorgegeben. Operatoren mit höherer Priorität werden vor Operatoren mit niedriger Priorität ausgeführt:

```
//Der Ausdruck
c = sizeof(x) +  ++a / 3;
//wird aufgrund der Prioritäten wie folgt ausgewertet:
c= (sizeof(x)) + ( (++a) / 3);
```

Bei identischer Priorität ergibt sich die Abarbeitungsreihenfolge aus der Assoziativität (L-R oder R-L)

```
a=33 / 5 / 2;
//Wird aufgrund der Assoziativität wie folgt ausgewertet.
//a= (33 / 5) / 2;
//und damit zu 3 und nicht zu 16 (bei 33 / (5/2)) ausgewertet.
a = b = c = d*2;  //→ a=(b=(c=(d*2)));
a = b = 1+c = d;  //→ a=(b=((1+c)=d)); //Compilerfehler, da
                                       //1+c kein lvalue ist
```

Für das Rechnen/Vergleichen müssen beide Operatoren vom identischen Datentyp sein! Sind diese nicht identisch, so müssen die Datentypen 'angeglichen' werden. Dies kann einerseits ‚automatisch' mittels 'implizierter' Typumwandlung oder ‚manuell' mittels ‚expliziter' Typumwandlung erfolgen.

Beim Zuweisungsoperator (inkl. Parameterzuweisung bei Funktionsaufrufen und Funktionsrückgabewerte) gilt dies ebenso, nur dass hier der Quelldatentyp an den Zieldatentyp angepasst werden muss. Die impliziten Regeln finden hier keine Anwendung.

Hinweis:

- Empfehlenswert ist, die Datentypen der Variablen so zu wählen, dass der Compiler keine implizite Typumwandlung tätigt. Kann dies nicht vermieden werden, so sollte die explizite Typumwandlung genutzt werden (um sich einen möglichen Datenverlust bewusst zu machen).
- Der Datentyp selbst sollte den möglichen Wertebereich der Variablen entsprechen und nicht unnötig groß gewählt werden.

```
Datentyp zur Speicherung der Stundenzeit:
    Wertebereich: 0 ... 23  -> unsigned char
Datentyp zur Speicherung einer Jahreszahl:
    Wertebereich: 2000 v.C. ... 4000 n.C -> signed short
Datentyp zur Speicherung einer Temperatur:
    Wertebreich: -273,0°C ... 2000,0°C  -> float
```

19 https://de.wikibooks.org/wiki/C-Programmierung%3A%20_Liste_der_Operatoren_nach_Priorit%C3%A4t

5.6.1 Implizite Typumwandlung

Diese Regeln wurden so aufgestellt, dass dabei stets ein Datentyp in einen anderen Datentyp mit höherem Rang umgewandelt wird (Rangordnung: long double, double, float, long long, long, int) [C11 6.3.1.8] .

Nach [Harbison: S. 199]			
Regel/ Priorität	**If either operand has Type**	**And the other operand has Type**	**Converts both to**
1	long double	any real type	long double
2	double	any real type	double
3	float	any real type	float
4	any unsigned type	any unsigned type	The unsigned typewith the greater rank
5	any signed type	any signed type	The signed typewith the greater rank
6	any unsigned type	a signed type of greater rank that can represent all vaues of the unsigned type	The signed type
7	any unsigned type	a signed type of greater rank that cannot represent all values ot the unsigned type	The unsigned version the the signed type
8	any other type	any other type	No conversion

Ergänzend gilt das Regelwerk zu Integer Promotion [C11 6.3.1.1], welche Datentypen kleiner als signed int zu int und kleiner als unsigned int und unsigned int konvertiert (Compiler kann hiervon abweichen, wenn sichergestellt ist, dass kein Datenverlust eintritt).

Regel 6 und 7 sind der nicht klar definierten Bitbreite der ganzzahligen Datentypen geschuldet und versuchen, Konvertierungsverluste zu vermeiden. Sie lesen sich zunächst kryptisch, lassen sich aber einfach am folgenden Beispiel erklären:

```
Operand 1: unsigned int (hier 32-Bit)
Operand 2: long         (hier 32-Bit  / 64-Bit)
```

- Im Fall, dass der Datentyp `long` 64-Bit breit ist, kann dieser problemlos die vorzeichenlose 32-Bit Zahl ohne Konvertierungsverluste repräsentieren, so dass der Zieltype `signed long` ist (Regel 6)
- Im Fall, dass der Datentyp `long` 32-Bit breit ist, kann dieser mit seinen Wertebereich von -2.147.483.648 bis +2.147.483.647 nicht den vorzeichenlosen Zahlenbereich von 0...4.294.967.295 darstellen. In diesem Fall hat der unsigned Datentyp einen höheren Rang, so dass als Zieltyp `unsigend long` gewählt wird(Regel 7)

Regel 7 bedeutet die Gefahr eines Konvertierungsverlustes, dessen man sich bewusst sein sollte! Diese tritt insb. dann in Kraft, wenn ein Operand vom Typ 'unsigned long long' ist (64-Bit).

Für die Konvertierung von Pointer, Arrays, Strukturen, Unions zu anderen Datentypen, als sich selbst gilt Regel 8. In diesem Fall gibt der Compiler zumeist einen Error, in wenigen Ausnahmefällen eine Warning aus:

```
char *string;             //Pointer
int   arr[3];             //Array
struct {int x,y;} var1;   //Strukturen
union  {int x,y;} var2;   //Union
var1 = var2;    //Error Incompatible Types
string=arr;     //Error Incompatible Types
arr=var1;       //Error Assignment to expression with array type
                //(=incompatible Types)
string=var2;    //Error Incompatbiles Types
```

Beispiele von impliziten Typumwandlungen:

```
char  varc=100;
short vars=100;
int   vari=100;
vars=varc + vars;
//Wird aufgrund der impliziten Regel wie folgt umgesetzt
vars=(short)((int)varc+(int)vars);

vari=5.0*(int)sin(vars);
//Wird aufgrund der impliziten Regel wie folgt umgesetzt
vari=(int)(5.0*(double)(int)sin((double)vars));
//Hinweis: da sin() nur Werte im Bereich -1…0…+1 zurückgibt kommen
//als Werte für a hier nur 5, 0 und -5 in Frage!
```

Hinweis:

- Im CompilerExplorer können sie sich die impliziten Typumwandlungen anzeigen lassen. Dazu gehen sie bitte wie folgt vor:
  - Im Source-Fenster mit '+Add new' ein Compiler Fenster öffnen
  - Im Compiler-Fenster mit '+Add new' ein 'GCC Tree/RTL' Fenster öffnen
  - Im GCC Tree/RTL-Fenster unter 'Select a pass...' 'original tree' auswählen

Um implizite Typumwandlung besser erkennen zu können, wird oftmals bei Variablennamen die **'Ungarische Notation'**angewendet (siehe w:Ungarische Notation[20]). Aufgrund der besonderen Namensgebung kann der Programmierer ohne großen Aufwand frühzeitig mögliche Typkonflikte erkennen. Der Variablenname setzt sich wie folgt zusammen:

```
{Präfix}{Datentyp}{Bezeichner}
```

Präfix: p->Pointer h->Handle i->index c->count f->flag rg->Array

Datentyp: ch->Character st->string w->word b->byte...

Bezeichner: Zum Binden der Variable an eine konkrete Aufgabe (keine Unterstriche).

Bezeichner ist optional, wenn aus Präfix und Datentyp die 'Aufgabe' der Variable direkt sichtbar ist

Beispiele:

20 https://de.wikipedia.org/wiki/Ungarische%20Notation

```
char  rgchtemp[10];   //Array (Range) vom Datentyp character
int   ich;            //Index zum Adressieren eines Arrays vom Datentyp
                      //character
```

5.6.2 Explizite Typumwandlung

Programmierer erzwingt durch explizite Typumwandlung (auch CASTen genannt) eine Umwandlung eines Datentyps in einen anderen. Dazu wird der Zieldatentyp in runden Klammern vor Quelldatentype geschrieben:

```
short a=4;
double b=(double)(a+1);   //Das Ergebnis von (a+1) wird nach double gecastet
                          //(Addition erfolgt auf Basis des Datentyps.
                          //integer)
double c=(double)a+1;     //a wird nach double gecastet, so dass nachfolgende
                          //Addition auf Basis von double basiert.
```

In der Rangfolge der Operatoren steht der Cast-Operator unterhalb von bspw. Funktionsaufrufen, Arrayzugriffen aber auch der Dereferenzierung:

Priorität	**Symbol**	**Assoziativität**	**Bedeutung**
15	...	...	...
14	++/– (Präfix) +/- (Vorzeichen) !/~ & * (TYP) ...	R-L	Präfix-Inkrement/Dekrement Vorzeichen Logisches/Bitweises Nicht Adresse Zeigerdereferenzierung **Typumwandlung (Explizites Cast)** ...
13	• / %	L-R	Multiplikation/Division/Modulo
12	...	...	...

Im Zweifel gilt auch hier, den zu wandelnden Typ ergänzend zu klammern.

Hinweis:

- Bedenke, dass die explizite Typumwandlung so aufgestellt sein sollte, dass der Zieldatentyp dem notwendigen Datentyp entspricht. Andernfalls wendet der Compiler ergänzend eine implizierte Typumwandlung an:

```
int    a;
double b=4.7;
short  c1= (long)(  (float)a+b );
//b ist vom Typ double, so dass a nach dem Cast auf float
//   implizit vom Compiler auf double gecastet wird
//c ist vom Typ short, so dass das Ergebnis der Addtion nach dem expliziten
//Cast auf long auf short gecastet wird.
//Nach Anwendung der impliziten Cast sieht der Ausdruck wie folgt aus:
short c2=(short)(long)( (double)(float)a+4.7);
```

Mittels expliziter Typumwandlung können Ganzzahl nach Gleitkommazahlen (und andersherum) und Zeiger in einen anderen Zeiger und auf andere Datentypen gewandelt werden. Eine Konvertierung von Arrays, Strukturen, Unions zu anderen Datentypen als sich selbst ist weiterhin nicht möglich:

```
char *string;                //Pointer
int arr[3];                  //Array
struct stru {int x,y;} var1; //Strukturen
union  unio {int x,y;} var2; //Union

var1 = (struct stru) var2;   //Error Conversion to non-scalar type
string=(char *)arr;          //OK
arr=(int *)var1;             //Error Assignment to expression with array type
string=(char *)var2;         //Error cannot convert to pointer type
```

Bei den möglichen Konvertierungen sollte Folgendes berücksichtigt werden:

- **Vorzeichenlose Ganzzahl → Vorzeichenlose Ganzzahl**: Wenn der Zieldatentyp größer ist, wird die Zahl durch führende '0' erweitert. Wenn der Zieldatentyp kleiner ist, werde die zu vielen Bitstellen abgeschnitten/verworfen (ggf. Datenverlust).
- **Vorzeichenbehaftete Ganzzahl → Vorzeichenbehaftete Ganzzahl**: Wenn der Zieldatentyp größer ist, wird vorzeichenrichtig erweitertet (d.h. das Auffüllen erfolgt auf Basis des Vorzeichens). Wenn der Zieldatentyp kleiner ist, werden auch hier die zu vielen Bits abgeschnitten/verworfen (Vorsicht: Negative Zahlen können dabei in positive Zahlen gewandelt werden)
- **Gleitkommazahl → Gleitkommazahl**: Hier wird vereinfacht ausgedrückt sowohl die Mantisse als auch der Exponent einzeln kopiert. Wenn die Zielmantisse kleiner ist, gehen 'Nachkommastellen' verloren. Wenn der Quellexponent einen größeren Wert beinhaltet, als der Zielexponent 'aufnehmen' kann, so wird die Zahl auf Unendlich gesetzt. Umgedreht, wenn der Quellexponent kleiner ist, als der Zielexponent 'aufnehmen' kann, so wird die Zahl auf 0 gesetzt
- **Ganzzahl → Gleitkommazahl**: Hier wird, vereinfacht ausgedrückt, die Ganzzahl in die Mantisse kopiert. Um Datenverluste zu vermeiden, sollte die Bitbreite der Zielmantisse größer gleich der Bitbreite der Ganzzahl sein
- **Gleitkommazahl → Ganzzahl**: Im Wesentlichen wird hier die Mantisse übernommen, so dass hier die Bitbreite des Zieldatentyps mindestens der Bitbreite der Mantisse sein sollte
- **Zeiger ↔Ganzzahl**: Die Konvertierung ist möglich. Da bei 64-Bit Systemen der Zeiger 64-Bit und Integer 32-Bit breit ist, meckert ggf. der Compiler. Für solche Fälle existieren die Datentypen 'intptr_t' und 'uintptr_t', über welche sichergestellt ist, dass ein Zeiger in einen Ganzzahldatentyp gespeichert werden kann!
- **Zeiger ↔Gleitkommazahl**: Diese Konvertierung ist nicht möglich!
- **Struktur/Union ↔Ganzzahl/Gleitkommazahl**: Diese Konvertierung ist nicht möglich. Es können nur einzelnen Strukturelemente konvertiert werden, sofern diese vom Typ Ganzzahl/Gleitkommazahl sind
- **Array ↔Ganzzahl/Gleitkommazahl**: Der Arrayname entspricht einen Zeiger, so dass hier ein Zeiger in eine Ganzzahl oder umgedreht konvertiert wird (siehe Zeiger ↔ Ganzzahl)

In C++ gibt es weitere CAST-Operatoren, auf welche hier derzeit noch nicht weiter eingegangen wird!

- Static_cast (Entspricht dem Expliziten Cast)
- Const_cast
- Dynamic_cast
- Reinterpret_cast

5.7 Struktur/Verbundtyp

Nach dem Wikipedia Artikel w:Verbund (Datentyp)[21] ist *ein Verbund (englisch object composition) ist ein Datentyp, der aus einem oder mehreren Datentypen zusammengesetzt wurde. Die Komponenten können wiederum Verbünde enthalten, wodurch auch komplexe Datenstrukturen definiert werden können.*
Für jedes Strukturelement wird Speicher reserviert. Alle Strukturelemente liegen hintereinander im Speicher.

`Syntax: struct Strukturname`$_{\text{Opt}}$ {Datentype Strukturelementname; ... }OPT Variablenliste$_{\text{Opt}}$;

Ein Verbund entspricht einer Java Klasse mit dem Unterschied zu C (nicht C++) , dass ein Verbund keine Methoden beinhaltet und keine Zugriffsbeschränkung für die Attribute gesetzt werden können.

Der Syntax erlaubt es, gleichermaßen einen Datentyp (über Strukturname) zu definieren und Variablen von diesem Datentyp (über Variablenliste) anzulegen. Da beide Elemente optional sind, ergeben sich diverse Kombinationsmöglichkeiten:

- Nur Strukturname → Definition eines neuen Datentyps

```
struct xyz1 {int x; int y,z;};   //Definition eines neuen Datentyps
struct xyz1 var1;                //Definition einer Variablen dieses
                                 //Datentyps
var1.x=7;                        //Zugriff auf ein Strukturelement
```

C: Der Strukturname ist nur mit dem vorangestellten `struct` gültig. Der Strukturname stellt somit einen eigenen Namensraum, getrennt von dem Namensraum der Variablen/Funktionen, dar. Daher kann ein Strukturname identisch zu einem Variablennamen sein:

```
struct xyz2 {int x,y,z;};      //Definition des Datentyps 'struct xyz2'
struct xyz2 xyz2;              //Definition der Variablen 'xyz2' auf
                               //Basis des Datentyps 'struct xyz2'
xyz2 var2;                     //Compilerfehler, zur Nutzung des Datentyps
                               //'struct xyz2' muss struct vorangestellt
                               //werden!
```

C++: Der Strukturname ist sowohl mit als auch ohne dem vorangestellten `struct` gültig. Auch hier stellt der Strukturname einen eigenen Namensraum dar, der jedoch eine niedrigere Priorität als der der Variablen hat:

21 `https://de.wikipedia.org/wiki/Verbund%20%28Datentyp%29`

```
struct xyz3 {int x,y,z;};
xyz3 xyz3={1,1,1};        //'xyz3' beschreibt hier den Datentypen
xyz3.x=1;                 //und nach Definition einer Variable die Variable!
xyz3 var2;                //Fehler, xyz beschreibt hier die Variable
```

- Nur Variablenliste → Definition einer/mehrerer Variablen von einem 'unnamed' Datentyp

```
struct {
  int x,y,z;
} xyz,arr[3],*ptr=&xyz;  //Definition von Variablen des Datentyps
                         //struct {...}. Da dieser Datentyp nicht benannt
                         //wurde, kann im späteren keine weitere Variable
                         //von diesem Datentyp angelegt werden.

xyz.x    =8;             //Nutzung der Variablen
arr[1].y =9;
ptr->x   = arr[1].y;
```

Angewendet wird die Schreibweise gerne, wenn eine Struktur innerhalb einer Struktur definiert wird und die innere Struktur zur besseren Strukturierung dient:

```
struct aussen {         //Äußere Struktur
  struct {              //Innere Struktur
    int a,b,c;
  } anwendung1;
  struct {              //Innere Struktur
    int x,y,z;
  } anwendung2;
};
struct aussen var;

var.anwendung1.a=10;    //Nutzung der Variablen
var.anwendung2.y=12;
```

- Strukturname und Variablenliste → Definition eines Datentyps und Definition von Variablen. D.h. es können im Nachhinein weitere Variablen von diesem Datentyp angelegt und die hier angelegten Variablen genutzt werden.

```
struct xyz{                //Definition eines Datentyps
  int x,y,z;
} var1,*ptr1;              //und Definition von Variablen dieses Datentyps
struct xyz  var2,*ptr2;    //Definition weiterer Variablen dieses Datentyps

var1.x=var2.y;             //Nutzung der Variablen
```

- Kein Strukturname und keine Variablenliste → ”Anonymous Struct”, d.h. Definition eines 'unnamed' Datentyps, von der ergänzend keine Variable angelegt wird. Anonymus Structs sind nur innerhalb von Strukturen erlaubt/einsetzbar. Hier dienen sie der besseren Strukturierung und ersparen dem Programmierer die Benennung des normalerweise notwendigen Strukturelementes:

```
struct außen {
  int a;
  struct {
    int b,c,d;
  } anwendung1;   //'Normale' innere Struktur
  struct {
    int x,y,z;
  };               //Anonyme Struct als innere Struktur
} var;

var.a=4711;        //Nutzung der Variablen
var.anwendung1.b=1;//Bei Zugriff auf anonyme Struktur ist die Benennung
                   //des Strukturelementes notwendig
var.x=1;           //'Einfacherer' Zugriff auf inneres Element bei
                   //'anonymous struct'
```

Hinweise:

- Eine Struktur (und auch eine C++ Klasse) wird mit einem Semikolon abgeschlossen
- Eine alternative Beschreibung zu Strukturen finden sie in Structures in C: From Basics to Memory Alignment[22]

5.7.1 Strukturelement

Innerhalb der Struktur können beliebige Strukturelemente von beliebigen Datentypen angelegt werden. Einzige Voraussetzung, der Datentyp des anzulegenden Strukturelementes muss zuvor definiert worden sein. Ergänzend kann der neue Datentyp auch in der Struktur selbst definiert werden, sofern mit der Definition des Datentyps auch eine Variable angelegt wird:

```
struct abc {
  int a,b,c;
};                  //Definition des Datentyps struct abc
struct xyz {        //Definition des Datentyps struct xyz
  int x,y,z;        //Strukturelement vom Type int
  struct abc abc;   //Strukturelement vom Type struct abc
  struct def {      //Definition eines neuen Datentyps innerhalb
    int d,e,f;      //der Struktur bei gleichzeitigen Anlegen zweier
  } def,geh;        //Strukturelementen von diesem neuen Datentyp
  struct uvw {
    int u,v,w;      //Warning, reine Datentypdefinition innerhalb
  };                //einer Struktur nicht möglich
};
struct def abc;     //Innere Datentypbeschreibung auch
                    //außerhalb der Struktur nutzbar
```

5.7.2 Zugriff auf Strukturelemente

Die Programmiersprache C unterscheidet zwei 'Zugriffsarten' auf die Strukturelemente, abhängig vom zugrundeliegenden Datentyp:

- Handelt es sich beim zugrundeliegenden Datentyp um eine (struct)Variablen, so erfolgt der Zugriff über den Punkt-Operator .:

22 https://abstractexpr.com/2023/06/29/structures-in-c-from-basics-to-memory-alignment/

```
struct {
  int a,b,c;
} abc;
abc.a=4711;     //abc ist eine Variable einer Struktur
abc.b=abc.c;
```

- Handelt es sich beim zugrundeliegenden Datentyp um einen Zeiger auf eine (struct)Variable, so erfolgt der Zugriff über den Zeiger-Operator ->:

```
struct xyz {
  int x,y,z;
};
struct xyz  var1;  //Variable von Datentyp struct xyz;
struct xyz *ptr1;  //Zeiger auf eine Struktur vom Datentyp struct xyz
ptr1=&var1;        //Initialisierung des Zeigers
                   //(mit der Adresse der Variablen var1)
var1.x=  7;        //Zugriff auf das Strukturelement x über die Variable var1
ptr1->x=7;         //Zugriff auf das Strukturelement x über den Zeiger ptr1

(*ptr1).x=7;       //Der Zeiger-Operator entspricht dem Zugriff auf ein
                   //dereferenziertes Strukturelement
```

Werden innerhalb von Strukturen weitere Strukturen und Zeiger auf Strukturen angelegt, so müssen diese Regeln auf jedes innere Strukturelement einzeln angewendet werden:

```
struct abc {          //Äußere Struktur
 int a,b,c;
 struct xyz {
   int x,y,z;
  } xyz;              //Inneres Strukturelement xyz vom Datentyp struct xyz
  struct {
    char str[10];
  } strstr;           //Inneres Strukturelement strstr vom Datentyp
                      //struct strstr
  struct xyz *ptr;    //Inneres Strukturelement ptr vom Datentyp Zeiger
                      //auf struct xyz
} var2,*ptr;

var2.a=8;              //var2 vom Datentyp 'struct abc'. Zugriff über '.'
var2.xyz.x=1;          //xyz vom Datentyp 'struct xyz'. Zugriff über '.'
var2.strstr.str[0]='a';//strstr vom anonymes Struct. Zugriff über '.'

ptr=&var2;            //ptr mit einer Adresse initialisieren
var2.ptr = &var2.xyz; //Strukturelement ptr mit einer Adresse initialisieren

ptr->a='a';           //ptr vom Datentyp 'Zeiger auf struct abc'
                      //Zugriff über '->'
var2.ptr->x=3;        //ptr vom Datentyp 'Zeiger auf struct xyz'.
                      //Zugriff über '->'
ptr->xyz.x=3;
ptr->ptr->x=3;
```

5.7.3 Initialisierung von Strukturen

Mit dem Anlegen einer Strukturvariablen kann diese auch initialisiert werden. Eine 'Vorbelegung' der Strukturelemente bei der Definition der Struktur ist in C nicht möglich.

Die Initialisierung erfolgt über eine Initialisierungsliste (siehe Kap Grundlagen:Initialisierungsliste / Compound Literal[23]). Innerhalb der Initialisierungsliste stehen die Initialisierungswerte entweder in der Reihenfolge der Datentypdefinition oder werden explizit mit dem '.' Operator angesprochen (designated initializers). Werden einzelne Strukturelemente ausgelassen, so werden diese mit 0 initialisiert:

```
struct abc{
  int a,b;
  int c=3;    //KO: Kein Initialisierungswert von Strukturelementen möglich
  char str[10];
};
struct abc v1={1,2,"hallo"}; //Alle Strukturelemente werden initialisiert
struct abc v2={ 1,2 };       //Initialisierung der Strukturlemente a und b
                             //Rest wird mit 0 initialisiert
struct abc v3={ .b=3};  //Initialisierung einzelner/designated Strukturele.
                        //Rest wird mit 0 initialisiert
struct abc v4={.a=strlen(v1.str)}; //Initialisierungswert ergibt
                                   //sich erst zur Laufzeit.
                                   //Nur als lokale Variable möglich!
struct abc v5={.b=1,.a=2,.b=12}; //Nur C: Reihenfolge und Doppelbenennung
                                 //der Strukturelemente egal/möglich
struct abc v6={};  //Alle Strukturelemente mit 0 initialisieren
```

Eine Initialisierung einer Struktur über nachfolgende Art bewirkt etwas anderes, als erwartet:

```
struct abc { int a,b,c; };
struct abc var={var.b=12,var.c=3};
```

Innerhalb der Initialisierungsliste wird über 'var.b=12' und 'var.c=3' die zu diesem Zeitpunkt noch nicht initialisierte Variable var die Elemente b und c initialisiert. Mit dem Werten der Initialisierungsliste (hier {12,3}) wird nachfolgend die Variable erneut initialisiert und 12 dem Strukturelement a und 3 dem Strukturelement b zugewiesen. Abhängig vom Compiler ist das Strukturelement c entweder 3 oder 0.

Bei verschachtelten Strukturen muss entsprechend obiger Aussage für jede innere Struktur eine eigene Initialisierungsliste erstellt werden. Die inneren Initialisierungslisten können entfallen, was jedoch nicht empfohlen wird und ergänzend vom Compiler bei '-Wall' als Warning angemerkt wird:

```
struct abc {
  int a,b;
  struct xy {
    int x,y;
  } xy;
  struct xy arr[2];
};
struct abc v1={     //Initialisierung über Reihenfolge
  1,2,
  {3,4},
  { {5,6},{7,8}}
};
```

23 Kapitel 4.11 auf Seite 58

```
struct abc v2={     //Initialisierung über 'designated initializers'
  .arr={{.x=5,.y=6},{7,8}},
  .xy={3,4},
  .a=1,.b=2
};
struct abc v3={     //Fehlende innere Initialisierungsliste
  1,2,
  3,4,
  5,6,7,8
};
```

5.7.4 Zuweisen/Kopieren von Strukturen

Strukturen werden über den Namen als 'ganzes' angesprochen, so dass eine Zuweisung als 'ganzes' möglich ist. Bei bspw. einer 12-Byte großen Struktur werden bei Nutzung der Variable die gesamten 12-Byte gelesen/geändert!

```
struct xyz {int x,y,z;} var1,var2;
var1=var2;           //12-Bytes kopiert
//--> entspricht memcpy(&var1,&var2,sizeof(xyz));
```

Die Zuweisung als 'ganzes' gilt auch bei der Parameterübergabe und -rückgabe von Funktionen:

```
struct xyz {int x,y,z;};
//Funktionsdefinition
struct xyz function(struct xyz par) {
  return (struct xyz){par.z,par.y,par.x};
}
//Funktionsaufruf
struct xyz var=function((struct xyz){1,2,3});
//Mit dem Aufruf der Funktion werden 12-Bytes in die Variable par kopiert
//Mit dem Ende der Funktion werden 12-Bytes in die Variable var kopiert
```

Bedenke, dass bei großen Strukturen das Kopieren Rechenzeit benötigt und folglich vermieden werden sollte! Alternativ sollten Zeiger auf Strukturen über- und zurückgeben werden.

Über Compound Literal (siehe Kap. Grundlagen:Initialisierungsliste / Compound Literal[24]) kann einer Strukturvariablen als 'ganzes' einen neuen Wert zugewiesen werden:

```
struct abc {int a,b,c;};
var1={.a=7};                //KO, Initialisierungsliste kann hier nicht
                            //angewendet werden
var1=(struct abc){.a=7};    //Wertezuweisung über Compound Literal möglich
```

Formel gesehen entspricht das Compound Literal einer 'unnamed Variable', die als erstes angelegt und initialisiert wird. Der Inhalt dieser Variablen wird im Anschluss der eigentlichen Variable zugewiesen!

5.7.5 Prototyp / Deklaration einer Struktur

Soll der Datentyp einer Struktur genutzt werden, der erst an weiter hinten liegenden Stellen im Programm definiert wird, so ist wie bei Variablen/Funktionen ein Prototyp/Deklaration notwendig:

24 Kapitel 4.11 auf Seite 58

```
struct abc;                //Prototyp der Struktur abc
...
struct abc { int a,b,c;};  //Definition der Struktur abc
```

Die Deklaration sagt einzig aus, dass die Struktur später definiert wird, beinhaltet aber nicht die Strukturelemente. Folglich kann auf Basis dieses Prototyps keine Variable definiert werden, sondern einzig ein Zeiger auf solche eine Struktur:

```
struct xyz;           //Deklaration / Prototyp des Datentyps
                      //    d.h. keine Benennung der Strukturelemente
struct xyz var1;      //KO, es kann auf Basis der Deklaration keine
                      //    Variable angelegt werden
struct xyz *pttr1;    //OK, von einer Strukturdeklaration kann ein Zeiger
                      //    angelegt werden!
```

Anwendung:

- Struktur, welche auf sich selbst verweist (Verkettete Liste)

```
struct vl {            //Entspricht gleichermaßen einem Prototyp, so dass
                       //innerhalb dieser Struktur dieser Datentyp zum
                       //Anlegen eines Strukturelementes genutzt werden kann
    struct vl  *next;  //Zeiger auf das nächste Element
    char daten[100];
};
```

- Entsprechend der objektorientierten Programmierung, wenn alle Attribute der Klasse private sind:

class.h

```
struct class;  //Prototyp für Struktur
               //so dass Nutzer dieser
//Struktur ein Zeiger auf diese anlegen,
//aber nicht auf die Strukturelemente
//zugreifen können.

//Prototyp der public Methoden
void konstruktor(struct class ** me);
```

main.c

```
#include "class.h"

int main(int argc, char*argv[])
{
  struct class *obj1;
  konstruktor(&obj1);
  ..
  //Kein Zugriff auf
  //Strukturelemente möglich
  return 0;
}
```

class.c

```
#include "class.h"
struct class {
  int attr1;
  int attr2;
};
void konstruktor(struct class ** me)
{
  struct class *this;
  this=(struct class *)
        malloc(sizeof(struct class));
  this-
>attr1=10;
  *me=this;
}
```

5.7.6 Vergleichen von Strukturen

Ein Vergleich von Strukturen über den direkten Weg ist nicht möglich. Vielmehr müssen die Strukturelemente händisch verglichen werden:

```
struct xyz {int inx,y,z;} var1,var2;
if(var1 == var2)  //KO, ein Vergleich von Strukturvariablen ist nicht möglich

//Manueller Vergleich über den Vergleich der Strukturelemente
if(var1.x == var2.x && var1.y == var2.y && var1.z == var2.z)
```

Hinweis

- Bei Rechnerarchitekturen mit nicht ausgerichteter Speicherausrichtung (Alignment) kann ergänzend ein Vergleich über memcmp() erfolgen

5.7.7 Speicherplatzbedarf einer Struktur

Für jedes Strukturelement wird Speicher entsprechend der Größe des Datentyps reserviert. Der Speicherplatzbedarf der Gesamtstruktur ergibt sich aus der Summe der Strukturelemente:

```
struct xyz{
  int x;   //Strukturelement belegt 4 Byte Speicher
  int y;   //Strukturelement belegt 4 Byte Speicher
  int z;   //Strukturelement belegt 4 Byte Speicher
}var1;   //Speicherplatz der Struktur = 4+4+4 = 12 Byte
sizeof(var1)       //ergibt 12
sizeof(struct xyz) //ergibt 12
```

Die Ermittlung der tatsächlichen Speichergröße einer Struktur erfolgt über den sizeof-Operator:

```
struct xyz{int x,y,z;} a;
sizeof(a);                    //=12 OK
sizeof(a.x);                  //=4  OK
sizeof(struct xyz);           //=12 OK
sizeof(struct xyz.x);         //    KO
//Ist dies dennoch notwendig, so kann dies über den Umweg eines Zeigers
//erfolgen:
sizeof(((struct xyz *)0)->x); //=4  OK
```

Hinweis:

- Bei Rechnerarchitekturen mit ausgerichteter Speicherausrichtung (Alignment) (siehe w:Speicherausrichtung[25]) kann der tatsächliche Speicherbedarf größer sein! Hier fügt der Compiler ggf. zwischen den Strukturelemente Füllbytes (Padding Bytes) ein
- Die Reihenfolge der Strukturelemente wird vom Compiler nicht geändert. D.h. die Zuordnung der Speicherstellen zu den Strukturelementen erfolgt in der Definitionsreihenfolge

25 https://de.wikipedia.org/wiki/Speicherausrichtung

5.7.8 Interne Organisation

Compilerintern werden die Strukturelemente über einen Offset dargestellt. Das erste Strukturelement bekommt dabei immer den Offset 0 zugewiesen. Das nachfolgende Element bekommt als Offset die Größe des Datentyps des vorherigen Elementes zugewiesen. Usw..

Der Compiler setzt den Zugriff auf ein Strukturelement einer (Struktur)Variablen so um, dass er zunächst die Startadresse der Variablen holt und zu dieser den Offset des Strukturelementes addiert. Die Anzahl der zu lesenden/schreiben Bytes ergibt sich aus dem Datentyp des Strukturelementes:

```
struct xyz {         //Das Anlegen der Variable var bedeutet,
  int x,y,z;         //12Byte Speicher zu reservieren.
} var1;              //(Beispielhaft soll var1 die Speicheradressen
                     //0x100..0x10B belegen). Der Zugriff auf
                     //die Variable erfolgt immer über die Startadresse
var1.x=7;            //Zugriff auf Speicheradresse 0x100+0 (Offset)
var1.y=8;            //Zugriff auf Speicheradresse 0x100+4 (Offset)
var1.z=9;            //Zugriff auf Speicheradresse 0x100+8 (Offset)
```

Der Offset eines Strukturelementes kann mit dem offsetof-Operator ermittelt werden:

`Syntax: offsetof(type,Strukturelement)`

Wie beim sizeof-Operator gilt der offsetof-Operator als Konstantenausdruck und der Rückgabedatentyp ist `size_t`. Zur Nutzung des Operators muss die Header-Datei 'stddef.h' inkludiert werden:

```
#include <stddef.h>  //Zur Nutzung des OffsetOf Operators
struct xyz {int x,y,z;} var1;
offsetof(struct xyz,z)  --> 8
offsetof(var1,z)        --> KO: Es wird ein Datentyp und
                                keine Variable erwartet
```

5.7.9 Explizites Cast

Eine Struktur kann nicht in einen anderen Datentyp und damit auch nicht in einen anderen Strukturdatentyp mit identischen Strukturelementen konvertiert werden:

```
struct xyz {int x,y,z;} xyz;
struct abc {int a,b,c;} abc;
int var;

xyz=abc;                  //KO, Datentyp struct xyz != struct abc
xyz=(struct xyz)abc;      //KO, siehe zuvor

int neu1 = (int) xyz;     //KO
int neu2 = (int) xyz.x;   //OK (Das Strukturelement x ist vom Datentyp int)
struct xyz neu2 = (struct xyz)3;  //KO
```

5.7.10 Incomplete Array

Das letzte Element einer Struktur kann ein Array ohne Angabe einer Dimensionsgröße sein (Incomplete Array/Flexible Array Member). Bei der Definition einer Variablen von solch einem Datentyp wird in der Tat kein Speicher für das letzte Element reserviert, so dass dies 'händisch' erfolgen muss. Auf das Array kann ungeachtet dessen unproblematisch zugegriffen werden:

```
struct vl {
  struct vl *next;
  char data[];   //incomplete Array / flexible Array Member
};
struct vl a;      //Es wird nur Speicher für den next Zeiger reserviert
a.data[0]='7';    //Syntax OK, jedoch erzeugt dies ein Laufzeitfehler,
                  //da kein Speicherplatz für das Incomplete Array reserviert
                  //wurde
```

Die Speicherplatzreservierung für das incomplete Array kann auf zwei verschiedene Arten erfolgen:

- Speicherplatzreservierung für das Incomplete Array über Variableninitialisierung (nur GCC-C und nur im Falle einer globalen Variablen)

```
struct vl ele = {
  .next = NULL,
  .data = {32, 31, 30}
};  //Vorsicht, sizeof(ele) gibt dennoch nur 4/8 zurück
```

- Speicherplatzreservierung für das Incomplete Array über malloc

```
struct vl *b=malloc(sizeof(struct vl)+  //Größe der Struktur
                    sizeof(char[3]  )); //Größe des Arrays
b->data[1]='7';           //OK, da mit dem Anlegen von b Speicher für 2
                          //data Element reserviert wurde
```

Das Incomplete Array bietet sich überall dort an, wo Daten gespeichert werden sollen, deren Größe sich erst zur Laufzeit ergibt.

5.7.11 Anwendung

Die Nutzung des Datentyps struct empfiehlt sich an vielen Stellen:

- Zusammengehörende Daten zu Kapseln (entsprechend der objektorientierten Programmierung
- Verkette Listen
- Aufgrund der Typsicherheit zur Darstellung von sicherheitskritischen Aufgaben

Siehe Übungsbereich!

5.7.12 C++

Der wesentliche Syntax von Strukturen wurde in C++ übernommen. D.h.:

- das abschließende Semikolon
- der Zugriff auf die Strukturelemente
- Speicherplatzreservierung
- die Definition von Variablen mit der Definition des Datentyps
- ...

Ergänzend wurde in C++ der Datentyp class eingeführt, der weitestgehend identisch zu struct ist. Geändert/Hinzugefügt wurden folgende Sachverhalte:

- Classen/Strukturen können Methoden haben
- Operatoren (Zuweisungsoperator, Addition, ...) können überladen werden
- Alle Strukturelemente (und Methoden) einer struct sind per default public
- Alle Attribute (und Methoden) einer Class sind per default private
- Zur Nutzung des Datentyps ist das führende struct nicht notwendig (dies bedingt dann auch, dass der Strukturdatentyp kein separater Namensraum ist)

```
struct xyz {
  int x,y,z;
};
xyz var_xyz;
```

- Initialisierungswerte für Strukturelemente und Klassenattribute bei der Datentypbeschreibung angegeben werden können:

```
struct xyz {
  int x=3;
  int y=1;
  int z;
};
xyz var={7};    //var.x=7   var.y=1  var.z=0
```

- In C++ kann eine Struktur ebenfalls über 'designated initializers' initialisiert werden. Jedoch muss die Reihenfolge der Initialisierungselement identisch zur Datentypdefinition sein. Auch ist keine Doppeltbenennung erlaubt:

```
struct xyz {
  int x,y,z;
};
xyz var1={.x=1, .y=2, .z=3};  //OK, Reihenfolge identisch
xyz var2={.y=1, .z=3};        //OK, Reihenfolge identisch
xyz var3={.y=1, .x=0};        //KO, Reihenfolge nicht identisch
xyz var4={.x=1, .x=2};        //KO, keine Doppeltbenennung möglich
```

5.8 Union

Ähnlich wie eine Struktur ist ein Union ein Datentyp, der aus einem oder mehreren Datentypen zusammengesetzt wird. Bei sog. Union beginnen jedoch alle Komponenten nicht wie bei Strukturen an nacheinander folgenden Speicheradressen, sondern an der identischen Speicheradresse, d.h. ihre Speicherbereiche überlappen sich ganz oder zumindest teilweise. Eine Union kann folglich zu einem Zeitpunkt nur ein Element enthalten. Der benötigte Speicherplatz ergibt sich aus der größten Komponente.

`Syntax: union Unionname`$_{\text{Opt}}$ {Datentyp Variablenname; ...}$_{\text{Opt}}$ Variablenliste$_{\text{Opt}}$;

Ein Schutz/Zugriffssteuerung der Unionelemente ist nicht vorhanden. Es kann in beliebiger Reihenfolge auf die einzelnen Elemente zugegriffen werden.

Die Anwendung ist identisch wie bei Strukturen, so dass im Folgenden nur die Abweichungen zu den Strukturen beschrieben werden.

5.8.1 Speicherplatzbedarf und interne Struktur einer Union

Die Größe der Union ergibt sich aus dem größtem Unionelement. Alle Unionelemente überlappen sich, so dass der Offset zum Basiselement 0 ist:

```
union abc {
  char  a;
  short b;
  int   c;
} abc;
printf("Größe:   %zd\n",sizeof  (union abc));   //-> 4
printf("Offset a:%zd\n",offsetof(union abc,a)); //-> 0
printf("Offset b:%zd\n",offsetof(union abc,b)); //-> 0
printf("Offset c:%zd\n",offsetof(union abc,c)); //-> 0
```

Wird ein kleineres Unionelement belegt und im Anschluss ein größeres Unionelement gelesen, so ist der Inhalt der durch das kleinere Element nicht beschriebenen Speicherstellen undefiniert:

```
union abc {
  char a;
  short b;
  int   c;
} abc;
abc.a=0x11;
printf("%08x",abc.c);
```

5.8.2 Initialisierung von Union

Bei einer Union kann nur ein Element initialisiert werden, d.h. die Initialisierungsliste kann nur einen Wert beinhalten. Ohne explizite Benennung des Unionelementes in der Initialisierungsliste wird das erste Element aus der Datentypbeschreibung initialisiert. Mit expliziter Benennung mittels 'designated initializers' können auch andere Elemente initialisiert werden:

```
union abc {
  char a;
  short b;
  int c;
} abc;

abc=3;                //KO keine Initialisierungsliste
abc=(union abc){3};    //OK Initialisierung des ersten Elementes a
abc=(union abc){.b=3}; //OK Initialisierung des Elementes b
abc=(union abc){3,4};  //KO nur ein Initialisiuerngswert erlaubt
```

5.8.3 Anwendung

Mittels des Datentyps union können diverse Anwendungsfälle abgedeckt werden:

- Über Union lassen sich größere Datentypen in kleinere Datentypen unterteilen, um z.B. einzelne Bytes zu extrahieren:

```
union floatu {
  float        var;       //Float ist 4-Byte groß
  unsigned int hex;       //Integer ist 4-Byte groß
  char         byte[4];   //Ohne Worte
};
union floatu var={1.234};
printf("%f\n",var.var);   //Darstellung des Float-Wertes
printf("%x\n",var.hex);   //Darstellung des Float-Wertes 'BinäreZahl'
printf("%hhx %hhx %hhx %hhx\n",   //Darstellung der einzelnen Bytes
  var.byte[0],var.byte[1],var.byte[2],var.byte[3]);

union longlong {
  long long ll;       //Long Long ist 8-Byte groß
  int       i[2];     //Ohne Worte
  short     s[4];
  char      c[8];
};
union longlong var2={0x123456789ABCDEFULL};
printf("%llx\n",var2.ll);
printf("%x %x\n",var2.i[0],var2.i[1]);
//Vorsicht: Die Aufteilung der Bytes ist von der Rechnerarchitektur
//(Endianes) und von der Ausrichtung durch den Compiler abhängig
```

- Ein weiterer Anwendungsfall ergibt sich, wenn in einer Struktur unterschiedliche Datensätze gespeichert werden sollen:

```
struct set {
  char *index;
  char *value;
};
struct get {
  char *index;
  char *value;
};
struct cli{
  //Enum zur Darstellung des aktiven Unionelementes hilfreich
  enum {SETTER,GETTER} tag;
  union {                //Interpretation abhängig von tag
    struct set set;
    struct get get;
  } ;
};
struct cli var={.tag=SETTER, .set.index="hallo"};

if(var.tag==SETTER)
  printf("Set: %s",var.set.index);
else
  printf("Get: %s",var.get.index);
```

5.8.4 C++

Die Eigenschaften des Datentyps union entsprechen den Eigenschaften des Datentyps struct in C++.

5.9 Enum/Aufzählungstyp

Nach dem Wikipedia Artikel w:Aufzählungstyp[26] ist *ein Aufzählungstyp (englisch enumerated type) ein Datentyp für Variablen mit einer endlichen Wertemenge. Alle zulässigen Werte des Aufzählungstyps werden bei der Deklaration des Datentyps mit einem eindeutigen Namen (Identifikator) definiert, sie sind Symbole.*

`Syntax: enum enumname`$_{\text{Opt}}$ {definition-list[=expression]}$_{\text{Opt}}$ Variablenliste$_{\text{Opt}}$

Die Anwendung ist identisch wie bei Strukturen, so dass nachfolgend nur die Abweichungen zu den Strukturen beschrieben werden.

5.9.1 Datentyp von enum

In C entspricht der Datentyp Enum dem Datentyp `int` so dass Zuweisungen/Vergleiche mit Ganzzahlen möglich sind. Ganzzahloperationen funktionieren ebenso:

```
enum STATUS {OK,KO=5};  //Definition des Datentyps
enum STATUS status;     //Definition einer Variable dieses Datentyps
status=OK;
if(status == 5)
status++;
status=5.7;
```

5.9.2 Enumelemente

Die Elemente der Definitionsliste sind Integerkonstanten. Diese können auch in Kombination mit anderen Datentypen genutzt werden. Auch gibt es keinen gesonderten Namensraum für Elemente der Definitionsliste. Sie 'konkurrieren' folglich mit allen Variablen- und Funktionsnamen:

```
enum mode {OK, KO};
enum {FIRST,SECOND,LAST};  //Anonymes Enum!
enum {EINS,ZWEI,LAST};     //KO, Symbolname LAST bereits vergeben
enum mode var1=OK;
var1=4711;           //OK, enum entspricht Datentyp int
int var2=KO;         //OK, EnumElement entspricht Datentyp int
int OK=KO;           //KO, Symbolname OK bereits für EnumElement vergeben
var1=FIRST;          //OK, var1 und FIRST sind zwar unterschiedlich
                     //Enumtypen,jedoch entspricht beides dem Datentyp int
```

Der erste Enumelement der Definitionsliste bekommt den Wert 0 zugewiesen. Folgeelemente bekommen den Wert des Vorgängerelementes +1 zugewiesen. Enumelemente können mit einer Konstanten (und Konstantenausdruck) initialisiert werden:

```
enum {bill=10, john=bill+2, fred=john+1} ;
//Negative Werte sind möglich
enum {error=-3, warning, info, ok};   //warning=-2  info=-1 …
//Doppelte Wertzuweisung sind möglich
enum {Ostfalia=10,Wolfenbuettel=10};
//Hinter dem letzten Enumelement kann ein Komma folgen
enum {test=10,};
```

Hinweis:

26 https://de.wikipedia.org/wiki/Aufz%C3%A4hlungstyp

- Es empfiehlt sich, die Enumelemente in GROSSBUCHSTABEN zu schreiben. Hiermit wird gekennzeichnet, dass es sich um Konstanten und nicht um Variablen/Funktionen handelt

5.9.3 Anwendung

Die Nutzung des Datentyps enum empfiehlt sich überall dort, wo eine Fallunterscheidung notwendig ist:

- Statusrückgabe von Funktionen:

```
enum STATUS {OK,MEMORY_OVERFLOW, DIVISION_BY_ZERO};
enum STATUS funct(…) {
    return MEMORY_OVERFLOW;
}
```

- Beschreibung der Zustände und der Ereignisse eines Zustandsautomates:

```
enum EREIGNIS {EREIGNIS1,EREIGNIS2,EREIGNIS3};
void zustandsautomat(enum EREIGNIS ereignis) {
  static enum {IDLE,OPERAND,OPERATOR,} zustand=IDLE;
  switch(zustand) {
    case IDLE:
    case OPERAND:
      //Compiler meckert bei fehlendem Case über OPERATOR
  }
}
```

5.9.4 C++

Ergänzend zu den Abweichungen bei Structs sind hier weitere Unterscheidungsmerkmale vorhanden:

- Enums stellen einen eigenen Datentyp dar, der nicht mit int kompatible ist. Das bedingt unter anderem, dass einige Operatoren wie z.B. ++ nicht mehr auf Variablen des Datentyps enum angewendet werden. Die Enumelemente als solches sind jedoch kompatibel zum Ganzzahldatentyp:

```
enum Color {RED,GREN,BLUE};
enum Color var1=RED;
var1=4;              //KO
var1=var1+1;         //KO
var1++;              //KO
int var2=RED;        //OK
if(RED==2)           //OK
```

- Auch enums können Methoden besitzen.
- Über Operatorüberladung können z.B. nicht vorhandene Operatoren wie ++/− ergänzt werden
- Über unscoped/scoped Enumerations wird der 'Namensraum' gesteuert:

- Unscoped Enumeration (Elemente der Definitionsliste stehen wie bei C im Zugriff)

Syntax: `enum` $\texttt{name}_{\text{Opt}}$: type_{OPT} {enumerator=constexp, ...};

```
enum Color {RED,GREN,BLUE };
Color r = RED;
```

- Scoped enumeration (Elemente der Definitionsliste haben einen eigenen 'Namensraum')

Syntax: `enum class|struct` $\texttt{name}_{\text{Opt}}$: type_{OPT} {enumerator=constexp, ...};

```
enum class Color {RED,GREN=20,BLUE };
Color r = Color::BLUE;
```

5.10 Bitfelder

Nach dem Wikipediaartikel w:Bitfeld[27] *bezeichnet in der Informationstechnik und Programmierung ein Bitfeld ein vorzeichenloses Ganzzahldatentyp, in dem einzelne Bits oder Gruppen von Bits aneinandergereiht werden. Es stellt eine Art Verbunddatentyp auf Bit-Ebene dar. Im Gegensatz dazu steht der primitive Datentyp, bei dem der Wert aus allen Stellen gemeinsam gebildet wird.*

Der Syntax entspricht dem struct-Syntax, mit der Ergänzung, dass hinter den Strukturelementen noch die Bitbreite getrennt durch ein Doppelpunkt angegeben wird:

```
struct time {
  unsigned int hour:   5;   //0..23  Bits 0..4
  unsigned int minute: 6;   //0..59  Bits 5..10
  Unsigend int second: 6;   //0..59  Bits 11..16
} myTime;
```

Als mögliche Datentypen für die Strukturelemente stehen nur die Ganzzahldatentypen int, short, long , long long und char zur Verfügung. Der Datentyp eines Strukturelementes muss mindestens die Anzahl der Bits enthalten, wie diese über die Bitbreite gefordert wird. Von den Strukturelementen kann keine Adresse bestimmt werden! Auch der `offsetf` - Operator schlägt fehl.

Der Datentyp entspricht im Wesentlichen einem Ganzzahlzahldatentyp, so dass bei Zugriff auf eine Variable dieses Datentyps eine Ganzzahl gelesen/geschrieben wird. Bei Zugriff auf die einzelnen Strukturelemente werden die entsprechenden Bits dieser Ganzzahlvariablen maskiert, so dass nur die betroffenen Bits geändert werden:

```
struct test {
  unsigned char first:2;
  unsigned char second:4;
  unsigned char third:2;
};
struct test var={.first=1,.second=1,.third=1};
```

27 https://de.wikipedia.org/wiki/Bitfeld

```
//entspricht
unsigned char var1=0b01000101;

var.second=1;
//entspricht:
var1=(var1&0b11000011) | (1<<2);

var.second++;
//entspricht
int second =(var1>>2)&0b00001111;
second++;
var1=(var1&0b11000011) | ((second&0b00001111)<<2);
```

5.10.1 Anwendung

- Komprimierte Speicherung der Uhrzeit (wie dies z.B. in Realtimeclocks erfolgt)

```
union {
  struct {             //Datentyp zum 'Bitweisen' Zugriff
    unsigned int hour:5;
    unsigned int minute:6;
    unsigned int second:6;
    };
  unsigned int hex;    //Datentyp zum Zugriff aller Elemente
} myTime;
myTime.hour   = 13;
myTime.minute = 37;
myTime.second = 59;
printf("Zeit: %02d:%02d:%02d\n", myTime.hour,myTime.minute,myTime.second );
//Alternativer/Händischer Zugriff auf die einzelnen Bits
printf("Es ist jetzt %02d:%02d:%02d Uhr\n",((myTime.hex>> 0)&0b011111),
                                           ((myTime.hex>> 5)&0b111111),
                                           ((myTime.hex>>11)&0b111111));
```

- Aufteilung von FloatingPoint Zahlen in seine Komponenten

```
union float_mes {
  float flo;
  struct ieee { //aus ieee754.h kopiert
#if __BYTE_ORDER__ == __ORDER_BIG_ENDIAN__
    unsigned int negative:1;
    unsigned int exponent:8;
    unsigned int mantissa:23;
#endif /* Big endian.  */
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
    unsigned int mantissa:23;
    unsigned int exponent:8;
    unsigned int negative:1;
#endif /* Little endian.  */
  } ieee;
  int hex;
};
//aus ieee754.h kopiert
#define IEEE754_FLOAT_BIAS    0x7f /* Added to exponent.  */
union float_mes test={0.1};
printf("float: %f\n",test.flo);
printf("hex:   %x\n",test.hex);
printf("bin:   ");
for(unsigned int flag=0x80000000; flag; flag=flag>>1)
  printf("%c%s",test.hex&flag?'1':'0',
```

```
          flag&0x80800000?":":(flag&1?"\n":""));
  printf("bitfield: (%c)%x * 2^(%d-%d)\n",(test.ieee.negative==1?'-':'+'),
                                          test.ieee.mantissa,
                                          test.ieee.exponent,
                                          IEEE754_FLOAT_BIAS);
```

5.10.2 C++

Bitfelder existieren mit den bekannten Ergänzungen auch in C++.

6 Datatype-/Storage Class Specifier

6.1 Typedef Storage Class Specifier

Anweisung zum Erstellen eines Alias für einen Datentyp.

`Syntax: typedef T type_ident[,type_ident];`

`T` ist der zu ersetzende Datentyp. `Type_ident` ist der Alias, wobei auch mehrere mit Komma separierte Aliase gesetzt werden können. Der Syntax entspricht einer normalen Variablendefinition mit vorangestellten `typedef`. Aufgrund des vorangestellten wird nun keine Variablen angelegt, sondern der 'Variablenname' ist der Alias.

Auf Basis des Alias können in Anschluss beliebig viele Variablen definiert werden:

```
typedef unsigned long long ull;
ull var1;

typedef signed long     sl1,sl2,*psl;
sl1   var2;               //--> signed long var2;
psl   var3;               //--> signed long *var3;

struct xyz {
  ull x,y,z;
};
typedef struct xyz   xyz_t;
xyz_t  var4;              //--> struct xyz var4;

typedef struct abc { int a,b,c;} abc_t;
abc_t       var5;   //Definition über Alias
struct abc  var6;   //Definition über struct abc

typedef struct {int d,e,f;}  def_t;  //Unnamed Struct
def_t       var7;

typedef int arr_alias[4];
arr_alias arr;

typedef void func_alias(void);  //Alias für eine Funktion
func_alias *func;               //Zeiger auf Funtion
```

Anwendungsfälle von typedef sind:

- Source-Code unabhängig von der Rechnerarchitektur zu entwickeln:

```
//Bitbreite des Datentyps int ist von der Rechnerarchitektur abhängig
#if ARCHITEKTUR == 16
typedef unsigned int uint16_t;
#elif ARCHITEKTUR == 32
typedef unsigned short uint16_t;
#endif
//Diese Definition sind in der Header-Datei stdint.h enthalten!
```

- Weniger Tipparbeit:

```
//Vermeidung des führende struct/union/enum
typedef enum {OK,KO,var11}  STATUS_T;
STATUS_T  var1;

//Vermeidung der Angabe weiterer Specifier
typedef const volatile unsigned int cvuint16_t;
cvuint16_t var2=4711;
```

- Dem Datentyp über seinen Namen eine Bedeutung zuzuordnen:

```
typedef  char * cpointer;
typedef struct { int raeder; enum {ROT,GELB}farbe;} auto_t;
typedef  void (*fptr_t)(void);
```

Hinweise:

- **Zur Verdeutlichung, dass es sich bei dem Alias nicht um eine Variable oder Funktion handelt, wird oftmals der Suffix '_t' an den Alias angehängt**
- Der Alias konkurriert mit Variablen und Funktionsnamen, kann also nicht doppelt benutzt werden
- Wird ein Typedef innerhalb eines Blockes definiert, so ist dieser nur innerhalb des Blockes gültig

6.2 Internal/External Linkage

C-Projekte bestehen aus mehreren C-Dateien, welche beim Compilieren in Objektdateien gewandelt werden und über den Linker zu einer ausführbaren Datei zusammengebunden werden. Der Zugriff auf Inhalte 'externer' Objektdateien wird über die 'internal', 'external' und 'none Linkage' gesteuert, wobei none gleichzusetzen mit external Linkage ist. External Linkage bedeutet, dass solche Variablen/Funktionen im gesamten Projekt (bestehend aus div. Objektdateien und Libraries) in Zugriff steht und somit nur einmal existiert (= globale Variable/Funktion). Internal Linkage bedeutet, dass die Variable/Funktionen nur in der aktuellen Objekt-/C-Datei im Zugriff stehen.

6.2.1 External Linkage / Extern Storage Class Specifier

Mit dem Storage Class Specifier `extern` wird ausgesagt, dass die Variable/Funktion 'external linkage' hat (also projektweit im Zugriff steht) und an dieser Stelle kein Speicherplatz für die Variable/Funktion definiert wird. Sie entspricht folglich dem Funktions-Prototyp mit der expliziten Aussage, dass diese Variable/Funktion 'anderweitig' definiert ist.

```
Syntax: extern Datentyp Variablenname;
        extern Datentyp Funktionsname(Datentyp,Datentyp);
```

Beispiel:

datei1.c

```
extern int var; //Zugriff auf
                //'externe' Variable
extern int func(int par); //Zugriff
          //auf 'externe' Funktionen

int main(int argc, char *argv[])
{
  extern int var2;  //Zugriff auf
            //'externe' vVariable
  func(var+var2);
}
```

datei2.c

```
int var;  //Definition der
          //'externen' Variable
int var2=8; //Definition der
            //'externen' Variable

int func(int par) //Definition der
{                 //'externen' Funkt.
    return(par*2);
}
```

Das Weglassen des Schlüsselwortes extern entspricht der Definition einer Variable/Funktion, d.h. der Speicherreservierung. Die Linkage ist in diesem Fall ebenfalls extern. Die Extern Anweisung wird benötigt:

- wenn auf Funktionen / Variablen zugegriffen werden sollen, die erst weiter hinten im Code definiert werden und projektweit im Zugriff stehen
- wenn auf Funktionen / Variablen zugegriffen werden, die in anderen Dateien enthalten sind
- wenn auf Funktionen / Variablen zugegriffen werden, die in Libraries enthalten sind
- wenn auf Funktionen / Variablen zugegriffen werden, die mit einer anderen Sprache übersetzt wurden

Hinweise:

- Wenn in zwei unterschiedlichen Dateien eines Projektes je eine initialisierte globale Variable mit identischen Namen angelegt wird, so gibt der Linker den Fehler ”multiple definition” dieser Variablen aus
- Wie bereits erwähnt wurde Unix und C parallel entwickelt und einer der neuen Features von Unix war die Bereitstellung eines Linkers. Der Extern Storage Class Specifier ist eine Anweisung an den Linker, an den Stellen der Nutzung der Variablen/Funktion in der eigenen Objektdatei die Adresse der eigentlichen Variablen/Funktion aus einer anderen Objektdatei einzutragen

6.2.2 Internal Linkage / Static Storage Class Specifier

Die genaue Bedeutung ist abhängig davon, ob es sich um eine globale Funktion/Variable oder einer lokalen Variable handelt.

Static globale Variable / Static Funktionen

Weist den Compiler an, dass diese Variable/Funktion nur innerhalb dieser C-Datei genutzt wird (internal Linkage) und für anderen C-/Objekt-Dateien nicht im Zugriff stehen. Static entspricht dem Zugriffsmodifikator 'public' aus der objektorientierten Programmierung mit dem Unterschied, das der Zugriff auf diese Variable/Funktion nicht innerhalb des Objektes,

sondern innerhalb der Datei beschränkt ist.

Ergänzend dient static der besseren Lesbarkeit des Codes (Zugriff auf diese Variable/Funktion nur aus dieser Datei). Auch der Compiler kann den Zugriff auf solche Variablen/Funktionen optimieren und schnelleren/kompakteren Code erzeugen.

Im Sinne der C-Spezifikation steht der Static Storage Class Specifier dafür, die Variable auf internal Linkage zu setzen (diese nach außen zu anderen Objektdateien nicht bekannt zu machen). Extern im Gegenzug steht für external Linkage. Die Nutzung beider Anweisungen zusammen widersprechen sich folglich und führen zu Compilerfehler. Auch bedeutet static, dass hiermit Speicherplatz reserviert wird (entgegen der 'extern' Anweisung, die eher einem Prototyp entspricht).

Für eine statische lokale Variable kann kein Prototyp erzeugt werden. Solche Variablen müssen immer vor der ersten Verwendung definiert werden! Beispiele:

- Nutzung von static Variablen/Funktionen innerhalb einer Datei

```
//Prototypen basierend auf den unten stehenden statische Variablen/Funktionen
static void func(void);         //OK
extern void func(void);         //KO (Prototyp muss ebenfalls static sein)
extern static void func(void);  //KO (extern und static wiedersprechen sich)
void func(void);                //KO (Prototyp muss ebenfalls static sein)
extern static int var1;         //KO a)extern und static widersprechen sich
                                //   b)von einer static Variable kann kein
                                //      Prototyp angelegt werden!

int main(int argc, char *argv[]) {
  func();
  var1=1;                     //KO, da kein Prototyp möglich ist
  return 0;
}
//Definition von statische Variablen/Funktionen
static int var1;             //Static Variable
static void func(void) {     //Static Funktion
  printf("var1=%d",var1);
}
```

- Nutzung von static Variablen/Funktionen innerhalb mehrerer Dateien

  Static entspricht dem private Zugriffsmodifikatior, mit dem Unterschied dass diese Variablen/Funktionen nur innerhalb der Datei gelten und in mehreren Dateien unabhängig voneinander existieren können

datei1.c

```
static int var=4711;
static void func(void) {
   printf("datei1.c var=%d\n",var);
}
```

datei2.c

```
static int var=0715;
static void func(void) {
   printf("datei2.c var=%o\n",var);
}
```

Static lokale Variable

Diese Variablen behalten über der Laufzeit der Funktion ihre Gültigkeit, d.h. sie verlieren nicht ihren Inhalt bei Beendigung der Funktion, sondern behalten den Wert bei. Static lokale

Variablen entsprechen folglich einer globalen Variablen, mit dem Unterschied, dass von außerhalb der Funktion keiner auf diese Variable zugreifen kann. Dementsprechend erfolgt die Initialisierung wie bei globalen Variablen über Speichervorbelegung (Initialisierungswert muss eine Konstanten sein) und nicht zur Laufzeit bei jeden 'Aufruf'. Nichtinitialisierte Variablen werden entsprechend mit 0 initialisiert:

Static Lokale Variablen	entsprechen weitestgehend

`void func(void) {`
`  int var=7;  //Initialisierung erfolgt mit`
`  //jedem Funktionsaufruf`
`  var++;  //7->8  7->8  7->8`
`  static int summe=8;  //Initialisierung`
`  //erfolgt vor Aufruf`
`  //der main-Funktion`
`  summe++;  //8->9  9->10  10->11`
`}`
	`//Namensverschleierung der static`
	`//Variablen, so dass kein anderer`
	`//auf diese Variablen 'zugreifen'`
	`//kann.`
	`int $a$1_xyz=0;`
	`void func(void) {`
	`  int var=7;`
	`  var++;`
	`  $a$1_xyz++;`
	`}`

Hinweis:

- Wird eine Funktion mit statisch lokalen Variablen von nebenläufigen Prozessen aufgerufen, so besteht ein Dateninkonsistenzproblem. Solche Funktionen sind folglich nicht Nebenläufigkeitsfest. In der Spezifikation sind diese als MT-UNSAFE (MultiThread-Unsafe) gekennzeichnet. strtok() beispielsweise speichert die aktuelle Position in einer statische Variablen, so das diese Funktion nicht gleichzeitig von nebenläufigen Prozessen aufgerufen werden darf.

6.3 Typeof Operator

Mittels des Typeof-Operator kann der Datentyp einer Variablen ermittelt werden.

```
Syntax: typeof (T) //GNU-C und C23
        __typeof__(T) //GNU-C
```

T ist wahlweise eine Variable, ein Datentyp oder eine Expression. typeof entspricht einem Konstantenausdruck und wird vom Compiler durch den ermittelten Datentyp ersetzt. Der Rückgabewert ist der ermittelte Datentyp, der nicht in einer Variablen gespeichert oder mit einem anderen Datentyp verglichen werden kann. Typeof kann einzig dazu genutzt werden, weitere Variablen von T anzulegen (entspricht somit dem auto Datentyp bei anderen Programmiersprachen):

```
typedef unsigned int UI;
UI var1[19];
typeof( UI ) var2;
typeof( int) var3;
typeof(var1) var4;
  var4[3]=1;
typeof(var1[2]) var5;
  var5=7;

struct {                  //Unnamed Struct
  int x,y,z;
}xyz;
typeof(xyz) abc;          //Definition einer neuen Variable von
```

```
                          //einem unnamed struct.

if(typeof(var1) == int)   //KO, Ersetzung findet zur Compilezeit
  printf("True");         //statt und der Vergleichsoperator kann
                          //nicht auf Datentypen angewendet werden
//Typeof einer Expression
char var6;
typeof(var6+1) var7;      //Es wird mit nichts kleineren gerechnet
                          //als int. Resultierender Datentyp ist int!
printf("sizeof(var6)=%zd sizeof(var7)=%zd\n",sizeof(var6),sizeof(var7));
```

Anwendung:

- Zur Darstellung von generischen Function-Like Makros (siehe Präprozessor:Function-Like Makros[1]), so dass auf Basis von Typeof lokale Variablen vom Datentyp der 'übergebenen' Variablen erzeugt werden können:

```
#define SWAP1(a,b)   ({typeof(*a) dummy=*a; *a=*b; *b=dummy;})
#define SWAP2(a,b)   ({auto       dummy=*a; *a=*b; *b=dummy;})  //C++
int a=7;
int b=8;
SWAP1(&a,&b);
```

Hinweis:

- typeof war bis C23 eine GCC-Compilererweiterung. Mit C23 wurde diese Erweiterung in den Standard übernommen

6.4 Register Storage Class Specifier

Hinweis an den Compiler, dass diese lokale Variable 'häufig' genutzt wird und folglich der Compiler diese so optimieren soll, dass ein schneller Zugriff auf diese möglich ist.

`Syntax: register Datentyp variablenname;`

Der Storage Class Specifier kann nur auf lokale Variablen und auf Funktionsparameter angewendet werden. Hiermit wird der Compiler gebeten, die Variablen, sofern möglich, in einem Prozessorregister zu halten:

C-Programm

```
         int var1=1;
register int var2=2;
         int var3=3;
if(var1==var2)
  printf("var1==var2");
```

x86 32-Bit Assemblerprogramm

```
  ...
  mov     DWORD PTR [ebp-12], 1 //var1
  mov     ebx, 2                //var2
  mov     DWORD PTR [ebp-16], 3 //var3
  cmp     DWORD PTR [ebp-12], ebx
  jne     .L2        //if(var1==var2)
  ...
.L2:
```

1 Kapitel 9.3 auf Seite 212

Hinweise:

- Moderne Compiler versuchen Variablen temporär im 'gecachten' Speicher zu legen oder teilen dem OS mit, dass die zugehörige Seite nicht ausgelagert werden darf. Folglich ist die Verwendung des Storage Class Specifier `register` nicht mehr nötig, bzw. wird sogar vom Compiler ignoriert
- Ein Register hat keine Speicheradresse, so dass mit dem Adress-Operator keine Adresse ermittelt werden kann
- Ein Debugger stellt beim Darstellen von Variablen den Speicherinhalt der Variablen dar. Aufgrund dessen, dass Compiler Variablen im gecachten Speicher temporär verwalten, kann der angezeigte Variableninhalt vom tatsächlichen Variableninhalt abweichen

6.5 Volatile Type Qualifier

Volatile 'informiert' den Compiler, dass die zugehörige Variable durch andere (nebenläufige Threads) manipuliert werden kann. Der Compiler schließt folglich diese Variable aus allen Optimierungsprozessen aus, so dass die Variable nicht über einen Sequence Point in Register zwischengespeichert werden darf. Stattdessen wird innerhalb eines Sequence Points bei einem Lesezugriff solch einer Variablen diese aus dem Speicher geladen resp. das Ergebnis zum Ende des Sequence Points zurückgeschrieben.

`Syntax: volatile Datentyp variablenname;`

Beispiel:

- Nutzung einer 'normalen' cachebaren Variablen zum Datenaustausch zwischen Threads

Consumer-Thread

```
int thread_run=0; //Cachebare Variable
void consumer(void) {
  //Warten, bis Producer
  //Daten erzeugt hat!
  while(thread_run==0);
  //Consume Data
}
```

Producer-Thread

```
extern int thread_run;
void producer(void) {
  //Produce Data
  //Signalisiere Consumer,
  //dass Daten bereitstehen
  thread_run=1;
}
```

x86 32-Bit Assemblerprogramm

```
consumer:
  mov     eax, DWORD PTR thread_run
.L2:
  test    eax, eax
  je      .L2
  ret
```

x86 32-Bit Assemblerprogramm

```
producer:
  mov     DWORD PTR thread_run, 1
  ret
```

Durch Compileroptimierung wird die Variable thread_run nur einmal aus dem Speicher gelesen und im jedem while-Zyklus der gecachte Inhalt (Inhalt der Variablen eax) genutzt. Änderungen in thread_run werden folglich nicht erkannt!

- Nutzung einer volatile Variablen zum Datenaustausch zwischen Threads

Consumer-Thread

```
volatile int thread_run=0;
//None cacheable variable
void consumer(void) {
  //Warten, bis Producer
  //Daten erzeugt hat!
  while(thread_run==0);
  //Consume Data
}
```

Producer-Thread

```
extern volatile int thread_run;
void producer(void) {
  //Produce Data
  //Signalisiere Consumer,
  //dass Daten bereitstehen
  thread_run=1;
}
```

x86 32-Bit Assemblerprogramm

```
consumer:
.L2:
  mov     eax, DWORD PTR thread_run
  test    eax, eax
  je      .L2
  ret
```

x86 32-Bit Assemblerprogramm

```
producer:
  mov     DWORD PTR thread_run, 1
  ret
```

Volatile bedingt, dass der Inhalt der Variable thread_run bei jedem Schleifendurchlauf aus dem Speicher gelesen wird.

Anwendung:

- wenn Variablen für den Datenaustausch zwischen Threads, zwischen Hauptprogramm und Signalhandler und zwischen dem Hauptprogramm und ISR genutzt werden. (Sorgt jedoch nicht dafür, dass die Daten konsistent sind! Diese müssen z.B. gesondert über Semaphoren oder _Atomic gesichert werden)
- bei Hardwarezugriffen, da
  - ein Lesezugriff immer von der Peripherie und nicht aus einem Cache bedient wird:

```
ad1=*(uint32_t *)(0xFFFFF000);
ad2=*(uint32_t *)(0xFFFFF000);  //Hier erfolgt kein separater Lesezugriff
                                //Stattdessen optimiert der Compiler
                                //den zweiten Lesezugriff durch den
                                //zuvor gelesen Wert weg!
```

  - Schreibzugriffe immer geschrieben werden und nicht wegoptimiert werden:

```
*(uint32_t *)(0xFFFFF004)=1;  //Compiler optimiert diesen Schreibzugriff weg
*(uint32_t *)(0xFFFFF004)=80; //und führt nur diesen aus!
```

Hardware/Peripherie kann wie ein nebenläufiger Thread angesehen werden!

- zum 'Ausbremsen' einer For-Schleife, also zum aktiven Begrenzen der Ausführungsgeschwindigkeit eines Programmes:

```
for(volatile int delay=0;delay<100000;delay++)
   ...
```

6.6 Auto Storage Class Specifier

In C ist der Storage Class Specifier auto ein Relikt aus der Vorgängersprache B, welche a) noch keine unterschiedlichen Datentypen kannte und b) lokale Variablen als solche explizit definiert werden mussten. Mit dem Schlüsselwort auto wurde dort ausgedrückt, dass es sich um eine lokale Variable (vom einzig vorhandenen Datentyp) handelt, resp. mit dem Schlüsselwort extern, dass es sich um eine globale Variable handelt.

In C (bis C17) wird mit auto (automatic storage Duration) dementsprechend ausgesagt, dass es sich um eine lokale Variable handelt. Wenn ergänzend kein Datentyp angegeben wird, wird der Datentyp der Variable implizit auf int gesetzt!

```
auto var;             //KO, Lokale Variable kann hier nicht angelegt werden
void foo(void){
  auto short a=1;     //Lokale Variable, Datentyp short
  auto b=1.0;         //Lokale Variable, Datentyp int (implicit int)
  printf("%zd %zd\n",sizeof(a),sizeof(b));
}
```

Variablendefinitionen innerhalb von Funktion werden auch ohne dem Storage Class Specifier Auto auf automatic storage Duration gesetzt, so dass dieses Schlüsselwort eigentlich entfallen könnte. Da jedoch einige Librarys (z.B. OpenSSL) diesen Storage Class Specifier nutzen (siehe Type Inference for object definitions[2]), ist 'auto' weiterhin Bestandteil der Spezifikation.

Ab C23 und in C++ wird mit dem Schlüsselwort auto ausgedrückt, dass der Datentyp der Variablen sich aus dem Initialisierungswert ergibt:

```
void foo(void) {
  auto a=1;      //Datentyp int
  auto b=1.0;    //Datentyp double
  printf("%zd %zd\n",sizeof(a),sizeof(b));
}
```

Im Gegensatz zu C++ hat das Schlüsselwort auto ab C23 diverse Einschränkungen (siehe auto in C23[3])

- je Variabledefintion kann nur eine Variable definiert werden

```
auto a=1.0,b=2LL;   //KO
```

- es kann kein Zeiger hierüber definiert werden

2 https://thephd.dev/c23-is-coming-here-is-what-is-on-the-menu#n3006--n3007---type-inference-for-object-definitions

3 https://medium.com/@pauljlucas/auto-in-c23-3ce07726e61a

```
short b;
auto *a=&b;
```

- auto kann nicht als Rückgabeparameter oder Übergabeparameter von Funktionen genutzt werden:

```
auto foo(auto par1, auto par2) {}
```

6.7 Const Type Qualifier

Mit dem Type Qualifier const wird eine Variable auf nur lesbaren Zugriff gesetzt.

```
Syntax: Datentyp const Variablenname;
        const Datentyp Variablenname;
```

Die Wertzuweisung solcher Variablen erfolgt über die Initialisierung. Der Schreibschutz erfolgt im Wesentlichen zur Compilezeit, welcher den Compilevorgang abbricht, wenn eine Zuweisung an solche eine Variable erfolgt (eine Const Variable kann kein lvalue sein):

```
      int var0;    //ReadWrite
const int var1=7;  //ReadOnly
int const var2;    //wenn var2 ein globale Variable, dann 0,
                   //andernfalls Zufallswert
var1++;            //KO
var2=var0;         //KO Datentyp (const int)=(int)
var0=var1;         //OK Datentyp (int)=(const int)
```

Zur Laufzeit erfolgt keine Kontrolle. Wird bspw. die Adresse solch einer Variablen in einem Zeigervariablen gespeichert und diese im Anschluss dereferenziert, so hängt das Laufzeitverhalten von diversen Faktoren ab:

```
const int var=7;
int *ptr=&var;
*ptr=4711;      //Schreibender Zugriff auf var.
                //bei lokaler Variable führt dies zur Änderung der Konstanten
                //bei globaler Variable führt dies zum Programmabsturz
```

Alle C-Compiler weisen in der Regel globalen und static lokalen Const-Variablen einer separaten Speicherklasse zu (rodata-Segment). In der Regel wird diese Speicherklasse bei:

- Embedded Systemen in der Tat nur im Speicher mit reinen Lesezugriff gehalten (ROM, Flash). Ein indirekter Schreibzugriff auf solche Variable bewirkt keine Änderung. Das Programm wird normal fortgeführt
- Systemen mit MMU (Windows, Linux, macOS) in einen Speicherbereich gehalten, der durch die MMU schreibgeschützt wird. Ein indirektes Schreiben auf diesen Bereich bewirkt somit ein Laufzeitfehler (Segmentation fault), verursacht durch die MMU

Const lokale Variablen werden wie normale lokale Variablen auf dem Stack gehalten, so dass hier kein zusätzlicher Schutz besteht. Ein indirektes Schreiben solch einer Variablen führt zur Änderung der Konstanten.

Anwendung

- Variablen, die nicht geändert werden sollen
- Variablenübergabe an Funktionen, zur Kennzeichnung, dass diese ReadOnly sind

```
void foo(const int par1) {
  par1=4711;  //KO
```

Hinweise:

- Const Variable sollten initialisiert werden, da später kein schreibender Zugriff möglich ist. Der Compiler gibt leider bei nicht initialisierten Konstanten keinen Fehler aus!
- Const als Rückgabewert einer Funktion ist möglich, aber wenig sinnvoll. Im Gegensatz zu C++, bei welcher mit constexptr gesagt wird, dass die ganze Funktion Konstant ist (und dadurch zur Compilezeit ausgerechnet werden kann) wird bei C einzig gesagt, dass der Rückgabewert konstant ist:

const als Funktionsrückgabewert bei C	**constexpr in C++**

`const int func(int var) {`
`  return var*3;`
`}`
 	
`int main(int argc, char *argv[]) {`
`  int par=7;`
`  //Rückgabewert ist Konstant`
`  const int var = func(par+1);`
`  return 0;`
`}`	

```
constexpr int func(int var) {
  return var*3;
}
int main(int argc, char *argv[]) {
  int par=7;
  //Normaler Funktionsaufruf
  int var = func(par+1);

  //Aufgrund des konstanten Übergabeparameters
  //ist die ganze Funktion konstant und wird
  //zur Compilezeit ausgerechnet
  int var2 = func(4711);
  //Compiler ersetzt Funktionsaufruf durch
  int var=3*4711;
  return 0;
}
```

- CONST Variablen erhöhen die Sicherheit eines Programmes. So sind bspw. in der Programmiersprache RUST alle Variablen const, resp. die Variable darf nur vom Owner geändert werden. Alle Borower haben nur lesenenden Zugriff
- Alternativ zu const Variablen können in C/C++ konstante Werte über Makros dargestellt werden, welche einen kompakteren/schnelleren Code ermöglichen:

```
const int var=7;
#define MAKRO 7
if(var==7) ...
if(MAKRO==7) ...
```

6.8 _Atomic Type Qualifier

Threads sind nebenläufige Ausführungsstränge innerhalb eines Prozesses, welche sich alle globalen Variablen teilen. Die Zuteilung der Rechenzeit der einzelnen Threads erfolgt durch den Scheduler. Ein laufender Thread kann zu jedem Zeitpunkt die Rechenzeit entzogen/zugeteilt werden. Zu jedem Zeitpunkt bedeutet, dass der Entzug auf Maschinenspracheebene und nicht auf C-Befehlsbene stattfindet.

Ein Datenaustausch zwischen den Threads erfolgt typischerweise über globale Variablen. Wird ein Thread während des Zugriffs auf eine globale Variable angehalten, so besteht dies Gefahr von Dateninkonsistenz:

thread1

```
long long var=0x00000000FFFFFFFF;
void *thread1(void *par) {
  while(1) {
    //Do something
    //var++;
  }
}
```

thread2

```
extern long long var;
void *thread2(void *par) {
  long long copy;
  while(1) {
    copy=var;
    //Do something
  }
}
```

x86 32-Bit Assemblerprogramm

```
mov     eax, DWORD PTR var
mov     edx, DWORD PTR var+4
add     eax, 1
adc     edx, 0
mov     DWORD PTR var, eax
//Threadwechsel hier
mov     DWORD PTR var+4, edx
```

x86 32-Bit Assemblerprogramm

```
mov     eax, DWORD PTR var
mov     edx, DWORD PTR var+4
mov     DWORD PTR [ebp-8], eax
mov     DWORD PTR [ebp-4], edx
```

Findet ein Thread-Wechsel zwischen den beiden Schreibbefehlen `mov DWORD PTR var...)` statt, so würde die Variable var zum einen aus dem neuen niederwertigen 32-Bits (hier 0xFFFFFFFF+1=0x00000000) und dem alten höherwertigen 32-Bits (hier 0x00000000) bestehen.

Zur Lösung des Problems stehen diverse Möglichkeiten zur Verfügung:

- Scheduling auf Hochsprachenebene (in C nicht gegeben)
- Absicherung des Zugriffs auf diese Variable über Semaphoren (händisch)
- Absicherung durch Sperrung des Interrupts (nicht sinnvoll)
- Absicherung über atomare (nichttrennbare/-unterbrechbare) Maschinenbefehle

Über den Type Specifier _Atomic wird seit C11 der Compiler angewiesen, Schreibzugriffe auf solche Variablen über atomare Maschinenbefehle abzusichern.

```
Syntax: _Atomic(type-name) Variablenname
        _Atomic type-name Variablenname
```

Zugriffe auf solche Variablen werden als nicht unterbrechbare Einheit dargestellt, so dass sichergestellt ist, dass bei einem Zugriff immer der gesamte Wert gelesen/geschrieben wird:

```
_Atomic long long var=0x00000000FFFFFFFF;
void *thread1(void *par) {
  while(1) {
    //Do Something
    var++; //Befehls wird als atomare Befehl ausgeführt
  }
}
```

Der Type Specifier ist optional und muss vom Compiler nicht unterstützt werden. Auch werden nicht alle Datentypen unterstützt. In der Header-Datei ”stdatomic.h” sind diverse Makros als Alternative/Ergänzung enthalten.

6.9 _Thread_local Storage Class Specifier

Ergänzend zu lokalen (automatic Storage Duration) und globalen (static Storage Duration) Variablen wird mit Thread Local Storage eine für jeden erzeugten Thread eigenständige Variable erzeugt (Thread Storage Duration).

```
Syntax: __thread Datentyp Variablenname;      //GCC
        _Thread_local Datentyp Variablenname; //ab C11
```

Die Speicherplatzreservierung und Initialisierung solcher Variablen erfolgt mit dem Start des Threads.

Beispiel:

```
//Compilerschalter: -lpthread

__thread int tls=2;    //Thread Local Storage
//Alternative Weg für Thread Local Storage
struct data {
  pthread_t id;
  int       data;
} data[2];
static void *run(void *arg)
{
  struct data *this=arg;
  printf("data->data=%d",this->data++);
  printf("->%d\n",this->data);

  printf("tls=%d",tls++);
  printf("->%d\n",tls);
  ...
}
```

```
int main(int argc, char *argv[])
{
  for(int lauf=0;lauf<sizeof(data)/sizeof(data[0]);lauf++) {
    pthread_create(&data[lauf].id,NULL,run,(void *)&data[lauf]);
  }
  ...
}
```

Hinweis:

- Ändert die zugrundeliegenden Adressierungsart, wie auf Variablen zugegriffen wird
  - Global: über feste Adressen -> absolute Adressierung
  - Lokale: relativ zum Basepointer -> indizierte Adressierung
  - Thread: relativ zu einem weiteren Pointer, der mit jedem Threadwechsel neu gesetzt wird. Siehe auch `https://gcc.gnu.org/onlinedocs/gcc/Thread-Local.html#Thread-Local`

6.10 Sonstiges

Storage Class Specifier (`static, extern, auto, register, _Thread_local`) definieren die 'Speicherklasse', in der der Speicher für Variablen reserviert werden soll. Infolgedessen sind diese Specifier nur bei der Definition von Variablen anwendbar. Eine Anwendung bei der reinen Datentypbeschreibung und bei Struktur-/Unionelemente ist nicht möglich:

```
static struct xyz { //KO, static kann nicht auf den Datentyp angewendet
                    // werden
  extern int x;     //KO, extern kann nicht auf ein Strukturelement
                    //angewendet werden
  int y;
};
void foo(void) {
  register struct cd{ //OK, register wirkt auf die Variablendefinition
    int c;
    int d;
  } var;
}
```

Datatypespecifier (`const, volatile, _Atomic`) wirken sich auf den Zugriff aus, d.h. sie beschränken den Zugriff oder garantieren bestimmte Eigenschaften beim Zugriff (durch Nutzung anderer Maschinensprachebefehle). In diesem Sinne können Datatypespecifier sowohl auf Variablen, als auch auf Struktur-/Unionelemente angewendet werden:

```
struct xyz {
           int a;
  const    int b;  //Strukturelement ist nur lesbar
  volatile int c;  //Strukturelement wird beim Zugriff
                   //nicht gecached
  _Atomic int d;
  const volatile int e;
};
struct xyz xyz1;
xyz1.b=7;          //KO Strukturelement x ist nur lesbar
xyz1.c=7;          //OK
const struct xyz xyz2={1,2};
xyz2.x=7;         //KO
xyz2.y=7;         //KO
```

```
const struct ab{  //KO Datentyp kann nicht auf const gesetzt werden
  int a;
  int b;
};
volatile struct c{   //OK hier wirkt volatile auf die Variablendefinition
  int c;
  int d;
} hallo;
```

Compound-Literal entsprechen anonymen Variablen. Per Default sind diese Variablen 'normale' Variablen, also nicht im Zugriff beschränkt. Über Type Qualifier kann der Zugriff auf diese Variablen eingeschränkt werden:

```
(const float []){1e0, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6};
struct xyz{int x,y,z;};
(volatile struct xyz){.y=2};
(_Atomic int []) {1,2,3};
```

7 Zeiger

7.1 Grundlagen

Die Nutzung von Zeiger bedeutet, mit Speicheradressen zu hantieren und basierend auf diesen Adressen den dazugehörigen Speicherinhalt zu lesen/schreiben. Für ein besseres Verständnis, wie ein Prozessor funktioniert und was alles im Speicher 'abgebildet' ist, wird auf das Einführungskapitel Einführung/Literatur:Alles ist Speicher[1] verwiesen.

7.1.1 Speicherauszug

Über einen Speicherauszug (engl. Dump) (siehe w:dump[2]) kann man sich den Speicherinhalt eines Adressbereiches ausgeben lassen. Hier ist für aufeinanderfolgende Speicheradressen der Speicherinhalt dargestellt:

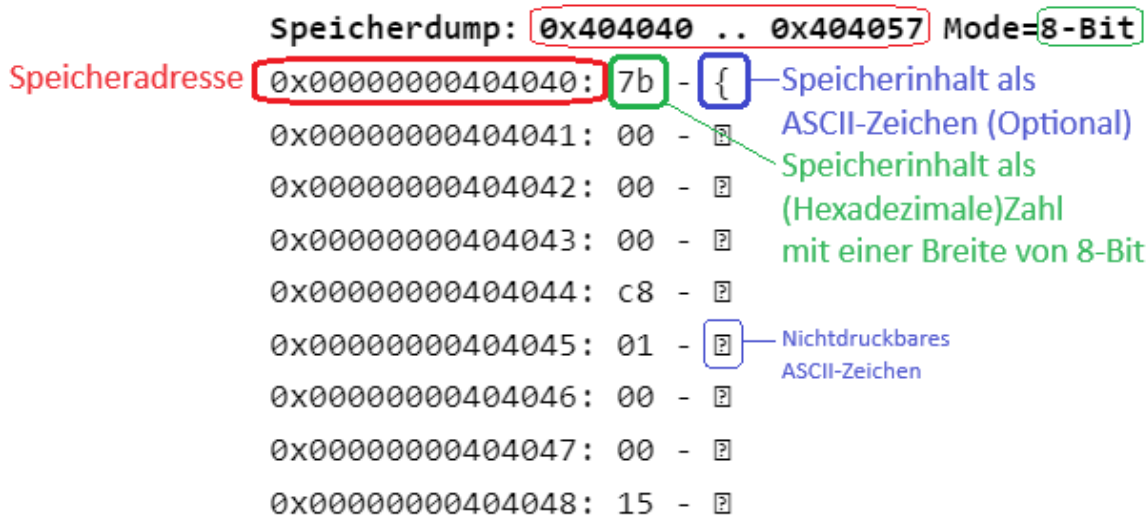

Abb. 7 Beispiel eines Speicherauszuges (Dump)

Der Dump beginnt mit der in hexadezimaler Schreibweise angegebenen Speicheradresse (ggf. gefolgt durch einen Doppelpunkt). Anschließend folgt der Speicherinhalt. Aufgrund der Tatsache, dass der Speicher byteweise organisiert ist, wird der Speicherinhalt als 8-Bit Wert in hexadezimaler Schreibweise ausgegeben. Optional gefolgt von der Interpretation des Speicherinhaltes als ASCII-Zeichen.

1 Kapitel 2.8 auf Seite 20
2 `https://de.wikipedia.org/wiki/dump`

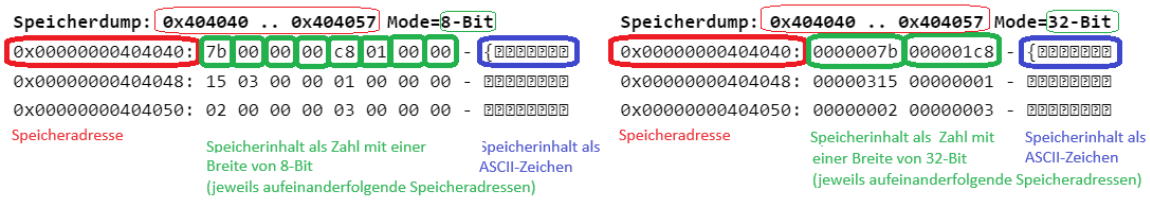

Abb. 8 Beispiel eines Speicherauszuges (Dump)

Zur Platzeinsparung wird oftmals nicht nur der Inhalt einer Speicheradresse, sondern mehrerer Speicheradressen angegeben. Der linke Speicherinhalt entspricht dann dem Inhalt der angegebenen Speicheradresse. Der nächste Speicherinhalt dem Inhalt der Speicheradresse + 1, usw. .

Wenn die Variablen vom Datentyp integer sind, bietet es sich an, anstatt den Speicherinhalt von einer Speicherstelle den Inhalt von 4 Speicherstellen auszulesen und als eine 32-Bit Zahl auszugeben. Die Interpretation der 4 Speicherstellen hängt dabei von der Endianness der Rechnerarchitektur ab (hier Little Endian, bei welcher 7B den niederwertigen Teil der 32-Bit Zahl darstellt).

Hinweis:

- Bei Speicheradressen werden zumeist die führenden Nullen nicht angegeben. Egal mit oder ohne führenden Nullen, die Speicheradresse besteht aus so viel Bitstellen, wie die Rechnerarchitektur adressieren kann
- Bei grafischer Darstellung von Speicherinhalten wird ggf. der Speicher beginnend von der größten Speicheradresse hinabsteigend zur kleinsten Speicheradresse dargestellt

7.1.2 Speicherorganisation von Variablen

Für jede Variable (und für jede Funktion) wird Speicher reserviert. Der Compiler ordnet im Source-Code hintereinander definierten Variablen (des identischen Gültigkeitsbereiches) aufeinanderfolgenden Speicheradressen zu:

```
//Globale Variablen
int arr[3]={123,456,789};       //Alle 3 Arrayelemente liegen im Speicher
                                //direkt hintereinander
int var1=1,var2=2;              //Alle 3 Variablen liegen im Speicher direkt
int var3=3;                     //hintereinander.
int *ptr=&var2;                 //Auch für die Zeigervariable wird Speicher
                                //reserviert.
printf("Adresse von arr[0]=%p\n"
       "            arr[1]=%p\n"
       "            arr[2]=%p\n",&arr[0],&arr[1],&arr[2]);
printf("Adresse von var1  =%p\n"
       "            var2  =%p\n"
       "            var3  =%p\n",&var1,&var2,&var3);
printf("Adresse von ptr   =%p\n",&ptr);
```

Ausgabe:

```
Adresse von arr[0]=0x00000000404040
            arr[1]=0x00000000404044
```

```
            arr[2]=0x00000000404048
Adresse von var1  =0x0000000040404c
            var2  =0x00000000404050
            var3  =0x00000000404054
```

Schaut man sich über den Speicherdump den Inhalt der Speicheradressen der obigen Variablen an, so erkennt man, dass der Speicher der globalen Variablen mit deren Initialisierungswerten vorbelegt ist. Integervariablen belegen 4-Byte Speicher, so dass die Endianness beim Interpretieren des Speicherinhaltes zu berücksichtigen ist (bei Little Endian steht in der kleineren Speicheradresse das niederwertige und in der höchsten Speicheradresse der Variable das hochwertige Byte):

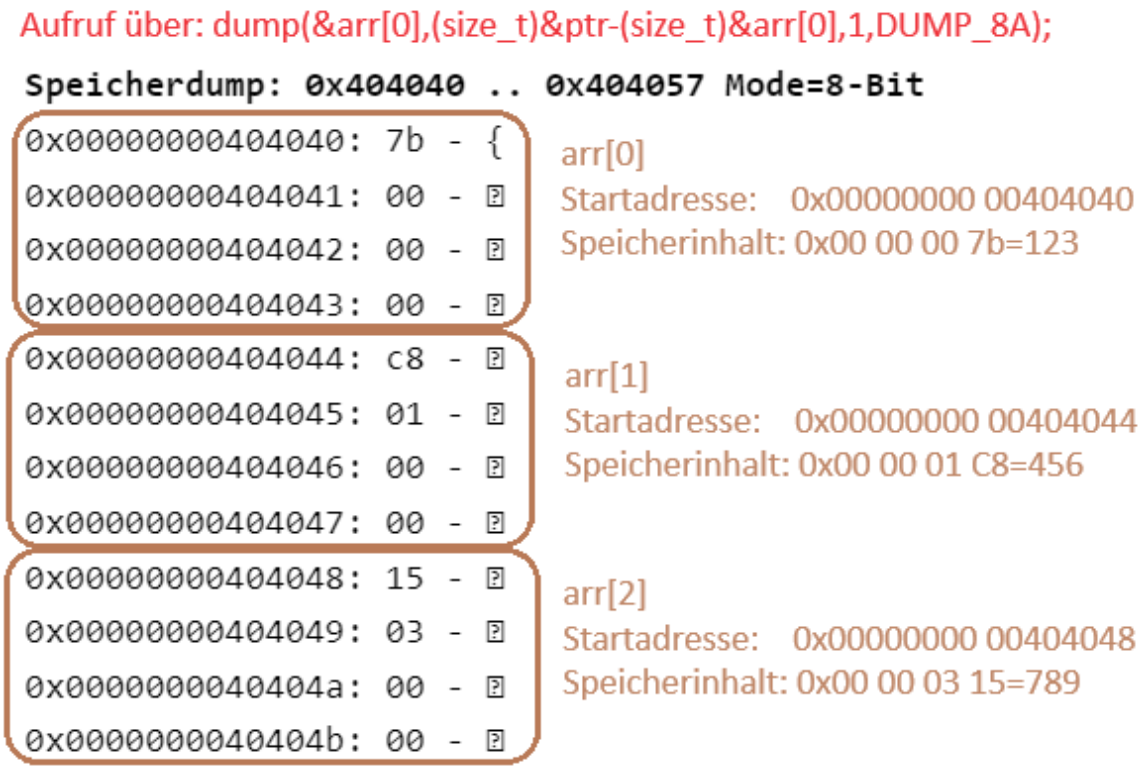

Abb. 9 Erklärung anhand eines Speicherauszuges (je Adresse ein Inhalt), wie Variablen im Speicher angeordnet werden

Werden nicht nur 1 Byte je Speicherstelle angezeigt, so kann der Speicherinhalt mehrerer Variablen übersichtlich dargestellt werden:

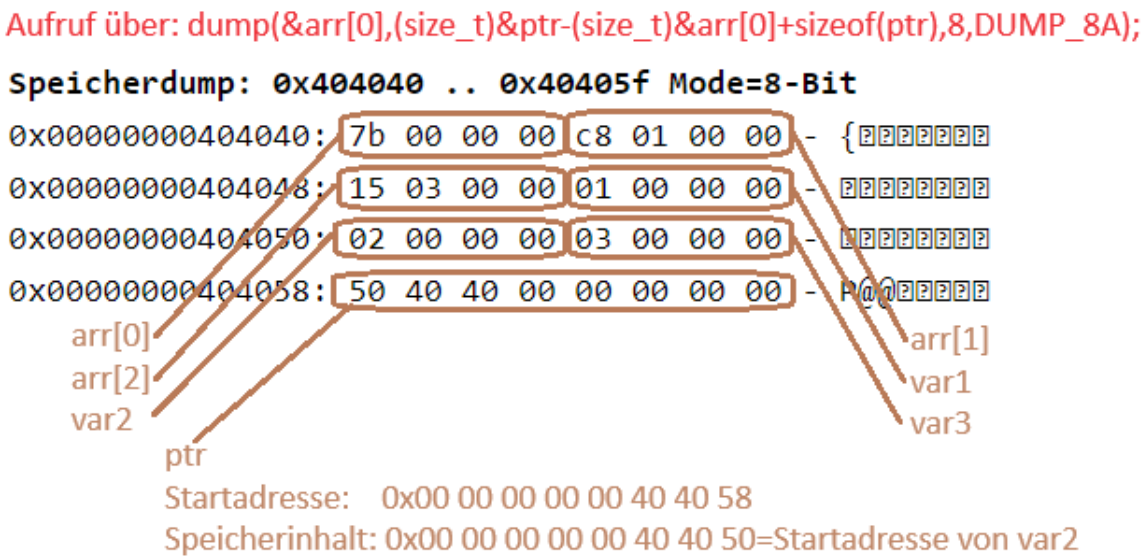

Abb. 10 Erklärung anhand eines Speicherauszuges, wie Variablen im Speicher angeordnet werden

Hinweis:

- Bei Rechnerarchitekturen mit ausgerichteter Speicherausrichtung (Alignment) können zwischen den Variablen noch Füllbytes vorhanden sein

7.1.3 Speicherorganisation unterschiedlicher Gültigkeitsbereiche

Für die einzelnen Gültigkeitsbereiche werden im Speicher (und auch in der Objekt-Datei) unterschiedliche Speicherbereiche (Segmente) vorgehalten, d.h. der Compiler teilt den Speicher in unterschiedliche Bereiche auf und weist den Variablen/Funktionen in Abhängigkeit der Gültigkeit/Sichtbarkeit unterschiedliche Speicherbereiche zu (siehe Grundlagen:Speicherzuweisung[3])! Diese unterschiedlichen Speicherbereiche sind gut an den Startadressen der Variablen erkennbar:

```
//Globale initialisierte Variablen
int var1=1,var2=2;
int var3=3;
//Globale nicht initialisierte Variablen
int bss1,bss2; int bss3;
//Globale Konstanten
const int con1=4,con2=5;
const int con3=6;

//Lokale Variablen
void foo(int par1,int par2) {
  int lok1=7,lok2=8;
  int lok3=9;
}
```

3 Kapitel 4.5.6 auf Seite 41

Die Speicherverwaltung erfolgt unter bspw. Windows, Linux, macOS über die Memory Management Unit (MMU) (siehe w:Memory Management Unit[4]). Diese blendet für jeden Prozess einzelne Speicherbereiche aus dem physikalischen Speicher als virtuellen Speicher ein. Jedem Prozess kann auf dieser Basis der gesamte vom Prozessor adressierbare (virtuelle) Bereich zur Verfügung gestellt werden. Diese Zuordnung zwischen physikalischen- und virtuellen Speicher erfolgt jedoch nicht Byteweise, sondern auf Basis von Pages mit einer Größe von typischerweise 4-KiByte und z.T. 16-KiByte.

Ergänzend zu dieser Zuordnung können für jede virtuelle Page Zugriffsrechte gesetzt werden. 'Normale' Variablen erhalten die Zugriffsrechte 'Read' und 'Write'. Der ausführbare Code erhält die Zugriffsrechte 'Executeable' und ggf. 'Lesend'. Bei Konstanten wird nur das Zugriffsrecht 'Read' gesetzt.

Zur Speichereinsparung werden jedem Prozess nur so viele Pages zugeteilt, wie dieser tatsächlich benötigt. Bei bspw. einem Programm mit 9-KiByte Maschinencode, 5-KiByte Variablen und 1-KiByte Konstanten werden dem Prozess 3 Pages für den Maschinencode, 2 Pages für globale Variablen und eine Page für Konstanten zugeteilt. Mit dem Start des Programm fordert das Programm weitere Pages für den Stack und den Heap an.

Greift nun das Programm auf dem den Prozess nicht zugeteilte Pages zu, so erkennt dies die MMU und veranlasst über das Betriebssystem das Beenden des Prozesses. Dazu wird dem Prozess das Signal 'SIGSEGV:Segmentation violation' (siehe w:Signal (Unix)[5]) gesendet, welches zum Beenden des Prozesses führt. Identisches gilt, wenn die Zugriffsrechte der Pages missachtet werden.

7.2 Adress-Operator

Mit der Definition einer Variablen wird in Abhängigkeit des Datentyps Speicher reserviert. Der Variablenname zeigt auf die erste reservierte Speicherstelle dieser Variable. Die Variablenname als solches entspricht folglich einer Adresse, ab der sich der Inhalt der Variablen befindet. Die Speicheradresse kann mit dem Adress-Operator (`Syntax: &Variablenname/&Funktionsname`) ermittelt werden:

```
int var1=7;
printf("Adresse von var1=0x%zx/%p Inhalt von var1=%d\n",&var1,&var1,var1);

#define WERT 7        //Wert ist keine Variable, so dass hiervon
                      //keine Adresse ermittelt werden kann
printf("Adresse von WERT=----/---- Inhalt von WERT=%d\n",WERT);

struct {int x,y,z;} var2={47,11,12};
printf("Adresse von var2=0x%zx/%p Inhalt von
 var2=%d/%d/%d\n",&var2,&var2,var2.x,var2.y,var2.z);

union  {char a;short b;int c;} var3={47};
printf("Adresse von var3=0x%zx/%p Inhalt von var3=%d\n",&var3,&var3,var3.a);

enum {ROT,GRUEN,BLAU} var4=GRUEN;
printf("Adresse von var4=0x%zx/%p Inhalt von var4=%d\n",&var4,&var4,var4);
```

4 https://de.wikipedia.org/wiki/Memory%20Management%20Unit
5 https://de.wikipedia.org/wiki/Signal%20%28Unix%29

```
printf("Adresse von func=0x%zx/%p Inhalt von
 func=%#x/...\n",&func,&func,*(uint32_t *)&func);
```

Der Wertebereich einer Adresse (und damit die Speichergröße der Speicheradresse) ist unabhängig vom zugrundeliegenden Datentyp, jedoch abhängig von der Adressierungsbereite der Rechnerarchitektur: 2-Byte (bei 16-Bit Rechnerarchitekturen), 4-Byte (bei 32-Bit Rechnerarchitekturen) und 8-Byte (bei 64-Bit Rechnerarchitekturen).

Der Adress-Operator gibt die Startadresse und den Datentype, auf den die Adresse zeigt, zurück (der Datentyp eines Zeigers ist durch ein an den Grunddatentyp angehängtes '*' gekennzeichnet):

```
int   var1;                 //Startadresse 0x404028
short var2;                 //Startadresse 0x40402c
short var3;                 //Startadresse 0x40402e
struct xyz {int x,y;} var4; //Startadresse 0x404030
int arr[2];                 //Startadresse 0x404038
&var1    //entspricht:   (int *)0x404028
&var2    //entspricht:   (short *)0x40402c
&var3    //entspricht:   (short *)0x40402e
&var4    //entspricht:   (struct xyz *)0x404030
&var4.x  //entspricht:   (int *)0x404030
&var4.y  //entspricht:   (int *)0x404034
arr      //entspricht:   (int *)0x404038  //Ausnahme Array!
```

Programmtechnisch wird einzig die Adresse gespeichert. Der Datentyp wird nur vom Compiler beim Dereferenzieren und zur Darstellung der 'Typsicherheit' verwendet:

- Eine Adresse kann nur in einer zu ihrem Datentyp gehörenden Zeigervariablen gespeichert werden
- Bei einer Dereferenzierung wird über den Datentyp bestimmt, wieviel Bytes gelesen/geschrieben werden sollen und wie diese Daten zu interpretieren sind
- Ein Vergleich mit einer anderen Adresse kann nur erfolgen, wenn beide Adressen auf den identischen Datentyp verweisen
- ...

7.3 Zeigervariable/Zeiger

Eine Zeigervariable (Pointer) (`Syntax: Datentype * Zeigervariablenname`) dient dazu, eine Adresse aufzunehmen/zu speichern. Sie entspricht einer normalen Variablen, wobei deren Variableninhalt eine Adresse darstellt. Der Datentype der Zeigervariablen sagt aus, wie der Inhalt des Speichers ab der in der Zeigervariablen enthaltenen Adresse zu interpretieren ist. D.h. bei Zeigervariablen auf einen Char (char *) ist der Inhalt der Speicherstelle (auf den der Zeiger zeigt) als Char zu interpretieren:

```
int var1,var2;
int *ptr1;              //Zeigervariable ptr1 vom Datentype (int *)
int *ptr1a,var1a,*ptr1b,*(ptr1c);  //Integervariable var1a
                        //Zeigervariable ptr1a,ptr1b,ptr1c vom Datentype (int *)
                        //'*' wirkt nur auf Variable rechts vom *

ptr1=&var1;             //Zuweisung eines Pointers mit einer Adresse
int *ptr2=&var2;        //Zuweisung eines Pointers als Initialisierungswert
```

```
int var2a,*ptr2a=&var2a;//OK: var2a ist zum Zeitpunkt der Adressabfrage
                        //    bereits definiert (Single Pass Compiler)
int *ptr2b=&var2b,var2b;//KO: var2b ist zum Zeitpunkt der Adressabfrage
                        //    nocht nicht definiert --> Fehler
int *ptr3a=(int *)100;  //OK: Konstante 100 wird über Explit Cast zu einer
                        //    Adresse gewandelt
int *ptr3b=100;         //KO: Datentypkonflikt: (int *)=(int)
                        //    d.h. einer Zeigervariablen kann nur
                        //    ein Wert eines identischen Datentyps
                        //    zugewiesen werden!
                        //    zugewiesen werden
int *ptr4=&7;           //KO: Eine Konstante hat keine Adresse
                        //    Eine Adresse kann nur von einem LVALUE
                        //    bestimmt werden

char* ptr5,var5;
ptr1=ptr2;              //OK: (int *) = (int *)
                        //    ptr1 und ptr2 sind vom identischen Datentyp
ptr2=ptr5;              //KO: (int *) = (char *)
                        //    ptr2 und ptr5 sind vom unterschiedl. Datentypen

char str[]="Hallo";
char *start1=str;       //OK: (char*) = (char[]) = (char *)
                        //    start1 und str sind vom identischen Datentyp
char *ende=&str[strlen(str)-1]; //OK: (char *) = (char *)
```

Der Datentyp wird wie beim Adress-Operator nur vom Compiler genutzt, jedoch nicht in der Variable gespeichert!

Ein Zeiger ist ähnlich einer Java-Referenz, nur dass:

- eine Zeigervariable explizit mit dem zum Variablennamen vorangestellten `*` definiert werden muss
- Operationen mit den Zeigervariablen erlaubt sind (siehe Zeigerarithmetik[6])
- Bei der Dereferenzierung nicht geprüft wird, ob die Speicheradresse gültig ist (damit einhergehend die große Fehlergefahr beim Umgang mit Zeigern)
- Der Speicher nicht durch die Laufzeitumgebung defragmentiert (verschoben) werden kann, da andernfalls alle Zeigervariablen, die auf diesen Speicherbereich zeigen ebenfalls angepasst werden müssten (Referenzen werden in diesem Fall angepasst)

7.4 Dereferenzierung

Über die Dereferenzierung (`Syntax: * Zeigervariable oder * Adresse`) wird auf den Inhalt der Speicheradresse zugegriffen, auf welche der Zeiger/die Zeigervariable zeigt. Hier wird zunächst der Inhalt der Zeigervariablen gelesen und dieser Inhalt als Startadresse zum Lesen/Schreiben des eigentlichen Inhaltes aus dem Speicher genutzt (= Indirekte Adressierung). Der zugrundeliegende Datentyp der Zeigervariablen gibt an, wie viel Bytes ab dieser Startadresse gelesen werden sollen und wie der Inhalt zu interpretieren ist.

6 Kapitel 7.7 auf Seite 150

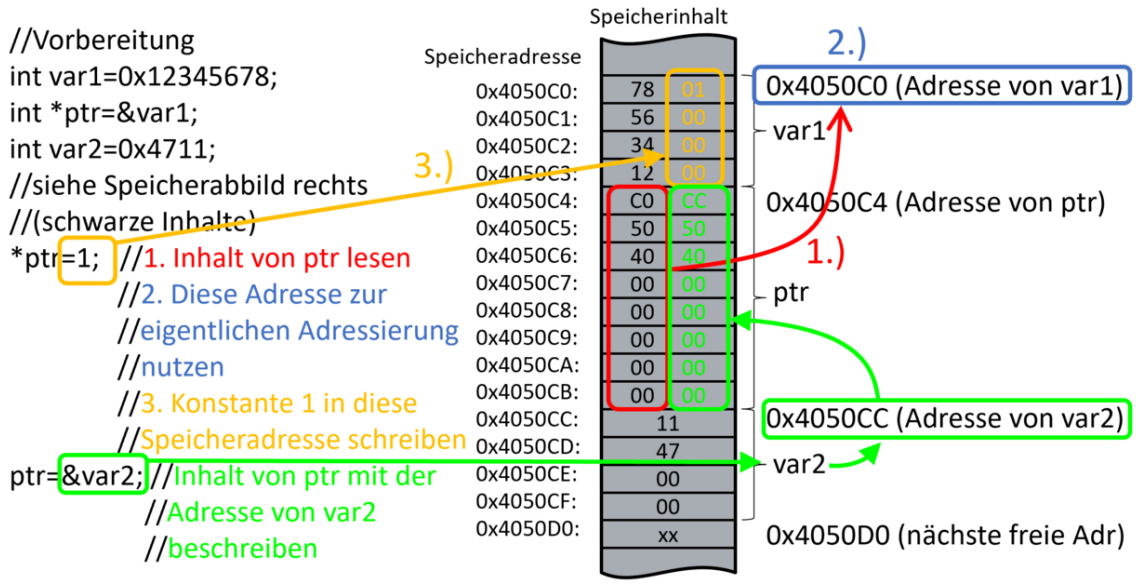

Abb. 11 Erklärung der Funktionsweise der Dereferenzierung

Mit `*ptr=1;` wird der Zuweisungswert 1 in den Speicherbereich geschrieben, auf welcher die Zeigervariable ptr zeigt. Die Zeigervariable ist vom Datentyp `(int *)`, so dass aufgrund des zugrundeliegenden Datentyps `(int)` 4 Bytes geschrieben werden. Die Konstante selbst ist ebenfalls vom Datentyp `(int)`.

Beim umgedrehten Fall `var2=*ptr;` (in der Grafik nicht dargestellt), würde als Zuweisungswert für var2 zunächst der Inhalt des Speicherbereiches gelesen, auf den die Zeigervariable ptr zeigt. Aufgrund des zugrundeliegenden Datentyps `(int)` der Zeigervariablen würden 4 Byte gelesen werden und diese als Integer interpretiert werden.

In `ptr=&var2` wird keine Dereferenzierung verwendet (kein `*` auf eine Zeigervariable oder Adresse angewendet). Hier erfolgt eine normale Zuweisung der Zeigervariablen ptr mit der Adresse der Variablen var2. Die Zeigervariable ist vom Datentyp `(int *)` und der Adress-Operator erzeugt aus dem Integerdatentyp var2 einen Zeiger auf Integer `(int *)`, so dass L-Value und R-Value vom identischen Datentyps sind.

Dereferenzierung aus Programmiersicht:

```
int  var1;            //(mögliche) Startadresse 0x100
int *ptr1=&var1;      //Startadresse 0x104
                      //Initialisierung dieser Variable mit
                      //  der Adresse von var1 (hier 0x100)
var1=7;               //Direktes Schreiben des Wertes 7 in var, d.h.
                      //  Schreiben von 7 in die Adresse ab 0x100
*ptr1=7;              //Indirektes Schreiben des Wertes 7 in var
                      //  d.h. Schreiben des Wertes 7 in die
                      //  Speicheradresse, auf die Variable ptr1
                      //  zeigt
//Bei Strukturen
struct xy{int x,y;};
struct xy var2;       //(mögliche) Startadresse 0x10C
struct xy var3;       //Startadresse 0x114
```

```
struct xy *ptr2;      //Startadresse 0x11C
ptr2=&var2;           //Startadresse der Struktur zuweisen
var3=var2;            //Direktes Lesen/Schreiben (8-Bytes)
var3=*ptr2;           //Indirektes Lesen (8-Bytes)
*ptr2=var3;           //Indirektes Schreiben (8-Bytes)

//Bei Strukturelementen
var2.x=1; var2.y=1;   //Direktes schreiben auf Strukturelemente
(*ptr2).x=2;          //Indirektes Schreiben über Dereferenzierung
(*ptr2).y=2;
ptr2->x=3;            //Indirektes Schreiben über Dereferenzierung
ptr2->y=3;            //  durch Nutzung des '->' Operators

//Sonstiges
int *ptr3=&var2.y;    //Adresse eines Strukturelementes speichern
*ptr3= 4711;          //Inhalt des Strukturelementes ändern
```

Hinweis:

- Der -> Operator, anwendet auf einen Zeiger auf eine Struktur/Union und der Beschreibung eines Struktur-/Unionelementes entspricht der Dereferenzierung des Zeigers an den gegebenen Offset (des Struktur-/Unionelemntes).

7.5 Sichtweise von Zeiger auf Basis des Datentyps

Mittels des Adress-Operators wird aus dem Datentyp ein Zeiger. Aus Sichtweise des Datentyps wird dem Datentyp ein * zugefügt:

```
int var1;      //aus Datentypsicht  (int)
    &var1      //aus Datentypsicht &(int) → (int *)
int *ptr1;     //aus Datentypsicht  (int *)
    &ptr1      //aus Datentypsicht &(int *) → (int **)
int **ptr2;    //aus Datentypsicht  (int **)
     &ptr2     //aus Datentypsicht &(int **) → (int ***)
//Doppelte Anwendung des Adress-Operators
     &&ptr1    //aus Datentypsicht &(&(int *)) → &(Konstanten) → KO
               //da von einer Konstanten keine Adresse gebildet werden kann
```

Mit des Dereferenzierung wird der Inhalt des Datentyps gelesen. Aus Sichtweise des Datentyps wird ein * vom Datentyp weggenommen:

```
int var;           //aus Datentypsicht   (int)
int *ptr1=&var;    //aus Datentypsicht   (int *)
    *ptr1          //aus Datentypsicht  *(int *) → (int)

int **ptr2=&ptr1;  //aus Datentypsicht    (int **)
     *ptr2         //aus Datentypsicht   *(int **) → (int *)
    **ptr2         //aus Datentypsicht  **(int **) → (int)
//Sonstiges
   *var            //aus Datentypsicht   *(int) → KO
    //da Dereferenzierung angewendet auf Ganzzahl oder Gleitkommazahl dem
    //Multiplikatoroperator entspricht!
```

Der resultierende Datentyp ist anschließend die Grundlage für den weiteren Umgang mit diesem Ausdruck.

Genauso, wie sich eine Division durch eine Multiplikation 'aufheben' lässt, hebt auch eine Dereferenzierung den Adress-Operator auf:

```
int var;
*&var=7;      //OK
//aus Datentypsicht *(&(int)) → *((int *)) → (int)

&*var=9;      //KO
//aus Datentypsicht &(*(int))
//Das * wird in diesem Fall als Multiplikationsoperator angesehen
```

Weitere Beispiele:

```
struct xyz {int x,y,z;};
struct xyz  var;   //aus Datentypsicht  (struct xyz)
struct xyz *ptr;   //aus Datentypsicht  (sruct xyz *)

  *ptr             //aus Datentypsicht *(struct xyz *) → (struct xyz)
  &ptr             //aus Datentypsicht &(struct xyz *) → (struct xyz **)
   var.x           //aus Datentypsicht  (int)
  &var.x           //aus Datentypsicht &(int) → (int *)
```

7.6 Zuweisung und explizites Cast

Bei der Zuweisung von Zeigern müssen beide Zeiger vom identischen Datentyp sein. Der Compiler führt keine implizite Typumwandlung durch, wenn die Zeiger auf unterschiedliche Datentypen zeigen (Ausnahme siehe Void-Zeiger[7]). Bei Inkompatibilität gibt der Compiler entweder einen Error oder eine Warning unter Angabe der inkompatiblen Datentypen aus (Warning in diesem Fall bitte als Fehler ansehen).

```
char         varc,*ptrc;
short        vars,*ptrs;
int          vari,*ptri;
unsigned int varu,*ptru;
ptrc=&varc;         //OK (char *)=(char *)
ptrc=&vars;         //KO (char *)=(short *) *1)
ptrc=&vari;         //KO (char *)=(int *)   *1)
ptrc=ptrs;          //KO (char *)=(short *)
ptrc=ptri;          //KO (char *)=(int *)

ptru=&vari;         //KO (unsigned int *)=(  signed int *)
ptri=&varu;         //KO (  signed int *)=(unsigned int *)
```

Über explizite Typkonvertierung (Cast Operator) kann ein Zeiger in einen anderen Zeiger gecastet werden. Es erfolgt dabei keine Konvertierung des Inhaltes, auf den der Zeiger zeigt, sondern nur eine andere Interpretation, wie der Inhalt der Speicheradresse zu interpretieren ist:

```
int    vari=0x01234567;
short  vars=0x89AB;
char   varc1=0xCD,varc2=0xEF;
int   *ptri=&vari;
short *ptrs=&vars;
char  *ptrc=&varc1;
dump(&vari,32,8,DUMP_8A);

ptri=&vari;           //OK (int   *)= (int *)
ptri=(int *)&vari;    //OK (int   *)=((int *))(int *)=(int *)
ptrs=&vari;           //KO (short *)= (int *)
ptrs=(short *)&vari;  //OK (short *)=((short *))(int *)=(short *)
```

7 Kapitel 7.9 auf Seite 158

```
ptrc=(char  *)&vari;  //OK (char  *)=((char  *))(int *)=(char  *)
printf("%x"  ,*ptri); //0x01234567
printf("%hx" ,*ptrs); //0x4567 *1)
printf("%hhx",*ptrc); //0x67   *2)

ptri=&vars;           //KO (int *)= (short *)
ptri=(int *)&vars;    //OK (int *)=((int  *))(short *)
printf("%x",*ptri);   //0xefcd89ab;  *3)

ptri=ptrc;            //KO (int  *)= (char *)
ptri=(int  *)ptrc;    //OK (int  *)=((int *)) (char *)
ptrc=(char *)ptri;    //OK (char *)=((char *))(int  *)

//Ausgabe
//Speicherdump: 0x404040 .. 0x40405f Mode=8-Bit
//0x00000000404040: 67 45 23 01 ab 89 cd ef - gE#
//0x00000000404048: 40 40 40 00 00 00 00 00 - @@@
//0x00000000404050: 44 40 40 00 00 00 00 00 - D@@
//0x00000000404058: 46 40 40 00 00 00 00 00 - F@@
```

Im obigen Beispiel belegt die Integervariable den Speicherbereich 0x404040..0x404043. Aufgrund der Little Endian Kodierung des Prozessors wird das niederwertige Byte zuerst im Speicher abgelegt. Die nächste freie Speicheradresse ist 0x404044, so dass die Shortvariable diese und die nachfolgende Adresse 0x404045 zugewiesen bekommt. Nachfolgende zwei Speicheradresse werden den beiden Charvariablen varc1 und varc2 zugewiesen. ptri, ptrs und ptrc werden im Anschluss abgelegt und belegen jeweils 8 Byte Speicher. Diese sind mit der Startadresse der jeweiligen Variablen initialisiert.

1) Mit `ptrs=(short *)&vari;` wird zunächst die Adresse von vari (0x404040) vom Datentyp `(int *)` geladen. Da dieser inkompatibel zum Zieldatentyp `(short *)` ist, wieder dieser per explizitem Cast in den korrekten Zieldatentyp gewandelt. Die Adresse wird dabei nicht verändert. Beim Dereferenzieren des Pointers mittels `*ptrs` wird nun zunächst die Speicheradresse aus der Variablen ptrs geladen (0x404040). Da der Zeiger vom Datentyp `(short *)` ist, wird aufgrund des zugrundeliegenden Datentyps ab dieser Adresse 2 Bytes gelesen. Der Inhalt 0x67 der kleineren Adresse bildet den niederwertigen Teil des 16-Bit Datenwertes und der Inhalt 0x45 der höheren Adresse den höherwertigen Teil, so dass der ausgelesen Wert 0x4567 ist. Da bei variadischen Funktionen keine Parameter kleiner als integer übergeben werden können, wandelt der C-Compiler diesen short Wert in einen int Wert um (0x00004567). Mit der Formatanweisung `%hx` wird printf gesagt, dass von diesem Integerwert nur die unteren 16-Bit zu nutzen sind.

2) Auch hier wird mit `ptrc=(char *)&vari;` die Adresse von vari geladen und der Integer-Zeiger auf einen Char-Zeiger gecastet. Beim Dereferenzieren mit `*ptrc` wird ab der Startadresse aufgrund des zugrundeliegenden Datentyps `char` nur eine Speicherstelle gelesen, so dass sich 0x67 ergibt. Dieser Wert wird aufgrund der variadischen Funktion in einen Integer gewandelt (0x00000067) und mit `%hhx` nur die unteren 8-Bit des 32-Bit Wertes ausgegeben.

3) Aufgrund `ptri=(int *)&vars` wird die Startadresse der Short-Variablen vars (0x404044) in eine Integer-Zeiger gewandelt. Mit `*ptri` wird ab der Adresse in ptri aufgrund des zugrundeliegenden Datentyps 4 Byte gelesen. Der Wert in der Speicherstelle 0x404044 mit

0xab stellt dabei die niederwertigsten 8-Bit und der Wert an der Speicherstelle 0x404047 mit 0xef die höchstwertigsten 8-Bit dar. Die Dereferenzierung ergibt somit 0xefcd89ab. **Vorhandene 'Variablengrenzen' spielen folglich beim Dereferenzieren keine Rolle mehr!**

Für den Anwendungsfall, dass die Adressvergabe von Variablen/Funktionen nicht durch die Toolchain, sondern durch den Programmierer erfolgt (z.B. für den Peripheriezugriff oder der Interprozess-Kommunikation), können Ganzzahlvariablen/-konstanten zu Zeiger gecastet werden (und umgedreht). In diesem Fall gilt es, die Datenbreite des Zeigers und die Datenbreite des Ganzzahldatentyps zu beachten:

```
char         var1,*ptr1;
int          var2;
long long    var3;
ptr1=(char *)0x100;   //OK, wobei der Compiler die Integer-konstante 0x100
                      //bei einer 64-Bit Rechnerarchitektur zu einer
                      //LongLong-Konstante erweitert.
ptr1=(char *)var1;    //Compilerwarning, da die Charvariable in einem
                      //64-Bit zu 'kurz' ist.
ptr1=(char *)var2;    //Compilerwarning, sie zuvor
ptr1=(char *)var3;    //OK, var3 ist eine 64-Bit Variable, welche
                      //unproblematisch eine Adresse beinhalten kann

float        varf;
ptr1=(char *)varf;    //KO Es können nur Ganzzahlkonstanten konvertiert
                      //   werden
```

Für diesen Sonderfall stellt C/C++ den Datentyp uintptr_t (vorzeichenlos) und intptr_t (vorzeichenbehaftet) bereit, welcher die identische Datenbreite wie ein Pointer hat (Definition in der Header-Datei stdint.h)

```
#include <stdint.h> //Definition von intptr_t  uintptr_t
uintptr_t  var1;    //Ganzahlvariable zur Aufnahme einer Speicheradresse
int        var2;
char      *ptr1;
var1=(uintptr_t)&var2;  //Adresse einer Variablen in einer Ganzzahlvariablen
                        //speichern
ptr1=(char *)var1;      //Ganzzahlvariablen zu einer Adresse konvertieren
```

Bei Nutzung von Pointercasts gilt es besondere Vorsicht walten zu lassen. Geübten Programmierer können hiermit die komplexesten Datenstrukturen schnell und effektiv realisieren. Ungeübte Programmierer richten hiermit eher Schaden an, als dass es ihnen Vorteile bringt.

7.7 Zeigerarithmetik

7.7.1 Zeigeraddition/-subtraktion

Arrays sind dadurch gekennzeichnet, dass nicht nur Speicherplatz für ein Element des zugrundeliegenden Datentyps, sondern Speicherplatz für mehrere Elemente reserviert wird. Alle Elemente liegen dabei im Speicher direkt hintereinander. Die Adressen der einzelnen Elemente haben folglich einen Abstand entsprechend der Größe des zugrundeliegenden Datentyps. Der Variablenname entspricht dabei der Startadresse des ersten Elementes. Die []-Klammer entsprechen der Dereferenzierung, wobei auf die Startadresse ein Offset gerech-

net wird, der sich aus dem Index multipliziert um die Größe zugrundeliegenden Datentyps ergibt:

```
int var[7];      //Es werden 7*4 Byte Speicher reserviert
var              //var ohne [] und ohne & entspricht der Startadresse
                 //des reservierten Speichers
&var[0]          //Startadresse +   0 *sizeof(int)
&var[1]          //Startadresse +   1 *sizeof(int)
&var[2]          //Startadresse +   2 *sizeof(int)
&var[-1]         //Startadresse + (-1)*sizeof(int)
…
```

Die Addition/Subtraktion einer Ganzzahl zu einem Zeiger entspricht (entsprechend der Array Dereferenzierung) der Addition/Subtraktion der Ganzzahl multipliziert mit der Größe des zugrundeliegenden Datentyps des Zeigers zu diesem Zeiger:

```
int   arri[4]={0x80000001,0x40000002,
               0x20000004,0x01000008};
short arrs[8]={0x0011,0x2233,0x4455,0x6677,
              0x8899,0xAABB,0xCCDD,0xeeff};
int   var=-1;
dump(&arri,(uintptr_t)&var-(uintptr_t)&arri+sizeof(var),8,DUMP_8A);
short *ptrs=arrs;

printf("*(ptrs+1)=%#hx",*(ptrs+1)); //*(Startadresse+1*sizeof(short)) *1)
printf("*(ptrs+3)=%#hx",*(ptrs+3)); //*(Startadresse+3*sizeof(short)) *2)
printf("*(3+ptrs)=%#hx",*(3+ptrs)); //*(Startadresse+3*sizeof(short)) *3)

int offset=3;
printf("*(ptrs+offset)=%#hx",*(ptrs+offset));     //0x6677 *4)
printf("*(ptrs+2*offset)=%#hx",*(ptrs+2*offset)); //0xCCDD *4)
printf("*(offset+ptrs)=%#hx",*(offset+ptrs));     //0x6677 *4)
printf("*(2*offset+ptrs)=%#x",*(2*offset+ptrs)); //0xFFFFCCDD *5)

printf("*(ptrs+8)=%#hx",*(ptrs+8)); //0xFFFF *6)
printf("*(ptrs-1)=%#hx",*(ptrs-1)); //0x0100 *6)

printf("*ptrs+1  =%#hx",*ptrs+1);   //0x0012 *7)

//Mit Cast eines Pointers auf einen anderen Zeigertyp
int *ptri =(int *)&arrs[2];
printf("*(ptri+1)=%#x",*(ptri+1));  //0xaabb8899 *8)

//Ohne Umweg über eine Pointervariable
printf("*(int *)&arrs[3]=%#x",*(int *)&arrs[3]); //0x88996677 *9)

printf("*  (short *)(arri +2) =%#hx\n",*  (short *)(arri+2));  //*10)
printf("*(((short *) arri)+2))=%#hx\n",*(((short *) arri)+2)); //*11)
printf("*  (short *) arri +2  =%#hx\n",*  (short *) arri+2);   //*12)

//Mit Strukturen
struct xyz{int x,y,z;} var1[7],*ptr1;
ptr1=var1;
*(ptr1+3)=(struct xyz){1,2,3};   //*13)

//Ausgabe
//Speicherdump: 0x404040 .. 0x404063 Mode=8-Bit
//0x00000000404040: 01 00 00 80 02 00 00 40
//0x00000000404048: 04 00 00 20 08 00 00 01
//0x00000000404050: 11 00 33 22 55 44 77 66
//0x00000000404058: 99 88 bb aa dd cc ff ee
//0x00000000404060: ff ff ff ff 00 00 00 00
```

Im Speicher wird zunächst der Speicher für das Integer-Array mit 4 Elementen a 4-Byte reserviert (Speicherbereich 0x404040..0x40404F) und dieser mit den Initialisierungswerten

vorbelegt. Der Variablenname `arri` entspricht der Startadresse 0x404040. Nachfolgend wird der Speicher für das Short-Array reserviert, welches sich von 0x404050 bis 0x40405f erstreckt. `arrs` zeigt auf die Speicheradresse 0x404050. Die nächsten 4-Byte von 0x404060..0x404063 sind für die variable `var` reserviert (-1 wird über das 2er Komplement von +1 gebildet, so dass sich 0xFFFFFFFF ergibt).

1) Mit `short *ptrs=arrs;` wird eine Short-Zeigervariable definiert und diese mit der Startadresse des Short-Arrays (hier 0x404050) initialisiert. Der Datentyp von `arrs` entspricht ein Zeiger auf den zugrundeliegenden Datentyp des Arrays, so dass der Datentyp der Zeigervariablen und des Arraynamens identisch ist. Mit `*(ptrs+1)` wird zunächst zum Inhalt von ptrs einmal die Größe des zugrundeliegenden Datentyps von `ptrs` (short *) addiert (0x404050+1*sizeof(short)=0x404052). Diese Adresse wird im Anschluss dereferenziert, d.h. aufgrund des Short-Zeigers der Inhalt der Speicherstellen 0x404052..0x404053 gelesen. Der Zugriff über `*(ptrs+1)` entspricht folglich `arrs[1]`.

2) Wie zuvor, nur dass hier nicht einmal sondern dreimal die Größe des zugrundeliegenden Datentyps auf die Startadresse addiert wird. Der ausgelesene Speicherbereich ist 0x404056..0x404057 und der Inhalt entsprechend 0x6677.

3) Bei der Zeigerarithmetik ist es unbedeutend, ob zum Pointer eine Ganzzahl oder zu einer Ganzzahl ein Pointer addiert wird.

4) Der Offset zu einem Pointer kann sich wahlweise aus einer Konstanten, einer Ganzzahlvariablen oder einem Ausdruck ergeben, der eine Ganzzahl zurückliefert.

5) Bei der Dereferenzierung eines Short-Zeigers werden 2-Byte gelesen. Diese 2-Byte werden aufgrund der variadischen Funktion in einen Integerwert gewandelt. Das Vorzeichenbit/das höchstwertige Bit der zu konvertierenden Zahl (0xCCDD=0b1100110011011101) ist '1', so dass aufgrund des vorzeichenbehafteten Datentyps short eine vorzeichenrichtige Erweiterung erfolgt. Die resultierende Zahl ist folglich 0xffffccdd, welche aufgrund des Formatstrings %x als 32-Bit Zahl ausgegeben wird.

6) Bei der Pointerarithmetik wird nicht darauf geachtet, ob die resultierende Speicheradresse auf eine gültige Adresse zeigt. Vielmehr wird die Arithmetik so ausgeführt, wie diese codiert ist. Im ersten Fall wird zur Startadresse 8*sizeof(short) addiert, so dass der Adressbereich 0x404060..0x404061 mit den Inhalt 0xFFFF dereferenziert wird. Im zweiten Fall ergibt sich als Adressbereich 0x40404E..40404F mit dem Inhalt 0x0100. Erster Speicherbereich gehört zur Variablen `vari` und zweiterer zum Array `arri`.

7) Die Dereferenzierung hat nach der Prioritätenliste der Operatoren eine höhere Priorität als die Addition. Folglich wird hier erst der Inhalt des Zeigers dereferenziert (0x0011) und zu diesem Wert 1 addiert.

8) `ptri` wird mit `int *ptri =(int *)&arrs[2];` auf die Startadresse des 2 Array-Elementes gesetzt (0x404054) und diese Adresse zu einem Integer-Zeiger gecastet. Wird nun mit `*(ptri+1)` dieser Integer-Zeiger dereferenziert, so wird entsprechend des zugrundeliegenden Datentyps 1*sizeof(int) auf den Inhalt des Pointers addiert. Daraus ergibt sich der adressierte Speicherbereich zu 0x404058..0x40405B und dem Inhalt 0xaabb8899.

9) Hier wird mit `*(int *)&arrs[3]` zunächst die Adresse des dritten Array-Elementes (0x404056) geholt und diesen Short-Zeiger in einen Integer-Zeiger gecastet. Dieser Integer-Zeiger wird im Anschluss dereferenziert. Adressiert wird der Speicherbereich 0x404056..0x404059 mit dem Inhalt 0x88996677.

10) `*(short *)(arri+2)` Der Arrayname entspricht einem Zeiger auf den zugrundeliegenden Datentyp, so dass `arri` ein Integer-Zeiger entspricht. Aufgrund der Klammerung wird zu dieser Adresse zweimal der Offset des zugrundeliegenden Datentyps addiert, so dass die Klammerung einen Int-Zeiger mit der Startadresse 0x404048 ergibt. Diese Startadresse wird nun zu einem Short-Zeiger gecastet, welcher im Anschluss dereferenziert wird. Adressiert wird der Speicherbereich 0x404048..0x404049 mit dem Inhalt 0x0004.

11) `*(((short *) arri)+2)` Der Arrayname entspricht einem Zeiger auf den zugrundeliegenden Datentyp, so dass `arri` ein Integer-Zeiger entspricht. Dieser Zeiger wird aufgrund der Klammerung in einen Short-Zeiger gewandelt. Die Pointerarithmetik wird nun auf diesen Short-Zeiger angewendet, so dass auf die Startadresse '2*sizeof(short)' addiert wird. Im Anschluss wird dieser Short-Zeiger mit der Adresse 0x404044..0x404045 mit dem Inhalt 0x0002 dereferenziert.

12) `*(short *) arri+2` Hier hat die Typumwandlung eine höhere Priorität als die Addition, so das zunächst der Integer-Zeiger in einen Short-Zeiger gewandelt wird. Auch die Dereferenzierung hat einen höhere Priorität als die Addition, so dass der Short-Zeiger mit der Adresse 0x404040..0x404041 und dem Inhalt 0x0001 dereferenziert wird. Zu diesem Ganzzahlwert wird nun die 2 addiert.

13) In diesem Anwendungsfall `*(ptr1+3)` ist `ptr1` ein Zeiger auf `(struct xyz *)`. Eine Addition von 3 bewirkt hier, dass zur Startadresse 3*sizeof(struct xyz) addiert wird.

Addition/Subtraktion eines Pointers mit einer Ganzzahl `(Syntax: Adresse/Zeiger ± Ganzzahl resp. Ganzzahl ± Adresse/ Zeiger)` bedeutet folglich Vielfaches des zugrundeliegenden Datentyps zu/von der Startadresse aufaddieren/abziehen.

Eine Addition zweier Pointer `(Syntax: Adresse/ Zeiger + Adresse/ Zeiger)` ist nicht möglich und führt zu einem Compilerfehler.

Eine Subtraktion zweier Pointer `(Syntax: Adresse/ Zeiger - Adresse/ Zeiger)` ist möglich (siehe Pointerdifferenz[8]).

8 Kapitel 7.6 auf Seite 148

Weitere Operatoren wie Multiplikation/Division/logische Operationen sind mit Zeigern nicht möglich, unabhängig davon, ob der zweite Operator eine Ganzzahl oder ein Pointer ist.

Zeigerarithmetik kann zu einer ungültigen Speicheradresse führen. Die Speicherung der ungültigen Adresse in einer Zeigervariablen ist dabei nicht das Problem, sondern die Dereferenzierung. Der Fehler macht sich unterschiedlich bemerkbar:

- **Programmabsturz, da auf eine dem Prozess nicht zugewiesen Speicherstelle zugegriffen wurde -> Segmentation Fault.**
- **Adresse zeigt auf eine gültige Speicheradresse, welche eine andere 'Bedeutung' hat, bspw. einer anderen Variablen zugehörig ist. Bei Dereferenzierung dieser Adresse würde sich der Inhalt einer anderen Variablen ändern/gelesen werden, welches erst bei der späteren Nutzung dieser Variablen auffällt. Das Programm verhält sich in diesem Falle zumeist 'merkwürdig'.**

Solche Fehler sind nur schwer zu finden, da die Codezeilen, bei welchen der Fehler erkannt wird und die Codezeilen, welche den eigentlichen Fehler erzeugen, weit voneinander entfernt sind.

7.7.2 Pre/Post Inkrement/Dekrement

Der Pre/Post Inkrement/Dekrement Operator bedeutet, den Inhalt einer Variablen um 1 zu erhöhen oder zu erniedrigen. Angewendet auf Zeiger bedeutet der Operator, die Adresse nicht um 1 sondern um die Größe des zugrundlegenden Datentyps zu erhöhen/erniedrigen:

```
int *ptr1=(int *)0x100;
int *ptr2;
ptr2=  ptr1++;  //ptr2    =ptr1_old             ptr1_new=ptr1_old+sizeof(int)
                //ptr2    =0x100                ptr1_new=0x104
ptr2=++ptr1;    //ptr1_new=ptr1_old+sizeof(int) ptr2    =ptr1_new
                //ptr1_new=0x108                ptr2    =0x108
ptr2=  ptr1--;  //ptr2    =ptr1_old             ptr1_new=ptr1_old-sizeof(int)
                //ptr2    =0x108                ptr1_new=0x104
ptr2=--ptr1;    //ptr1_new=ptr1_old-sizeof(int) ptr2    =ptr1_new
                //ptr1_new=0x100                ptr2    =0x100
```

Mit dem Inkrement/Dekrement kann ergänzend der Pointer Dereferenziert werden. Bei Anwendung von Post Inkrement/Dekrement `*ptr++` oder `*ptr--` bedeutet dies, dass die Dereferenzierung auf Basis der aktuellen Pointerinhaltes erfolgt und im Anschluss an die Dereferenzierung der Pointerinhalt erhöht/erniedrigt wird. Beim Pre Inkrement/Dekrement `*++ptr` oder `*--ptr` wird der Pointerinhalt vor der Dereferenzierung erhöht/erniedrigt:

```
int var[8]={0x00112233,0x44556677,0x8899AABB,0xCCDDEEFF,
            0xFFEEDDCC,0xBBAA9988,0x77665544,0x33221100};
                  //Startadresse von var ist 0x404040
int *ptr=&var[4]; //ptr=0x404050
int val=0;

val=*ptr++;    //val   =*ptr_old             ptr_new= ptr_old+sizeof(int)
               //val   =0xFFEEDDCC           ptr_new= 0x404054
val=*++ptr;    //ptr_new=ptr_old+sizeof(int) val   =*ptr_new
               //ptr_new=0x404058            val   =0x77665544
val=*ptr--;    //val   =*ptr_old             ptr_new= ptr_old-sizeof(int)
               //val   =0x77665544           ptr_new= 0x404054
val=*--ptr;    //ptr_new=ptr_old-sizeof(int) val   =*ptr_new
```

```
              //ptr_new=0x404050            val   =0xFFEEDDCC
}
```

Ein Vorteil dieser 'Kombi' ist, dass sie nicht nur Tipparbeit erspart, sondern der Compiler passende Maschinensprachebefehle anwendet, welche die Programmausführung beschleunigen. Insbesondere bei Schleifen wird daher diese Kombination aus Performancegründen gerne angewendet:

```
void strcpy(char *dst, char *src) {
  for(  ; *src;  )
    *dst++=*src++;
  *dst++=*src++;  //Stringendezeichen noch mit kopieren
}
```

7.7.3 Zeigerdifferenz

Die Differenz zweier Zeiger `Syntax: Adresse/Zeiger - Adresse/Zeiger` gibt den Abstand der beiden Zeiger in Vielfachen des zugrundeliegenden Datentyps an. Die beiden Zeiger müssen vom identischen Datentyp sein. Der resultierende Datentyp der Subtraktion ist kein Pointer, sondern eine vorzeichenbehaftete Ganzzahl vom Datentyp ptrdiff_t (entspricht intptr_t, ist jedoch in der Header-Datei stddef.h definiert):

```
#include <stddef.h>  //Definition des Datentyps ptrdiff_t
struct xyz {int x,y,z;} xyz[]={{1,2,3},{4,5,6},{0,0,0}};
struct xyz *ptr1= xyz;
struct xyz *ptr2=&xyz[2];
ptrdiff_t  diff=ptr2-ptr1;  //=2
           diff=xyz -ptr2;  //=-2
struct abc {int a,b,c;} abc[]={{1,2,3},{4,5,6},{0,0,0}};
           diff=xyz-abc;    //KO, aufgrund von unterschiedliche Datentypen
```

Ein Anwendungsfall der Zeigerdifferenz ist die Bestimmung der Stringlänge:

```
size_t strlen(char *str) {
  char *lauf=str;
  for( ;*lauf++; );
  return (size_t)(lauf-str)-1;
  //size_t und ptrdiff_t sind identische Datentypen
  //so dass eine Cast ohne Datenverlust einhergeht!
}
```

7.7.4 Zeigervergleich

Ein Zeigervergleich `Syntax: Adresse/Zeiger < / <= / == / >= / > Adresse/Zeiger` gibt an, ob die linke Adresse kleiner/gleich/größer als die rechte Adresse ist. Die zu vergleichenden Adressen müssen vom identischen Datentyp sein!

```
int arr[10];
int *ptr=&arr[2];
if(ptr == &arr[2])   …         //TRUE
if(ptr <  &arr[2])   …         //FALSE
if(ptr <= &arr[2])   …         //TRUE
if(ptr >   arr   )   …         //TRUE
```

Hinweise:

- Da der Arrayname immer ein Zeiger ist (siehe Kapitel Array) ergibt folgender Ausdruck einen gültigen Syntax, bewirkt jedoch etwas anderes als gegebenenfalls erwartet (siehe Array:Vergleichen von Arrays[9]):

```
char arr1[3] = {4,5,6};    //0x100 0x101 0x102
char arr2[3] = {1,2,3};    //0x102 0x103 0x104
if(arr1 > arr2)   //Es werden die Speicheradresse der Arrays verglichen
                  //Nicht deren Inhalt
```

- Dies gilt auch für Objekte in C++, welche als Zeiger angelegt werden

```
#include <string>
std::string *str1 = new std::string("hallo1");
std::string *str2 = new std::string("hallo1");
if(str1 == str2)
  printf("Adressen sind identisch\n");
if(str1->compare(*str2)==0)
  printf("Inhalte sind identisch\n");
if(*str1 == *str2)
  printf("Inhalte sind identisch\n");
```

7.8 Zeigerinitialisierung

Zeiger müssen vor der ersten Nutzung (Dereferenzierung) mit einer gültigen Speicheradressen initialisiert werden! Nicht initialisierte Zeiger entsprechen nicht initialisierte Variablen, d.h,

- nicht initialisierte globale und statische lokale Zeiger werden mit 0 initialisiert, d.h. eine Dereferenzierung würde den Inhalt der Speicheradresse 0 beschreiben/lesen. Dieser ist bei Windows/Linux/macOS nicht dem Prozess zugeordnet (genau aus diesem Grund) so dass ein Zugriff auf diese Speicheradressen zu einem Programmabsturz führt!
- nicht initialisierte lokale Zeiger sind mit einem Zufallswert belegt, d.h. eine Dereferenzierug wird höchstwahrscheinlich den Speicherinhalt einer dem Prozess nicht zugewiesenen Speicheradresse beschreiben/lesen.

Beispiele:

```
char *ptr1;     //Im Falle einer globalen Variablen wird diese mit 0
                //initialisiert
*ptr1=47;       //Schreiben der Konstanten 47 in die Speicheradresse 0
                //welche zumeist dem Programm vom Betriebssystem
                //nicht zugewiesen wird → Programmabsturz
void foo(void) {
  char *ptr2;   //Im Falle einer lokalen Variablen wird diese
                //nicht initialisiert, enthält somit eine 'Zufallsadresse'
  *ptr2=47;     //Schreiben der Konstanten 47 in diese Zufallsadresse
                //führt wahlweise zum Programmabsturz oder Änderung
}               //anderen Variablen
```

9 Kapitel 8.12 auf Seite 195

Bei der Initialisierung von Zeigern mit Variablen muss die Gültigkeitsdauer der Adresse berücksichtigt werden:

- Initialisierung mit Adresse von globalen/static lokalen Variablen

  Globale Variablen und static lokale Variablen sind während der gesamten Programmlaufzeit gültig und Zeiger auf diese Adressen sind somit immer gültig. Solch initialisierte Zeiger können zu jedem Zeitpunkt dereferenziert werden:

```
int glob=4711;
int *ptr;
void func1(void) {
  ptr=&glob;   //Initialisierung mit Adresse einer globalen Variable
}
void func2(void) {
  *ptr=1147;   //Nach Initialisierung ist dieser Pointer bis zum
               //Programmende gültig
}
```

- Initialisierung mit Adresse von lokalen Variablen

  Lokale Variable / Parameterübergabevariablen sind nur während der Laufzeit der Funktion / des Blockes gültig. Wird ein Zeiger mit solch einer Adresse initialisiert, so darf eine Dereferenzierung des Zeigers nur erfolgen, solange die dazugehörige Variable 'existiert'. Nach Verlassen des Blockes / Beendigung der Funktion darf solch ein Zeiger nicht mehr dereferenziert werden:

Korrekt

```
void func(int par)
{
 int lok;
 int *ptr1=&par;
 int *ptr2=&lok;

 *ptr1=0;  //OK
 *ptr2=0;  //OK
}
```

Inkorrekt

```
int *func(void)
{
 int lok=7;
 return(&lok);
}
int main(void)
{
 int *ptr;
 ptr=func();
 *ptr=7; //KO
 return 0;
}
```

Korrekt

```
void func1(int *ptr)
{
 *ptr=8; //OK
}
void func2(void)
{
 int lok=7;
 func1(&lok);
}
```

Inkorrekt

```
int *ptr;
void func(void)
{
 int lok=7;
 ptr=&lok;
 *ptr=8;  //OK
}
int main(void)
{
 func();
 *ptr=7; //KO
 return 0;
}
```

- Initialisierung mit Adresse vom Heap

  Der vom Heap angeforderte Speicher bleibt so lange reserviert/erhalten, bis er explizit freigegeben wird. Nach Freigabe ist kein Zugriff mehr auf den Speicher erlaubt:

```
char *ptr=malloc(100);
strcpy(ptr,"hello world"); //OK
free(ptr);
printf("%s",ptr);          //KO
```

Daher empfiehlt sich, mit der Freigabe des Speichers alle auf diesen Speicherbereich zeigenden Zeiger auf NULL zusetzen:

```
char *ptr=malloc(100);
strcpy(ptr,"hello world"); //OK
free(ptr);
ptr=NULL;
ptr[0]='a';               //KO, jedoch mit definierten Ende
                          //    d.h. Programmabsturz
//etwas Eleganter unter Nutzung des Preprozessors
#define SETNULL(ptr) ({typeof(ptr)dummy=ptr; ptr=NULL; dummy;})
char *ptr=malloc(100);
strcpy(ptr,"hello world");
free(SETNULL(ptr));
```

Hinweise:

- Im Falle von nicht initialisierten Zeigern oder Zeigern deren Adresse die Gültigkeit verloren haben, spricht man von Hängenden / wilden Zeigern (englisch dangling pointer). (Siehe Hängender Zeiger[10])
- C/C++ führt zur Laufzeit keine Kontrolle durch, ob der Zeiger auf eine gültige Adresse zeigt. Zur Compilezeit findet solch eine Überprüfung nur an wenigen Stellen statt. Da hängende Zeiger eine häufige Fehlerquelle sind, wäre hier Optimierungsbedarf beim Compiler sinnvoll.

7.9 Void-Zeiger

Bei der Zuweisung und dem Vergleich von Zeiger müssen beiden Operanden vom identischen Datentyp sein. Eine Ausnahme stellt der Void-Zeiger (Generische Zeiger, anonyme Zeiger, untypisierter Zeiger) dar. Dieser ist zu jedem anderen Zeiger kompatibel, so dass ein Vergleich / Zuweisung ohne explizites Cast möglich ist. C++ ist im Umgang mit void-Zeigern etwas restriktiver. Eine Zuweisung eines void-zeigers an einen typisierten Zeiger ist nicht möglich. Aufgrund des zugrundeliegenden void Datentyps kann der void-Zeiger nicht dazu genutzt werden, Inhalte zu dereferenzieren:

```
int   var;
int  *ptri;
void *ptrv;
ptri=&var;
ptrv=&var;

ptri=ptrv; //OK in C, KO in C++
ptrv=ptri;

*ptri=12;
*ptrv=12;   //KO, ein Void-Zeiger kann nicht dereferenziert werden

if(ptri >= ptrv) ...
if(ptrv >= ptri)
```

10 https://de.wikipedia.org/wiki/H%C3%A4ngender%20Zeiger

Anwendungen:

- Die Malloc Funktion gibt einen void-Zeiger zurück, so dass der Rückgabewert der malloc Funktion in C jedem Zeiger ohne explizites Cast zugewiesen werden kann (in C++ ist hier ein explizites Cast notwendig!):

```
int *ptr1=malloc(10*sizeof(int));
struct {int x,y,z;} *ptr2=malloc(12);
```

- Mittels den memxxx() Funktionen können Speicherbereiche:
  - Auf einen Wert gesetzt/initialisiert werden memset()
  - Miteinander verglichen werden memcmp()
  - Kopiert werden memcpy() (Speicherbereiche dürfen sich nicht überlappen)
  - Verschoben werden memmove() (Speicherbereiche dürfen sich überlappen)

  Diese Funktionen sind unter anderem Hilfreich beim Umgang mit Arrays (sieh Array[11]). Damit diese unabhängig vom Datentyp des Zeigers sind (=generische Funktion), nutzen diese Funktionen den void-Zeiger als Übergabeparameter:

```
#include <string.h>
//enthält u.A. folgende Prototypen
//void *memcpy(void *dst,void *src,size_t n);
//int   memcmp(void *s1, void *s2 ,size_t n);
```

- Soll eine Struktur vom Aufrufer 'versteckt' werden, so wird die Strukturdefinition nicht in die Header-Datei, sondern in der C-Datei beschrieben. Der Aufrufer speichert die Adresse dann in einem void-Zeiger:

class.h

```
void *class_init(void);
```

main.c

```
#include "class.h"
int main(void) {
   void *obj;
   obj=class_init();
   //Aufrufer kennt struct class
   //nicht und kann somit nicht
   //auf die Strukturelemente
   //x,y,z zugreifen
   //Die Strukturelemente
   //sind somit 'private'
    …
   return 0;
}
```

class.c

```
#include "class.h"
struct class {
     int x,y,z;
};

void *class_init(void)
{
  struct class *me;
  me=malloc(sizeof(struct class));
   …
  return me;  //Kein explizites Cast
              //notwendig
}
```

11 Kapitel 8 auf Seite 179

7.10 Null-Zeiger

Ein Zeiger auf die (eigentlich gültige) Speicheradresse 0 (=Null-Zeiger) wird in C/C++ als nicht initialisierter Zeiger angesehen. Ergänzend wird mit dem Null-Zeiger als Rückgabewert von Funktionen gesagt, dass die Funktion die geforderte Funktionalität nicht ausgeführt hat.

Der Test, ob ein Zeiger eine gültige Speicheradresse hat (d.h. ungleich 0 ist) erfolgt über einen Zeigervergleich. Dieser bedingt, dass beide Operanden ein Zeiger sind. Zur Vermeidung einer Datentypanpassung empfiehlt sich ein void-Zeiger. In der Header-Datei `stddef.h` ist dazu folgendes Makro enthalten:

```
#define NULL ((void *)0)
```

Hier wird die Ganzzahl 0 zu einem void-Zeiger auf die Speicheradresse 0 gecastet.

Anwendungen:

- Zeiger ist nicht intialisiert:

```
#include <stddef.h>
char *ptr=NULL;
printf("%s",ptr?:(char *)"(nil)");

ptr=(char *)malloc(100 * sizeof(char));
free((void *)ptr);
ptr=NULL;
```

- Funktion hat die geforderte Funktionalität nicht ausgeführt:

```
//Malloc() gibt im Fehlerfall NULL zurück
int *ptr=malloc(100);
if(ptr!=NULL)  //zum Überprüfen, ob Malloc dem Aufrufer die
               //angeforderte Speichergröße bereitstellen kann
if(strstr("Test","st")==NULL)  //Zum überprüfen, ob der Substring
                               //im String vorhanden ist
```

- Der Nullzeiger wird oft bei Arrays auf Zeigern genutzt, um entsprechend dem Stringendezeichen zu signalisieren, dass dies der letzte Eintrag ist:

```
#include <stddef.h>
char const * const strarray[]={"hallo","Du",NULL};
//For-Schleife über Index-Variable
for(int lauf=0;strarray[lauf];lauf++)
  printf("%s\n",strarray[lauf]);

//Optimierte For-Schleife über Zeiger
for(char const*const *ptr=strarray;*ptr; ptr++)
  printf("%s\n",*ptr);
//Wird auch bei argv angewendet
for(char **ptr=argv;*argv;argv++)
  printf("%s\n",*ptr);
```

Hinweise:

- Entgegen Windows/Linux, bei welche die Speicheradresse 0 nicht dem Prozess zugewiesen wird und eine Dereferenzierung folglich zu einem Laufzeitfehler führt, ist die Speicheradresse 0 bei Embedded Systemen sehr wohl dem 'Prozess' zugewiesen. Hier befindet sich zumeist der Interrupt-Vektor-Tabelle, so dass ein Schreiben auf diese Adresse gravierende Auswirkung haben könnte
- strcpy() und viele andere Funktionen gehen davon aus, dass eine gültige Adresse übergeben wurde. Wenn bspw. strcpy einen Nullzeiger übergeben wird, greift strcpy auf die Speicheradresse 0 zu, was zumeist zu einem Programmabsturz führt
- Für die Ausgabe eines Strings mittels printf() wird die Startadresse des Strings übergeben. In einigen Anwendungsfällen überprüft printf(), ob es sich um einen Nullzeiger handelt und gibt "(null)" aus. In anderen Anwendungsfällen dereferenziert printf() den NULL Zeiger, was zu einem Programmabsturz führt
- Die Header-Datei stddef.h wird durch viele Header-Dateien inkludiert (z.b. stdlib.h, stdio.h), so dass diese nicht gesondert inkludiert werden muss

7.11 Zeiger auf Zeiger

Nicht nur eine Variable, sondern auch eine Zeigervariable hat eine Speicheradresse. Nutzt man die Adresse einer Zeigervariablen, spricht man von Zeiger auf Zeiger:

```
int    var;
int   *ptr=&var;         //Datentyp: (int *)   → 'Einfacher' Zeiger
int  **ptrptr=&ptr;      //Datentyp: (int **)  → Zeiger auf Zeiger
int ***ptrptrptr=&ptrptr;//Datentyp: (int ***) → Zeiger auf Zeiger auf Zeiger
```

Bei der Dereferenzierung eines Zeigers auf einen Zeiger wird entsprechend der Datentypsicht jeweils ein * entfernt. Eine Mehrfachdereferenzierung ist möglich:

```
  *ptrptrptr=NULL;    //aus Datentypsicht:   *(int ***) → (int **)
 **ptrptrptr=NULL;    //aus Datentypsicht:  **(int ***) → (int *)
***ptrptrptr=NULL;    //aus Datentypsicht: ***(int ***) → (int)
 **ptrptr   =NULL;    //aus Datentypsicht:  **(int **)  → (int)
  *ptrptr   =NULL;    //aus Datentypsicht:   *(int **)  → (int *)
```

Ein Zeiger auf einen Zeiger entspricht einem 'normalen' Zeiger, d.h. er beinhaltet die Adresse einer anderen Variablen und belegt somit Speicherplatz wie ein 'normaler' Zeiger:

```
int   var1=0x01020304;   //*1)
int   var2=0x55667788;
int  *ptr1=&var1;        //*2)
int  *ptr2=&var2;
int **ptrptr=&ptr1;      //*3)

ptr2=*ptrptr;   //Gibt die Adresse der Variablen ptr1 zurück *4)
var2=**ptrptr;  //Gibt den Inhalt der Variablen var1 zurück *5)
*ptrptr=&var2;  //Ohne Worte, gerne im Dump nachvollziehen
ptrptr=&ptr2;   //Ohne Worte, gerne im Dump nachvollziehen
&&var;   //KO Der erste Adress-Operator gibt eine Konstante zurück
         //   von welcher keine Adresse abgeleitet werden kann
//Ausgabe
//Speicherdump: 0x404040 .. 0x40405f Mode=8-Bit
//0x00000000404040: 04 03 02 01 88 77 66 55
```

```
//0x00000000404048: 40 40 40 00 00 00 00 00
//0x00000000404050: 44 40 40 00 00 00 00 00
//0x00000000404058: 48 40 40 00 00 00 00 00
```

- 1) Die Variable var1 belegt den Speicherbereich von 0x404040..0x404043 und beinhaltet den Initialisierungswert
- 2) Der Zeiger ptr1 belegt den Speicherbereich von 0x404048..0x40404F und beinhaltet die Adresse der Variablen var1 (0x404040)
- 3) Der Zeiger auf Zeiger ptrptr belegt den Speicherbereich von 0x404058..0x40405F und beinhaltet die Adresse des Zeigers ptr1 (0x404048)
- 4) Mit der 'einfachen' Dereferenzierung des Zeigers auf Zeiger wird zunächst der Inhalt der Variablen ptrptr (0x404058..0x40405F) gelesen. Dieser gelesene Wert (0x404048) wird als Startadresse für einen weiteren Lesezyklus genutzt. Da der zugrundeliegende Datentyp des Zeigers auf Zeiger ein Zeiger ist, werden 8-Byte gelesen. Der ermittelte Wert (0x404040) ist die Startadresse der Variablen var1
- 5) Mit der 'doppelten' Dereferenzierung des Zeigers auf Zeigers wird zunächst der Inhalt der Variablen ptrptr(0x404058..0x40405F) gelesen. Dieser gelesen Wert (0x0404048) wird als Startadresse für einen weiteren Lesezyklus genutzt. Da der zugrundeliegende Datentyp des Zeigers auf Zeiger ein Zeiger ist, werden 8-Byte gelesen. Dieser gelesene Wert (0x404040) wird erneut als Startadresse für einen Lesezyklus genutzt. Da der zugrundeliegende Datentyp des Zeigers ein int ist, werden 4-Byte gelesen. Der ermittelte Wert (0x01020304) ist der Inhalt der Variablen var1

Anwendung:

- Soll die aufgerufene Funktion Speicher reservieren und diese an dem Aufrufer zurückgeben, so gibt es zwei Möglichkeiten. Entweder über den Rückgabewert oder über einen Funktionsparameter. Im letzteren Fall muss der Parameter ein Zeiger auf einen Zeiger sein:

```
//Rückgabe der Speicheradresse über Rückgabewert der Funktion
int *func1(size_t size) {
  int *ptr=malloc(size);
  return ptr;
}
//Rückgabe der Speicheradresse über Funktionsparameter
int func2(int **ret, size_t size) {
  int *ptr=malloc(size);
  if(ptr==NULL)
    return -1;
  *ret=ptr;
  return 0;
}
```

Hinweis:

- Zeiger auf Zeiger sind ähnlich wie eine komplizierte mathematische Formel nur schwer nachvollziehbar. Daher sollten solche Zeiger nur dort eingesetzt werden, wo sie unbedingt benötigt werden

7.12 Zeiger bei Funktionsaufrufen

Ein wichtiger Einsatzbereich von Zeigern ist die Verwendung von Zeigern als Übergabeparameter und Rückgabeparameter bei Funktionsaufrufen. Vorrangiges Ziel hier ist:

- einerseits das (zeit)aufwendige Kopieren von Daten zu vermeiden und
- ergänzend dem aufgerufenen (Callee) zu ermöglichen, direkt Werte des Aufrufers (Caller) zu verändern

Prinzipiell erfolgt bei einem Funktionsaufruf immer ein Kopieren/Zuweisen der Werte des Callers in die lokale Parametervariablen des Callees (Call by Value). Wird ein Zeiger übergeben, hat der Callee über die Dereferenzierung des Zeigers die Möglichkeit, die Variablen des Caller zu verändern (Call by Referenz).

7.12.1 Call by Value

Die Parameterübergabe bei Funktionen basiert darauf, dass der Inhalt des Caller-Parameters in die lokale Callee-Variable kopiert wird:

```
void func(int lok) {
  lok=lok*2;     //Änderung von lok bewirkt keine Änderung von 7/var
}
int main(void) {
  int var=8;
  func(7);       //lok=7
  func(var);     //lok=var
                 //var behält seinen Wert unabhängig vom Aufruf von func bei
  func(var+7);   //lok=var+7
  return 0;
}
```

Die Konstante 7 / der Inhalt der Variablen var wird mit dem Funktionsaufruf in die lokale Variable lok der Funktion func kopiert. Eine Änderung der lokalen Variablen lok hat folglich keine Auswirkung auf den Inhalt der 'Ursprungsvariablen'.

7.12.2 Call By Reference

Wird anstatt dem Inhalt einer Variablen die Adresse der Variablen übergeben, so hat der Callee nun die Möglichkeit, über die Adresse den Inhalt der Ursprungsvariablen zu ändern:

```
void func(int *ptr) {
  *ptr=10;        //Änderung des Inhaltes von var
  ptr=ptr+7;      //Änderung der lokalen Variablen ptr
}
int main(void) {
  int var=8;   //bspw 0x404060
  func(&var);  //ptr=&var
               //var hat nach Aufruf von func einen anderen Wert
  return 0;
}
```

Auch hier findet ein Kopiervorgang statt, wobei hier die Adresse der Variablen var in die lokale Variable vom Typ Zeiger kopiert wird. Über die Dereferenzierung kann auf den Inhalt und damit auf die Ursprungsvariable zugegriffen werden. Eine Änderung der lokalen Zeigervariablen ptr ist möglich, ändert dann einzig die Zeigervariable, aber nicht den ursprünglich referenzierten Inhalt.

Anwendungen:

- Als alternative zum Rückgabewert einer Funktion, insbesondere dann, wenn mehrere Rückgabewerte benötigt werden:

```
void func(double *ret1, double *ret2, double value)  {
  *ret1=sin(value);
  *ret2=cos(value);
}
//Auch strcpy(), strcat(), snprintf(), ..
//bekommen als ersten Parameter einen Zeiger übergeben, in welchen das
//Ergebnis (der Rückgabewert) geschrieben wird
```

- der eigentliche Funktionsrückgabewert als Funktionsstatus angesehen wird

```
enum status {OK,KO};
enum status func(int *retvalue, int para1, int para2) {
  if(!(para1!=0 && para2!=0))
    return KO;
  *retvalue=para1+para2;
  return OK;
}
```

- Für eine schnellere Programmausführung, damit nicht der gesamte Inhalt kopiert werden muss, sondern nur der Zeiger übergeben wird (insb. Bei komplexen Strukturen)

```
struct xyz {
  int arr1[1000];
  int arr2[100];
} var;
struct xyz {
  int arr1[1000];
  int arr2[100];
} var;
struct xyz func_slow(struct xyz lok)
{
  //Hier werden jeweils 1100*4 Bytes beim Aufruf
  //und beim Beenden der Funktion kopiert
  return lok;
}
func_slow(var);

void func_fast(struct xyz *ptr) {
  //Hier werden nur 8-Bytes kopiert.
}
func_fast(&var);
```

7.12.3 Werterückgabe

Neben der Verwendung von Zeigern für die Parameterübergabe kann auch ein Zeiger zurückgegeben werden:

```
char *strstr(const char *haystack, const char *needle);
char *strdup(char *src);
```

Dies ist sinnvoll/notwendig, wenn entweder größere Datenmengen zurückgegeben werden sollen oder die Anzahl der zurückgegebenen Daten sich erst zur Laufzeit ergibt. Besondere Achtsamkeit gilt hier der 'Adressfindung'. Der Speicherbereich, auf welchen die Adresse zeigt, muss nach Beenden der Funktion noch gültig sein. Damit verbietet sich die Rückgabe der Adresse einer lokalen Variablen (siehe auch Zeigerinitialisierung[12]).

Über die Werterückgabe wird oft der Status der Funktion zurückgeben. Ein Fehler bei der Funktionsausführung wird im Falle von Zeigern mit einem Zeiger auf die Adresse NULL dargestellt

7.13 Const-Zeiger

Nicht nur Variablen, sondern auch Zeiger können 'Konstant', also nicht änderbar/beschreibbar sein. Im Falle von Zeigern müssen zwei Fälle unterschieden werden. Einerseits kann die Zeigervariable als solches nicht änderbar sein und anderseits kann der Inhalt, auf welchen der Zeiger zeigt, nicht änderbar sein:

- Zeigervariable ist Kontant

`Syntax: Datentyp * const Variablenname;`

Die Zeigervariable ist Kontant und kann nicht geändert werden.

Der Speicherinhalt, auf welchen der Zeiger zeigt, kann geändert werden

```
char var;
char * const ptr=&var;
*ptr='a';   //OK
ptr++;      //KO
```

- Dereferenzierung ist Kontant

`Syntax: Datentyp const * Variablenname;`

`Syntax: const Datentyp * Variablenname;`

Die Zeigervariable ist nicht konstant und kann geändert werden

Der Speicherinhalt, auf welchen der Zeiger zeigt, ist konstant und kann nicht geändert werden

```
char var;
char const * ptr=&var;
*ptr='a';   //KO
ptr++;      //OK
```

- Zeigervariable und Dereferenzierung sind Konstant

12 Kapitel 7.8 auf Seite 156

```
Syntax: Datentyp const * const Variablenname;
```

```
Syntax: const Datentyp * const Variablenname;
```

Sowohl die Zeigervariable, als auch der Speicherinhalt, auf welchen der Zeiger zeigt, sind Konstant

```
char var;
char const * const ptr=&var;
*ptr='a';   //KO
ptr++;      //KO
```

Anwendungen:

- Werden Konstanten als normale Variablen angelegt und diese im Folgenden über Zeiger 'verwaltet', so sollten auch die Zeiger als Zeiger vom Typ auf eine konstante Dereferenzierung sein:

```
int  const var=4711;
int * ptr1=&var;
*ptr1=13;                   //KO Über die Dereferenzierung von ptr1
                            //   kann var geändert werden
int  const * ptr2=&var;     //Korrekt

char *str1="hallo";         //KO Änderungen über Dereferenzierung
                            //   führen zum Programmabsturz, da
                            //   String-Konstante in Read-Only Speicher
                            //   gehalten wird!
char const * str="hallo";   //Korrekt
```

- Zur Darstellung bei der Parameterübergabe mit Zeigern, dass der 'Inhalt' des Zeigers durch den Callee nicht geändert wird (Read Only Parameter)

```
char *strcpy(char *dst, char const *src); //const hier als Zusicherung
                        //von strcpy, dass der Inhalt von src nicht
                        //geändert wird.
int printf(const char *fmt , …);          //Dito
```

- Ein mit malloc() reservierter Speicher muss mit free() freigegeben werden. Dazu ist die von malloc erhaltene Adresse zu nutzen. Dieser Zeiger darf nicht verändert werden:

```
char * const ptr = malloc(100);
ptr++;  //KO
free(ptr);
```

- Zur Nachbildung einer Referenz, welche mit dem Anlegen initialisiert werden muss und danach nicht mehr geändert werden kann:

```
int var1;
int * const ptr=&var1;
```

```
int var2;
ptr=&var2;   //KO, Referenzen können nicht geändert werden
```

Leider findet man die Anwendung dieser Regeln nur selten in der praktischen Anwendung. Insbesondere unter Beachtung dieser Regeln können mit C/C++ einige Laufzeitfehler vermieden werden!

7.13.1 Typkonvertierung

Das Schlüsselwort const ist Bestandteil des Datentyps. Folglich gilt es dies bei der Zuweisung (incl. Parameterübergabe) von const zu nicht const Datentypen zu berücksichtigen.

C wendet hierbei folgende implizite Typkonvertierungen an:

```
Zieldatentyp  = Quelldatentyp
char const *  = char const *   <-- Keine Typkonvertierung notwendig
char const *  = char       *   <-- char * wird implizit zu char const* gewandelt
char *        = char const *   <-- char const * wird implizit zu char *
 gewandelt
                                    unter ggf. ergänzenden Ausgabe einer Warning
                                    Da nun über die Dereferenzierung des neuen
                                    Zeigers der eigentlich konstante Inhalt
 geändert
                                    werden kann, muss diese Warning als ERROR
                                    angesehen werden
char * const  = char * const   <- Keine Typkonvertierung notwendig
char *        = char * const   <- char *const wird implizit zu char * gewandelt
char * const  = char *         <- char * wird implizit zu char * const gewandelt
```

Das obige negative Beispiel führt in C nur in selten Fällen zu einer Compiler-Meldung. Der Programmierer ist somit gefordert, auf Typverletzung zu achten. In C++ führt diese korrekterweise zu einer Compiler-Meldung:

```
char *str="hallo";  //"hallo" ist ein Const-Zeiger
*str='p';           //Dereferenzierung eines nicht
                    //Const-Zeigers ist möglich
                    //--> Programmabsturz

strcpy("dst","src");//Const-Zeiger als Ziel für
                    //Kopierfunktion
                    //-> Programmabsturz
strtok("token=value","="); //strtok() verändert Inhalt des
                           //ersten Parameters
                           //--> Programmabsturz
```

7.13.2 Sonstiges

Bei Zeigern auf Zeiger kann der Zeiger als solches, die erste Dereferenzierung und die zweite Dereferenzierung konstant sein:

```
int   var;
int  *ptr=&var;
int      *     *      ptrr1 = &ptr;
int      *     *const ptrr2 = &ptr;
int      *const*      ptrr3 = &ptr;
int      *const*const ptrr4 = &ptr;
int const*     *      ptrr5=(int const** )&ptr;
...
```

7.14 Zeiger auf Funktionen

Ähnlich wie bei Zeigervariablen, welche die Adresse einer Variablen beinhalten und auf dessen Inhalt über die Dereferenzierung zugegriffen werden, gibt es Zeiger auf Funktionen, die die Startadresse einer Funktion beinhaltet und welche bei der 'Dereferenzierung' aufgerufen werden.

Der Datentyp zum Anlegen eines Funktionszeigers entspricht dem Funktionsprototyp, mit dem Unterschied, dass der Funktionsname der Variablenname ist und dieser geklammert und mit einem Stern (für Zeiger) versehen wird. Die Namen der Übergabeparameter sind optional:

```
//Über Funktionszeiger aufzurufende Funktion
void func(int par) {
  printf("%d\n",par);
}

void (*fptr1)(int par); //Funktionseiger auf obige Funktion
void (*fptr2)(int);     //Dito

fptr1=&func;            //Zuweisen der Funktionsadresse an Funktionszeiger
fptr2=func;             //Dito

(*fptr1)(4);            //Dereferenzierung=Aufruf
fptr2(3);               //Dito

fptr1(4,5);             //KO, da Parameter nicht mit Datentyp übereinstimmt
fptr1("hallo Welt");    //KO, da Parameter nicht mit Datentyp übereinstimmt
if(fptr1(4)==7)         //KO, da Rückgabewert nicht mit Datentyp übereinst.
```

7.14.1 Typedef

Auch von einem Funktionszeiger kann über Typedef ein Alias erzeugt werden. Der Alias steht jedoch nicht am Ende des Typedefs, sondern anstelle des Funktionszeigers:

```
typedef void * MALLOC_PTR(size_t);   //MALLOC_PTR ist der ALIAS
                                     //Sternchen gehört zu void
typedef void (*FREE_PTR)(void *ptr); //FREE_PTR ist der ALIAS

MALLOC_PTR  *malloc_ptr;    //Anlegen des Funktionszeigers
FREE_PTR     free_ptr;

malloc_ptr=&malloc;         //Zuweisung des Funktionszeigers
free_ptr  =&free;           //mit einer Startadresse

void *ptr=malloc_ptr(400);  //Entspricht dem Aufruf der malloc Funktion
free_ptr(ptr);
```

7.14.2 Typkonvertierung

Über explizite Cast auf einen anderen Funktionszeigerdatentyp kann ein Funktionszeiger eines Datentyps in einen anderen Funktionszeiger konvertiert werden. Dies ist jedoch eher ein theoretischer Aspekt und hat in der Praxis keine Anwendung, da dies zu einem fehlerhaften Funktionsaufruf und ggf. zum Programmabsturz führt:

```
void dummy(void) {
  printf("dummy() called");
}
void (*fptr)(int);
fptr=(void (*)(int))&dummy;  //fptr wird mit einer Funktion mit anderem
                             //Parameter initialisiert
fptr(10);                    //dummy() wird nun mit einem Parameter
                             //aufgerufen
```

7.14.3 Anwendungen

Funktionszeiger finden an diversen Stellen Anwendung:

- Angaben von Callback Funktionen, die bei bestimmten Ereignissen aufgerufen werden:

```
void callbackfcn(int ereignisnr) {
  //Reaktion auf Ereignis
}

//Globale Variablen zum Speichern des Funktionszeigers
void(*ActionListener_cb)(int)=NULL;

void AddActionListener( void(*fcn)(int) ) {
  ActionListener_cb =fcn;
}

int main(int argc, char *argv[]) {
  AddActionListener(callbackfcn);
  while(1) {
    int but=mouse_get();
    if((but!=0) && (ActionListener_cb!=NULL))
      ActionListener_cb(but);
  }
  return 0;
}
```

Insbesondere GUI-Anwendung nutzen dieses Feature zur Trennung von Applikation und Betriebssystem. Zunächst reagiert das Betriebssystem auf die Tastendrücke einer Maus. In Abhängigkeit der Mausposition ruft das Betriebssystem die hinterlegte CallBack Funktion auf.

Grundprinzip hierbei ist die ereignisorientierte Programmierung (siehe Ereignis (Programmierung)[13])

- Generische Such-/Sortieralgorithmus. Über Funktionszeiger kann einer generischen Such-/Sortieralgorithmus eine 'Funktion' übergeben werden, welche die Such-/Sortierbedingung beinhaltet:

13 https://de.wikipedia.org/wiki/%20Ereignis%20%28Programmierung%29

```
//Zu sortierende Datenstruktur
struct xyz {int x,y,z;};

//Vergleichsfunktion für obigen Datentyp
int sort_x(const void *p1, const void *p2) {
  const struct xyz *ptr1=p1;
  const struct xyz *ptr2=p2;
  if(ptr1->x < ptr2->x)
    return -1; //Zur Darstellung dass das erste Argument kleiner
  else if(ptr1->x > ptr2->x)
    return +1;  //Zur Darstellung das das erste Argument größer
  else
    return 0;
}
//Weitere Vergleichsfunktion für obigen Datentyp
int sort_y(const void *p1, const void *p2) {
  return ((const struct xyz*)p1)->y - ((const struct xyz *)p2)->y;
}

int main(int argc, char *argv[]) {
  struct xyz arr[]={{1,3,2},{7,1,9},{0,2,11}};
  qsort(arr,sizeof(arr)/sizeof(arr[0]),sizeof(struct xyz),sort_x);
  for(int lauf=0;lauf<3;lauf++)
    printf("%d",arr[lauf].x);

  qsort(arr,sizeof(arr)/sizeof(arr[0]),sizeof(struct xyz),sort_y);
  for(int lauf=0;lauf<3;lauf++)
    printf("%d",arr[lauf].y);
  return 0;
}
```

- DLL/Shared Librarys entsprechen einer Funktionsbibliothek, welche entweder zum Start der Anwendung oder zur Laufzeit durch die Anwendung zur eigentlichen Anwendung dazu gebunden werden. Werden die Funktionen zum Programmstart (durch den Loader) dazu gebunden, so übernimmt der Loader der Adressauflösung, so dass diese Funktionen wie 'normale' Funktionen aufgerufen werden können (**Early Binding**). Die Standard-C-Library libc.so, die Mathematische Library libm.so sind typische Beispiele hierfür. Werden die Funktion während der Laufzeit eingebunden, so ist die Anwendung für die Adressauflösung zuständig (**Late Binding**). In Posix Konformen Betriebssystemen erfolgt dies über dlopen() dlsym() und dlclose():

Anwendung, welche eine Libray zur Laufzeit einbindet	**Library: Ansammlung von Funktionen** **Entspricht einem C-Programm ohne main() Funktion**

```
#include <dlfcn.h>   //dlopen()/..

int main(int argc,char *argv[])
{
//Late Binding
//Einbinden der Library erfolgt händisch
void *handle= dlopen("./bin/liblib.so",
                     RTLD_LAZY|RTLD_LOCAL);

//Abfrage der Adresse der setter-
Funktion
void *ptr=dlsym(handle,"setter");
//Konvertierung in den korrekten Datentyp
void (*setter)(int var)=ptr;

//Abfrage der Adresse der getter-
Funktion
int (*getter)(void)=dlsym(handle,"getter");

//Aufruf der Funktion
setter(4711);
printf("%d",getter());

dlclose(handle);
}
```

```
void __attribute__ ((constructor)) my_load(void);
void __attribute__ ((destructor))  my_unload(void);

// Called when the library is loaded
// and before dlopen() returns
void my_load(void) {
  printf("LIB: " __FILE__ " attached\n");
}

// Called when the library is unloaded
// and before dlclose() returns
void my_unload(void) {
  printf("LIB: " __FILE__ " detached\n");
}

int glob_data=9;
int getter(void) {
  printf("Lib: Get %d\n",glob_data);
  return glob_data;
}

void setter(int var) {
  printf("Lib: Set %d=%d\n",glob_data,var);
  glob_data=var;
}
```

Dieser Anwendungsfall wird bspw. in Python genutzt, welche komplexe Rechenaufgaben über Librarys tätigt. Diese Librarys werden erst zur Laufzeit eingebunden.

- Eine virtuelle Methode[14] ist in der objektorientierten Programmierung eine Methode einer Klasse, deren Einsprungadresse erst zur Laufzeit ermittelt wird. Dieses sogenannte dynamische Binden ermöglicht es, Klassen von einer Oberklasse abzuleiten und dabei Funktionen zu überschreiben bzw. zu überladen:

14 https://de.wikipedia.org/wiki/Virtuelle_Methode

```
#include <iostream>

using namespace std;

struct Tier {
  // Rein virtuelle Methode
  virtual void essen() = 0;
};

struct Wolf: Tier {
  // Implementierung der virt. Methode
  void essen() {
    cout << "Essen Fleisch" << endl;
  }
};
struct Kuh: Tier {
  // Implementierung der virt. Methode
  void essen() {
    cout << "Sind Vegetarier!" << endl;
  }
};
```

```
int main(void) {
  //Tier obj0; kann aufgrund der
  //virtuellen Methode nicht
  //als Objekt angelegt werden
  Wolf obj1;
  Kuh  obj2;

  Tier *obj;
  obj=&obj1;
//Aufruf der virtuellen Mehoden
  obj-
>essen();
  obj=&obj2;
  obj-
>essen();
  return 0;
}
```

Der Compiler setzt virtuelle Methoden über Funktionstabelle / VTABLE (= Arrays von Zeiger auf Funktionen) um (siehe Tabelle virtueller Methoden[15]). In der Hauptklasse wird für jede virtuelle Methode ein Funktionszeiger als verstecktes Attribut integriert. Mit Aufruf des Konstruktors der abgeleiteten Klasse wird dieser Funktionszeiger auf die passende Funktion gesetzt. Beim Aufruf der virtuellen Methode wird eine Wrapper-Methode aufgerufen, welche den Funktionszeiger dereferenziert.

7.14.4 Sonstiges

Die Schreibweise für Funktionszeiger ist für sich schon 'kryptisch'. Werden Funktionszeiger als Rückgabewert von Funktionen benutzt, so lässt sich der eigentliche Datentyp nicht auf den ersten Blick ermitteln:

```
int (*fpfi (int (*)(long),int)) (int, ...);
//Dies ist ein Prototyp für eine Funktion, welche 2 Übergabeparameter hat.
//Der erste Parameter ist ein Zeiger auf eine Funktion mit Rückgabewert int
//und Übergabewert long und der zweite Parameter ist vom Typ int.
     //Rückgabewert der Funktion ist ein Zeiger auf eine Funktion, welche int
//zurückgibt und als Parameter ein int und belieibig viele weitere bekommt
int par_func(long);  //Protoyp
int main(int argc, char *argv[]) {
  fpfi(&par_func,3)(2);  //Aufruf der obigen Funktion
  return 0;
}
int par_func(long par) {
  return printf("par_func(%ld) called\n",par);
}
int ret_func(int par,...) {
  return printf("ref_func called(%d)\n",par);
}
```

15 https://de.wikipedia.org/wiki/Tabelle%20virtueller%20Methoden

```
//Implementierung der obigen Funktion
int (*fpfi (int (*fptr)(long)     ,int par))    (int,...) {
  printf("fpfi called(%p,%d)\n",fptr,par);
  fptr((long)2);
  return &ret_func;
}
```

Hier empfiehlt sich die Nutzung von Aliasen:

```
//Original Prototype
int (*fpfi (      int (*)(long)     ,int ))    (int,...);

//Besser lesbarer Prototype
typedef int (*PAR_FUNC)(long);
int (*fpfi(     PAR_FUNC,int)) (int, ...);
//Sehr gut lesbarer Prototype
typedef int (*PAR_FUNC)(long);
typedef int (*RET_FUNC)(int, ...);
RET_FUNC fpfi(PAR_FUNC,int);
```

7.15 Restrict-Zeiger Type Qualifier

`Syntax: Datentyp * restrict Variablenname`

Mit dem Type Qualifier restrict (seit C99 und nur in C, nicht in C++ enthalten) wird dem Compiler mitgeteilt, dass das Objekt, auf welches der restrict Zeiger zeigt, während der Lebenszeit des restrict Zeigers nur über diesen restrict Zeiger im Zugriff steht.

Ist das Objekt bspw. ein Array und dieses Array wird einer Funktion als restrict Zeiger übergeben, so wird dieses Objekt exklusiv dieser Funktion zugeordnet. In nebenläufigen Threads darf während der Laufzeit dieser Funktion nicht parallel auf das Objekt zugegriffen werden.

Diese Einschränkung ermöglicht es dem Compiler, einen effektiveren Code zu erzeugen (in dem er bspw. den Inhalt des über den Zeigers zugegriffen Speicherbereiches längere Zeit (auch über Sequence Points hinweg) cacht).

Beispiel:

```
void updatePtrs(size_t *restrict ptrA,
                size_t *restrict ptrB,
                size_t *restrict val) {
  *ptrA += *val;
  *ptrB += *val; //Compiler kann den dereferenzierten Wert von val cachen
                 //so dass hier ein Speicherzugriff gespart werden kann
}
```

Viele Standard-C-Library Funktionen nutzen diesen Qualifier:

```
void *memccpy(      void *restrict dest,
              const void *restrict src, int c, size_t n);
//memcpy erwartet ergänzend, dass sich die Speicherbereich
//von src[0..n-1] und dst[0..n-1] nicht überlappen.
```

Sonstiges

- Restrict kann auch auf Elemente einer Struktur / Union angewendet werden, wobei restrict dann nur für die Gültigkeit des dazugehörigen Objektes gilt

7.16 Referenzen

Zeiger sind ein mächtiges, aber auch fehleranfälliges Werkzeug. Zur Reduzierung der Fehlermöglichkeiten wurden in C++ und anderen Sprachen Referenzen eingeführt.

Referenzen sind 'interne Zeiger' auf Variablen. Sie werden genauso verwendet wie gewöhnliche Variablen, verweisen jedoch auf das Objekt, mit dessen Adresse sie initialisiert wurden. Beim Zugriff auf die Referenz ersetzt der Compiler den Zugriff durch die Dereferenzierung.

Dadurch, dass Referenzen mit dem Anlegen initialisiert werden und keine Zeigerarithmetik möglich ist, sind diese sicherer in der Nutzung als Zeiger. Da die Anwendung von Referenzen identisch zur Anwendung von Variablen ist, ist auf den ersten Blick nicht sichtbar, ob es sich um einen 'Call by Reference' oder einen 'Call by Value' Zugriff handelt. Bei Zugriff auf Referenzen wird das Objekt geändert, auf welches es zeigt. Bei Variablenzugriff wird nur die adressierte Variable geändert.

7.16.1 C++ Referenzen

C++ Referenzen entsprechen einem Zeiger, mit dem Unterschied:

- das Referenzen über den & Operator (und nicht dem * wie bei Zeigern) angelegt werden
- diese mit dem Anlegen initialisiert werden müssen, so dass die Referenz immer auf ein gültige Adresse zeigt
- das zum Dereferenzieren kein Dereferenzierungs-Operator '*' notwendig ist
- auf die Speicheradresse des internen Zeigers nicht zugegriffen werden kann
- keine Zeigerarithmetik hiermit getätigt werden kann

Beispiel:

```
int var1=0;
int &ref=var1;  //Definition einer Referenz
ref=1;          //Änderung von var1

int var2=2;
ref=var2;   //Referenz kann nicht neu gesetzt werden
            //Referenz zeigt weiterhin auf var1
ref=3;      //Änderung von var1

void foo(int par,int &ref) {
  par++;
  ref++;    //Änderung von var2
}
var1=var2=0;
foo(var1,var2);
```

7.16.2 Java Referenzen

Wird ein Objekt in Java angelegt, so wird Speicher im Heap reserviert und der Zeiger auf diesen Speicherbereich in einer Referenz gespeichert. In diesem Sinne entsprechen Java

Referenzen den C++-Referenzen.

Wird ein Objekt nicht mehr benötigt, so kann dieses durch 'Löschen' der Referenz gelöscht werden. Der Garbage Collector erkennt, dass auf dem im Heap reservierten Speicherbereich keine Referenz mehr zeigt und gibt diesen Speicherbereich frei. Damit der Heap nicht zu sehr fragmentiert wird (und ggf. weitere Speicheranforderungen im Heap nicht mehr bedienen kann), führt der Garbage Collector ergänzend eine Defragmentierung des Heaps durch. Dabei werden Speicherbereiche verschoben, so dass die entstandenen Lücken geschlossen werden. Mit dem Verschieben müssen dann auch die Referenzen, welche die Startadresse auf diesen Speicherbereich beinhalten, angepasst werden.

Im Unterschied zu C++ können sich die Speicheradressen, auf denen die Java-Referenzen zeigen, durch die virtuelle Maschine geändert werden.

7.17 Umgang mit Zeigern

Zeiger sind ein mächtiges, aber auch fehleranfälliges Werkzeug. Bei sachgemäßer Verwendung bieten Zeiger die Möglichkeit zur Geschwindigkeitssteigerung und zur effektiven Nutzung des Speichers. Auch werden hiermit Features ermöglicht, die sonst nicht darstellbar sind (Zugriff auf Peripherie, dynamische Funktionalitäten). Eine falsche Nutzung von Zeigern führt im Best Case Fall direkt zu einem Programmabsturz. Im Worst Case Fall wird eine Speicherstelle verändert, die erst im späteren Programmablauf sich bemerkbar macht.

7.17.1 Typische Fehler

C/C++ geht immer davon aus, das alle Zeiger auf 'gültige' Speicheradressen zeigen. Dereferenzierungen werden somit ungeprüft ausgeführt.

Bei Zugriff auf ungültige Speicheradressen wird oftmals von einem Bufferoverflow/Stackoverflow/Dangling Zeiger gesprochen.

Bufferoverflow

Von einem Bufferoverflow wird gesprochen, wenn außerhalb des reservierten Speichers eines Arrays oder eines Heap-Bereiches zugegriffen wird.

```
int arr[10];
arr[10]=11;   //Bufferoverflow
int *ptr=(int *)malloc(10);
*(ptr+10)=11;  //Bufferoverflow
```

Stackoverflow

Der von einem Prozess adressierbare Speicher ist begrenzt. Dementsprechend ist auch die max. Größe des Stacks begrenzt. Da für jeden Funktionsaufruf Speicher auf dem Stack reserviert wird, kann insbesondere bei Rekursion der verfügbare Speicher ausgeschöpft

werden. Im Anschluss an den Speicherbereich des Stacks befindet sich zumeist der Speicherbereich des Heaps. Ein Stackoverflow bedeutet somit, dass der Heap überschrieben wird.

Alternativ kann auch das Betriebssystem den Prozess beenden, wenn dieser mehr Stack vom OS anfordert, als das OS diesem Prozess/Thread bereitstellt.

Ergänzend wird mit Stackoverflow oftmals auch interpretiert, dass auf Bereiche außerhalb von lokalen Variablen zugegriffen wird. Im Gutfall wird auf den freien Bereich 'oberhalb' des Stacks zugegriffen. Im Schlechtfall wird auf den Stackbereich der aufrufenden Funktion zugegriffen.

Auf dem Stack wird nicht nur der Speicher für die lokalen Variablen gehalten, sondern auch die Rücksprungadresse zur aufrufenden Funktion. Wird diese aufgrund eines Bufferoverflows beschrieben, so würde mit Beenden der Funktion die Programmausführung an einer nicht definierten Stelle fortgesetzt werden. Virenhersteller nutzen diese Möglichkeit gerne aus, um Schadcode auf dem System zur Ausführung zu bringen.

Dangling Zeiger

Von einem Dangling Zeiger spricht man, wenn ein Zeiger auf eine Variable zeigt, die nicht mehr gültig ist:

```
int *ptr;
{
  int var;
  ptr=&var;
}
*ptr=4711; //Adresse des Zeigers nicht mehr gültig

ptr=(int *)malloc(100*sizeof(int));
free(ptr);
ptr[0]=4711;  //Adress des Zeigers nicht mehr gültig
```

7.17.2 Best Practice

- Code Review durch eine andere Person
- Zeiger müssen vor der Dereferenzierung mit einer gültigen Speicheradresse initialisiert werden
- Gültigkeitsbereich des Speichers, auf den der Zeiger zeigt, kontrollieren
- Funktionen, welche Zeiger zurückgeben, geben im Fehlerfall den NULL-Zeiger zurück. Der Rückgabewert von Funktionen ist zwingend zu kontrollieren
- Compilerwarnungen hinsichtlich Zeiger sind als Fehler anzusehen. Ggf. Compiler auf max. Warning Level stellen (-Wall)
- 'Googlen' hilft bei Problemen im Umgang mit Zeigern nicht weiter, da zumeist die 'anderen' ebenfalls Zeiger nicht verstanden haben und eher den Fehler mit einem Workaround umgehen
- Zeiger auf Zeiger sind schwer verständlich. Diese sind daher zu vermeiden, resp. nur dort einzusetzen, wo sie unbedingt benötigt werden.
- Entsprechend Referenzen Zeiger als const-Zeiger anlegen

- Zeiger auf NULL setzen, wenn deren Speicheradresse, auf welche diese zeigt, nicht mehr gültig ist

```
#define SETNULL(ptr) ({typeof(ptr)dummy=ptr; ptr=NULL; dummy;})
char *ptr=malloc(100);
strcpy(ptr,"hello world");
free(SETNULL(ptr));
```

8 Array

Der Datentyp Array ist einer der ältesten Datentypen. Die Aufgabe der ersten Computer war oftmals, stupide mathematische Berechnungen mit unterschiedlichen Eingabewerte zu berechnen (bspw. Berechnung von ballistischen Tabellen durch die ENIAC ENIAC[1]). Sowohl die Eingabedaten als auch die Ausgabedaten wurden hintereinander im Speicher (in Arrays) gespeichert. Zur einfacheren Adressierung der Elemente in den Arrays wurden u.A. die Index-Register eingeführt. Mit der Entwicklung der ersten höheren Programmiersprachen wurde dieses Konzept im Datentyp Array abgebildet.

8.1 Grundlagen

Arrays (`Syntax: Datentyp [size_t DIM1][size_t DIM2]`$_{\text{OPT}}$...) entsprechen Variablen, wobei anstatt Speicherplatz nur für ein Element Speicherplatz für die Aufnahme 'aller' Elemente reserviert wird (Speicherplatzgröße ergibt sich aus der Multiplikationen aller Dimensionen DIM_x multipliziert mit der Größe des Datentyps).

Bei eindimensionalen Arrays werden die Elemente hintereinander im Speicher angeordnet:

```
//Eindimensional    //Anordnung im Speicher
short arr1[10];     //arr1[0] arr1[1]…arr1[9]
```

Mehrdimensionale Arrays entsprechen einem Array eines Arrays (und nicht einem Zeiger auf einen Zeiger, wie es fälschlicherweise oft gesagt wird):

```
int matrix[12][10];

//kann wie folgt dargestellt werden
typedef int vector[10];  //Alias für ein Int-Array bestehend aus 10 Elementen
vector matrix1[12];      //Eindimensionales Array des Datentyps vector
                         //und damit ein eindimensionales Arrays eines
                         //eindimensionalen Arrays -> Zweidimensionales Array
//matrix1[0..11]           Dereferenzierter Datentyp -> vector resp. int(*)
//matrix1[0..11][0..9]     Dereferenzierter Datentyp -> int
```

Bei mehrdimensionalen Arrays erfolgt die Anordnung der Elemente im Speicher beginnend vom rechten Index:

```
//Zweidimensional   //Anordnung im Speicher
char arr2[4][3];     //arr2[0][0]  arr2[0][1]  arr2[0][2]  arr2[1][0]
                     //arr2[1][1]  arr2[1][2]  arr2[2][0]  …. arr2[3][2]
```

Der Arrayname zeigt auf die Startadresse des ersten Elementes des Arrays. Aus Datentypsicht ist der Arrayname ein Zeiger auf ein Array (`Syntax: Datentyp (*) [DIM1]…`).

1 https://de.wikipedia.org/wiki/ENIAC

Die Dereferenzierung dieses Zeigers erfolgt über []-Klammern. Da die Elemente des Arrays strukturiert im Speicher angeordnet sind, bedeutet ein Zugriff auf ein Array-Element der Zugriff auf einen Speicherbereich, dessen Adresse sich aus der Startadresse ergänzt um den aus den Indexen ergebenden Offset ergibt.

Bei einem eindimensionalen Array sieht die Berechnung wie folgt aus:

```
short arr1[10];
arr1[index] = 2; //          *(        arr1 +  index                 )=2;
arr1[index] = 2; // *(short *)((char *)arr1 + (index * sizeof(short)))=2;
```

Den Zugriff auf ein Element des zweidimensionalen Arrays setzt der Compiler in folgende Rechenvorschrift um:

```
int arr2[12][10]
arr2[idx1][idx2]=2; // *       (        arr2 + idx1 * 10   + idx2    )=2;
                    // *(int *)((char *)arr2 + idx1 * 10*4 + idx2 * 4)=2;
```

Allgemein gilt folgende Rechenvorschrift für den Zugriff auf ein Array Element:

```
int arr3...[DIM3][DIM2][DIM1];
arr3...[i3][i2][i1]
//*      (        arr4 +  i1+(i2*DIM1)+(i3*DIM1*DIM2)+...          )
//*(int*)((char *)arr4 + (i1+(i2*DIM1)+(i3*DIM1*DIM2)+...)*sizeof(int))
```

Zur Berechnung des Offsets nutzt der Compiler die Dimensionsangaben und den Datentyp des Arrays **aus der Definition der Variable. Die Dimension des Arrays wird nicht gespeichert.** Diese ist somit nur zur Compilezeit dem Compiler bekannt.

Da mit dem Arraynamen nur ein Zeiger auf ein Array in die 'Hand' genommen wird, kann ein Array nie als 'Ganzes', also per Inhalt übergeben/zugewiesen werden. Eine Zuweisung (incl. Parameterübergabe) entspricht nur der Zuweisung/Kopieren des Zeigers auf ein Array (soll der Inhalt kopiert werden, so muss dies händisch getätigt werden (siehe Kopieren von Arrays[2])). Da die Dimension nicht gespeichert wird (und damit auch nicht kopiert wird), muss die Zielvariable ein Zeiger auf ein Array mit dem identischen Datentyp und der identischen Dimension wie das Quellarray sein. Fehlen die Dimensionsangabe (was alternativ möglich ist), so können die Array-Elemente aufgrund der nicht möglichen Offset-Berechnung nicht angesprochen/dereferenziert werden:

```
//Übergabeparameter arr als Beispiel für die Zuweisung
void func1(int arr[5][3])  //Array-Datentypbeschreibung im Funktionskopf
                           //beschreibt einen Zeiger auf ein Array, so
                           //dass hier nur die Startadresse übergeben wird
{                          //Dimensionsgabe zwingend notwendig
  arr[1][2]=12;
    //Zuweisung ändert Inhalt der Aufrufervariable! (Call-by-Reference)
}
int main(void) {
  int arr1[5][3];  //Definition eines Arrays
  func1(arr1);     //Zuweisung von arr1 an die Funktionsvariable arr
                   //in Form der Zuweisung des Zeiger auf ein Array
  return 0;
}
```

Hinweise:

2 Kapitel 8.13 auf Seite 196

- Die linke Dimensionsangabe ist nur für die Berechnung der tatsächlichen Speichergröße durch den Compiler notwendig, jedoch nicht für die Offsetberechnung beim Zugriff auf einen Eintrag nötig. Bei der Definition von Zeiger auf Arrays kann die Angabe dieser Dimension entfallen:

```
void func2(int arr[][5]) { //Linke Dimensionsangabe nicht nötig
  arr[1][1]=3;
}
void func1(int arr[]) {  //Linke Dimensionsangabe nicht nötig
   arr[4]=3;
}
int main(void) {
  int arr1[5];
  func1(arr1);
  int arr2[3][5];
  func2(arr2);
}
```

- Die Dimension eines Arrays ergibt sich aus der Definition des Arrays resp. Zeiger auf Array. Nach Ausführung der Definition kann die Dimension nicht mehr geändert werden

8.2 Arrays von ...

Arrays können von allen Datentypen erstellt werden:

- Array von Strukturen und Strukturelementen

```
struct xyz{
  int x;
  int y;
  int array[10];   //Array eines Strukturelement
}  arr3[10];       //Array der Struktur
arr3[3].x          //Zugriff auf ein Strukturelement des Arrays
arr3[4].array[5]   //Zugriff auf ein Array-Strukturelement des Arrays
```

- Array von Zeigern

```
const char *arr4[]={"hello","world"};    //Eindimensionales Array aus Zeigern
//Entspricht
char const * const anonymous1="hello";
char const * const anonymous2="World";
const char * arr5[2] = {anonymous1,anonymous2};
printf("1. String:  %s",arr5[0]);
printf("5. Zeichen des 2.Strings: %c",arr5[1][4]);
```

Der Datentyp eines Zeigers auf ein eindimensionales Array `(T (*))` entspricht einem 'normalen' Zeiger `(T *)`,(siehe Array Datentypen[3]), so dass dann der Rückgabewert des Arrays auf Char-Zeiger über [] dereferenziert werden kann:

3 Kapitel 8.3.1 auf Seite 184

```
arr5[1][4] == 'd'
------
   +--> Zugriff auf das 2. Element, welcher einen Zeiger auf char *
        liefert. Dieser Zeiger zeigt auf die Startadresse von "World"
       [4]
        +--> Vom zurückgelieferten Zeiger wird das wird 4. Element
             dereferenziert 'd'
```

Anwendung:

1) Übergabeparameter argv

```
int main(int argc,char *argv[])
//argv ist ein Array von Zeigern, in welches die Laufzeitumgebung von C
//bei Programmstart die Adressen der einzelnen Übergabewerte einträgt
//bspw: >>./a.out para1 para2=47
//        |       |     +--> argv[2]
//        |       +--------> argv[1]
//        +----------------> argv[0]
```

2) Enum in einen String wandeln

```
typedef enum {OK,ERROR,WARNING} status_e;
status_e ret=ERROR;
const char *status2str[]= {"Status:OK","Status:Error","Status:Warning"};
printf("%d %s",ret,status2str[ret]);
```

8.3 Sichtweise auf Arrays auf Basis des Datentyps

Der Datentype eines eindimensionalen Arrays ist wie folgt definiert:

```
Datentype: T[DIM1]
```

Der Variablenname des Arrays ist ein Zeiger auf ein 'eindimensionales' Array:

```
Datentyp:  T(*)  //entspricht einen normalen
Datentyp:  T *   //Zeiger vom Datentyp
```

Mehrdimensionale Arrays sind Arrays von Arrays:

```
Datentype: T[DIM1][DIM2][DIM3]…
```

Der Variablenname des Arrays ist ein Zeiger auf ein Array von Arrays:

```
Datentype: T(*)[DIM2][DIM3]...   //Angabe der 'linken' Dimension nicht
                                 //möglich/nötig. Das '*' kann als
                                 //Ersatz für die fehlende linke Dimension
                                 //verstanden werden.
//Hinweis:
```

```
//Klammerung des '*' zwingend notwendig, andernfalls würde der '*'
//zum Datentyp T zugeordnet sein, welches ein Zeiger von diesem Datentyp
//erzeugt.
```

Bei der Dereferenzierung von Zeigen wird entsprechend Zeiger:Sichtweise von Zeiger auf Basis des Datentyps [4] ein * vom Datentyp weggenommen:

```
int var;           //(int)
int *ptr1=&var;    //(int *)
    *ptr1          //*(int *) → (int)
```

Die []-Klammern beim Arrayzugriff entspricht der Dereferenzierung. Mit jeder []-Klammer erfolgt eine Dereferenzierung, welche anstatt der Wegnahme eines * die Wegnahme der **linken** []-Klammer vom Datentyp entspricht:

```
//Code              → Datentyp
int array[5][4][3]; → (int[DIM1][DIM2][DIM3])
array[1]            → (int[DIM1][DIM2][DIM3])[]     → (int[DIM2][DIM3])
array[1][2]         → (int[DIM1][DIM2][DIM3])[][]   → (int[DIM3])
array[1][2][0]      → (int[DIM1][DIM2][DIM3])[][][] → (int)
```

Hieraus ergibt sich, dass ein mehrdimensionales Array teildereferenziert werden kann. Alternativ ausgedrückt bedeutet dies, dass ein mehrdimensionales Array ein Array von Arrays (von Arrays...) ist welches mit jeder Dereferenzierung das übergeordnete 'Array von' abspaltet. Das teildereferenzierte Array ist wie der Arrayname ein Zeiger auf ein Array, welcher in einem Zeiger auf Array gespeichert werden kann:

```
int array[5][4][3];
int (*parr1)[4][3]=array;
int (*parr2)[3]   =array[1];
int (*parr3)      =array[1][2];
int   value       =array[1][2][0];
```

Auch auf einen Zeiger von Arrays kann über die []-Klammern auf die Elemente zugegriffen werden. Mit jeder []-Klammer erfolgt auch hier eine Wegnahme der linken []-Klammer vom Datentyp. Beim allerletzten Zugriff wird anstatt der [] das Sternchen weggenommen:

```
//Code              → Datentyp
int *parr[4][3];    → (int (*)[DIM2][DIM3])
parr[1]             → (int (*)[DIM2][DIM3])[]      → (int (*)[DIM3])
parr[1][2]          → (int (*)[DIM2][DIM3])[][]    → (int (*))
parr[1][2][3]       → (int (*)[DIM2][DIM3])[][][]  → (int)
```

Ein Zeiger auf ein eindimensionales Array entspricht einem normalen Zeiger. Beide können demnach wie folgt dereferenziert werden:

```
int arr[4];
int (*parr)=arr;
int  *ptr  =arr;
//Variante 1: Dereferenzierung über []
parr[2]=1;
ptr[2] =1;
//Variante 2: Dereferenzierun über *
*(ptr+1)=1;
*(parr+1)=1;
```

4 Kapitel 7.5 auf Seite 147

8.3.1 Zusammenfassung

Mit der Definition eines Arrays wird Speicher entsprechend der Dimension des Arrays reserviert:

```
//Code                  → Datentyp
int array[5][4][3][2];  → (int [5][4][3][2])
```

Vollständige Dereferenzierung/Indizierung gibt den zugrundeliegenden Datentyp zurück:

```
//Code                → Datentyp
array[3][2][1][0]     → (int)
```

Eine Teildereferenzierung/Indizierung gibt ein 'SubArray' zurück, also ein Zeiger auf Arrays von Array von ..., wobei vom ursprünglichen Array der Index beginnend von links entfällt:

```
//Code            → Datentyp                  Dimensionen
array[1]          → (int[4][3][2]) entspricht array[1][0..3][0..2][0..1]
array[2][1]       → (int   [3][2]) entspricht array[2][ 1  ][0..2][0..1]
array[2][1][0]    → (int      [2]) entspricht array[2][ 1  ][ 0  ][0..1]
array[2][1][0][0] → (int         ) entspricht array[2][ 1  ][ 0  ][ 0  ]
```

Mittels dem Datentype Zeiger auf ein Array (Datentype: T(*)[]..) kann ein 'Zeiger' auf ein Array gespeichert werden. Der Compiler benötigt für die Adressberechnung nicht die linke Dimensionsangabe, so dass das Sternchen quasi der linken fehlenden []-Klammer entspricht:

```
int arr1[4];
int arr2[4][5];
int arr3[4][5][6];
int arr4[4][5][6][7];
int (*parr1)          = arr1;
int (*parr2)[5]       = arr2;
int (*parr3)[5][6]    = arr3;
int (*parr4)[5][6][7] = arr4;
//Alternativ kann in einem Zeiger auf ein Array auch ein
//Teildereferenziertes Array gespeichert werden:
int array[5][4][3][2];
int (*sub0)          = array[2][1][0];
int (*sub1)[2]       = array[2][1];
int (*sub2)[3][2]    = array[1];
int (*sub3)[4][3][2] = array;
```

Der Zugriff auf die Arrayelemente über einen Zeiger auf ein Array erfolgt wie bei einem Array:

```
//Code                          → Datentyp
sub0[0..1]=1;                   → array[  2 ][  1 ][  0 ][0..1]=1;
sub1[0..2][0..1]=1;             → array[  2 ][  1 ][0..2][0..1]=1;
sub2[0..3][0..2][0..1]=1;       → array[  2 ][0..3][0..2][0..1]=1;
sub3[0..4][0..3][0..2][0..1]=1; → array[0..4][0..3][0..2][0..1]=1;
```

8.3.2 Explizites Cast

Über explizite Typkonvertierung (Cast Operator) kann ein Array in ein anderes Array 'konvertiert' werden. Dabei kann sowohl der zugrundeliegende Datentyp als auch die Dimension neu gesetzt werden:

```
int arr[5][4][3];
char  (*ptr1)[4][3]=arr;  //KO, Datentyp von arr und ptr1 unterschiedlich
char  (*ptr2)[4][3]=(char (*)[4][3])arr;  //Cast auf einen anderen Datentyp
```

```
int   (*ptr3)[3][4]=(int  (*)[3][4])arr;  //Cast auf eine andere Dimension
float (*ptr4)[3]   =(float(*)[3])arr;     //Ohne Worte
```

Der Cast bewirkt keine Umsortierung der Daten im Speicher, sondern einzig einer Umwidmung. Der Cast wirkt sich einzig auf die Offsetberechnung bei der Dereferenzierung aus:

```
int arr[5][4][3][2];
arr[1][3][2][1]=1;   //*(arr+1*(4*3*2)+3*(3*2)+2*(2)+1)=1;
                     //*(arr+   24    +   18  +   4 +1)=*(arr+47)=1;

int (*parr)[3][4][2]=(int (*)[3][4][2])arr;
parr[1][2][3][1]=1;  //*(parr+1*(3*4*2)+2*(4*2)+3*(2)+1)=1;
                     //*(parr+   24    +   16  +   6 +1)=*(parr+47)=1;
```

8.3.3 Variable Length Array (VLA)

Im Normalfall ergibt sich die Dimensionsangabe eines Arrays aus einer Konstanten, so dass der Compiler zur Compilezeit passend Speicher reserviert. Bei Variable Lenght Array wird die Arraydimension nicht über eine Konstante, sondern über eine Variable oder einen Ausdruck angegeben, so dass sich tatsächliche Dimension des Arrays erst zum Ausführungszeitpunkt der Definition ergibt. Da Variablen / Ausdrücke erst zur Laufzeit ausgewertet werden, können VLAs nur als lokale Variable oder als Übergabeparameter definiert werden:

```
void foo(int dim1, int dim2) {
  int arr[dim1][dim2+2*dim1];
void far(int dim1, int dim2, int arr[dim1][dim1+dim2]) {
```

Auch bei Zeigern auf Arrays kann die Dimension über eine Variable / einen Ausdruck angegeben werden:

```
void foo(int dim1, int dim2) {
  int (*parr)[dim1][dim2];
```

8.3.4 Bedenke

Bei Zeiger auf Arrays muss das * geklammert sein. Andernfalls wirkt sich das * auf den zugrundeliegenden Datentyp aus und erzeugt aus diesem einen Zeiger:

```
int  array1[3][2];   //Hiermit wird Speicher für ein zweidimensioanles Array
                     //vom Datentyp int reserviert
int *array2[3][2];   //Hiermit wird Speicher für ein zweidimensionales Array
                     //vom Datentyp int * reserviert
int  (*ptr3)[3][2];  //Hiermit wird Speicher für eine Zeigervariable
                     //reserviert, welches auf ein Dreidimensionales Array
                     //vom Typ int zeigt
int *(*ptr4)[3][2];  //ohne Worte
```

8.4 Sichtweise auf Array auf Basis eines Speicherabzuges

Über die Darstellung, wie ein Array im Speicher organisiert ist, lässt sich die Arbeitsweise von Arrays, Zeiger auf Arrays und der Dereferenzierung alternativ beschreiben.

8.4.1 Zeiger auf Arrays und Dereferenzierung

Wie mehrfach beschrieben werden die einzelnen Array-Elemente vom rechten Index hochgezählt im Speicher abgelegt:

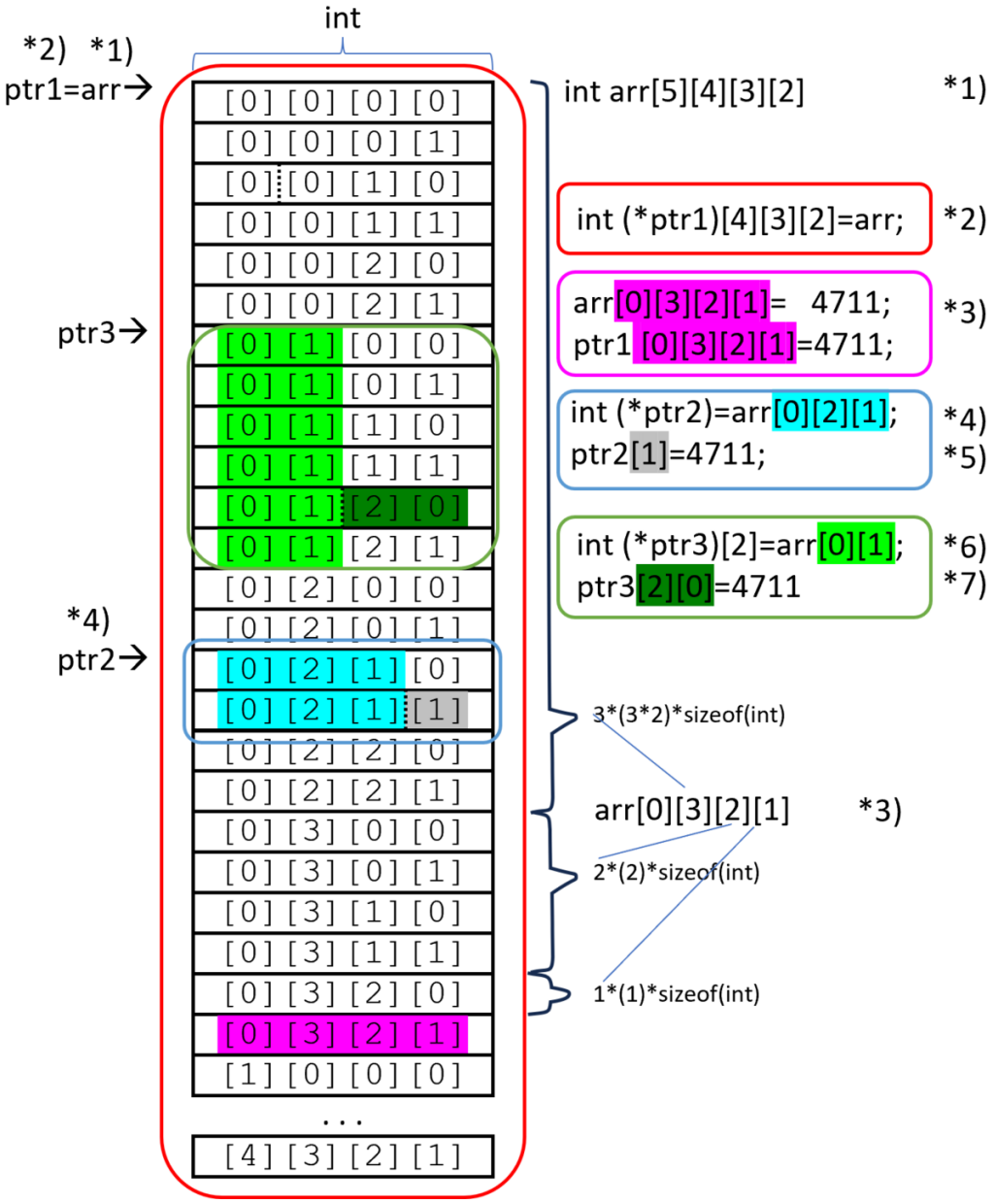

Abb. 12 Sichtweise auf Arrays auf Basis eines Speicherabzuges

*1) Der Arrayname ist ein Zeiger auf ein Array. Er zeigt auf die erste Speicheradresse (= Adresse des ersten Elementes) des vom Array belegten Speicherplatzes

*2) Der Arrayname kann in einem Zeiger auf ein Array gespeichert werden. Der Zeiger zeigt dann ebenfalls auf die erste Speicheradresse des vom Arrays belegten Speicherplatzes

3) Wird das Array 'arr' volldereferenziert, wird ein Element des zugrundeliegenden Datentyps des Arrays angesprochen. Die Adresse des Elementes ergibt sich aus der Startadresse des Arrays plus einen Offset (hier 0(4*3*2)+3*(3*2)+2*(2)+1)

4) Wird ein Array nur Teildereferenziert, so ist der Rückgabewert ein Zeiger auf ein Array. Die Adresse des Zeigers in das Array ergibt sich aus der Startadresse und den Offset (hier 0(4*3*2)+2*(3*2)+1*(2)+(entfällt) =14). Die Anzahl der Elemente, auf welches dieses teildereferenzierte Array zeigt, ergibt sich aus der verbleibenden Dimension (hier [2])

*5) Wird ein Zeiger vollständig dereferenziert (hier auf Basis des zuvor teilderefenzierten Zeigers auf ein Array), so wird auch hier ein Element des zugrundeliegenden Datentyps angesprochen. Die Adresse ergibt sich aus der Startadresse des Zeigers plus einen Offset, ermittelt auf Basis der Dimension des Zeiger auf Arrays (hier 1)

6) Die Startadresse des teildereferenzierten Arrays ergibt sich aus (0(4*3*2)+1*(3*2)+(entfällt)=6). Die Dimension des teildereferenzierten Arrays ist [3][2]

7) Der Offset beim Zugriff ergibt sich zu (2(2)+0)=2

8.4.2 Offsetberechnung

Bei einem Zugriff auf ein Arrayelement rechnet der Compiler die Indexe in einen Offset um. Hierzu nutzt er die Dimensionsangabe und den Datentyp des Arrays / Zeiger auf Array aus der Definition der Variable. Eine Überprüfung, ob der Index im gültigen Wertebereich ist, findet nicht statt:

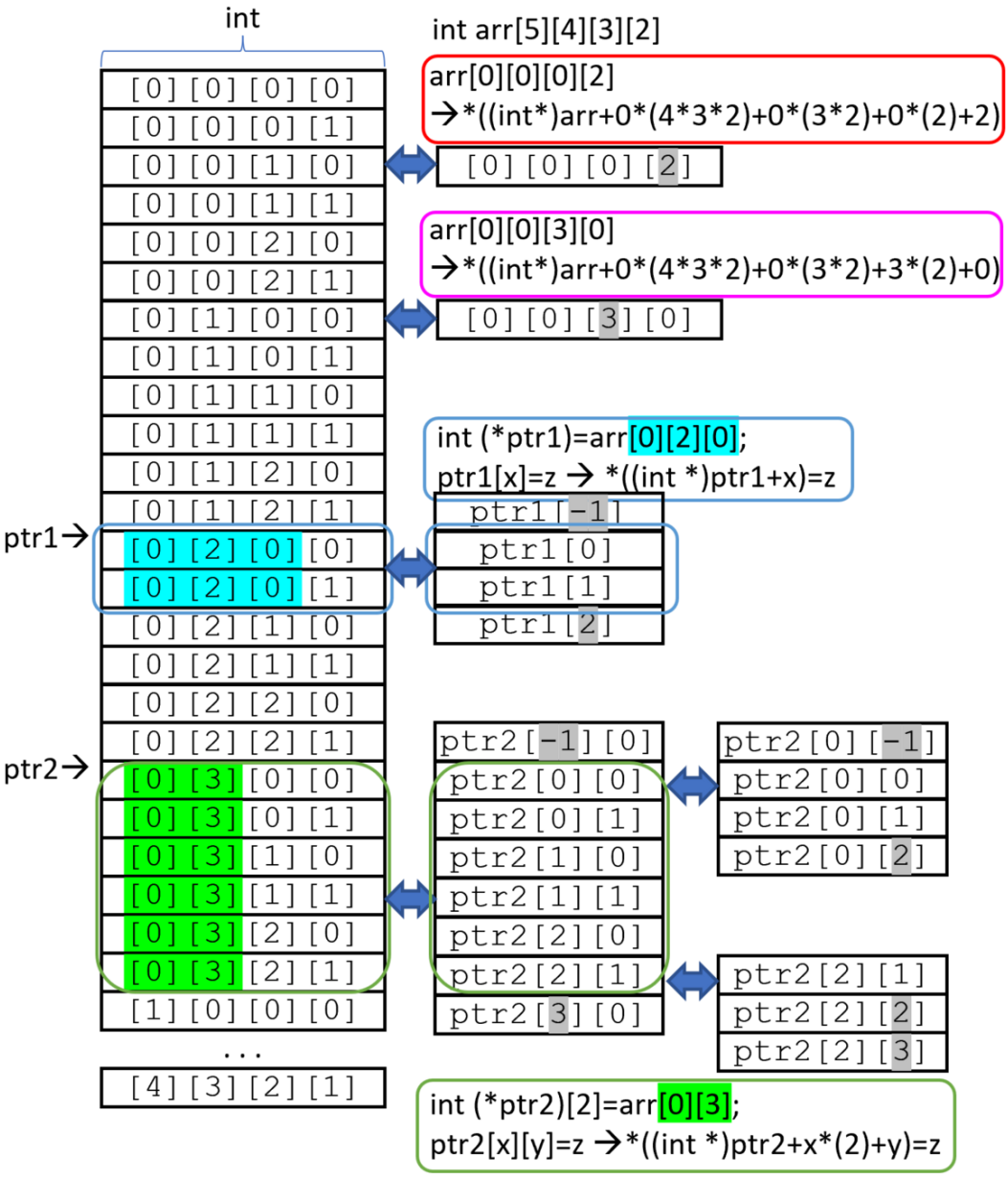

Abb. 13 Sichtweise auf Arrays auf Basis eines Speicherabzuges mit Darstellung der Offsetberechnung

8.4.3 Explizites Cast

Bei der 'Umwidmung' eines Arrays in einen anderen Arraytyp erfolgt keine Umsortierung der Daten:

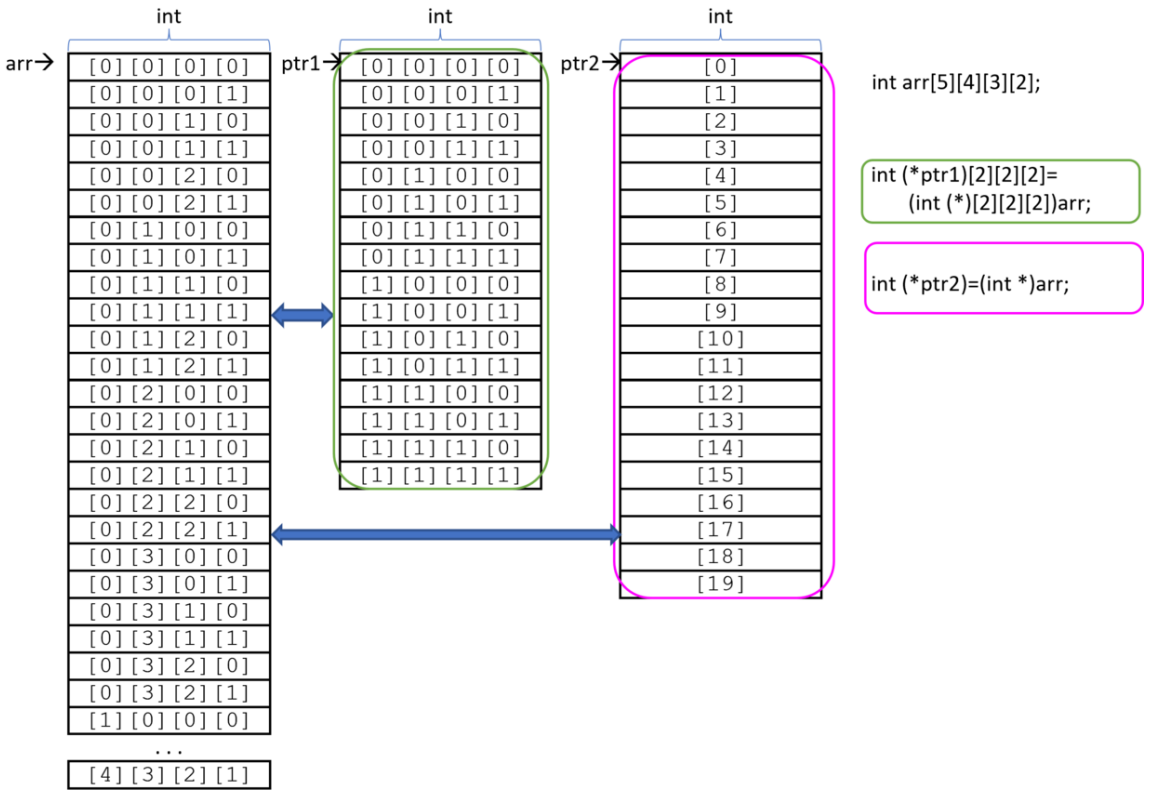

Abb. 14 Sichtweise auf Arrays auf Basis eines Speicherabzuges unter Anwendung einer Typkonvertierung

8.5 Arrays vs. Zeiger auf Zeiger

Mehrdimensionale Arrays sind Arrays von Array. Sie beinhalten Speicher zur Aufnahme aller Elemente des zugrundeliegenden Datentyps. Zeiger auf Arrays von Arrays enthalten einzig die Startadresse auf solche Arrays. Über die bei der Definition angegebenen Dimension bei Array und Zeiger auf Arrays erfolgt die Offset-Berechnung zum Zugriff auf die einzelnen Elemente.

Bei Zeiger auf Zeiger wird Speicherplatz zur Aufnahme eines Zeigers für jeden Subzeiger reserviert. Typischerweise sind diese Zeiger verkettet, so dass ein Zeiger auf die Startadresse der nächsten Zeigervariablen zeigt.

Auch wenn die Zugriffsart bei einem zweidimensionale Array und einen Array von Zeigern identisch ist, so bewirken die beiden Datenstrukturen unterschiedlichen Zugriffsmechanismen. Bei einem zweidimensionalen Array ergibt sich der Zugriff über eine einfache Offsetberechnung.

Bei einem Zeiger auf Zeiger (hier zaz) muss zunächst der Inhalt der Variablen zaz gelesen werden. Dieser zeigt auf die Startadresse des eindimensionalen Arrays. Über den Offset (hier 2*sizeof(int *)) kann nun die Startadresse eines weiteren eindimensionalen Arrays gelesen werden. Das Zielelement ergibt sich aus dieser Startadresse addiert um einen weiten Offset.

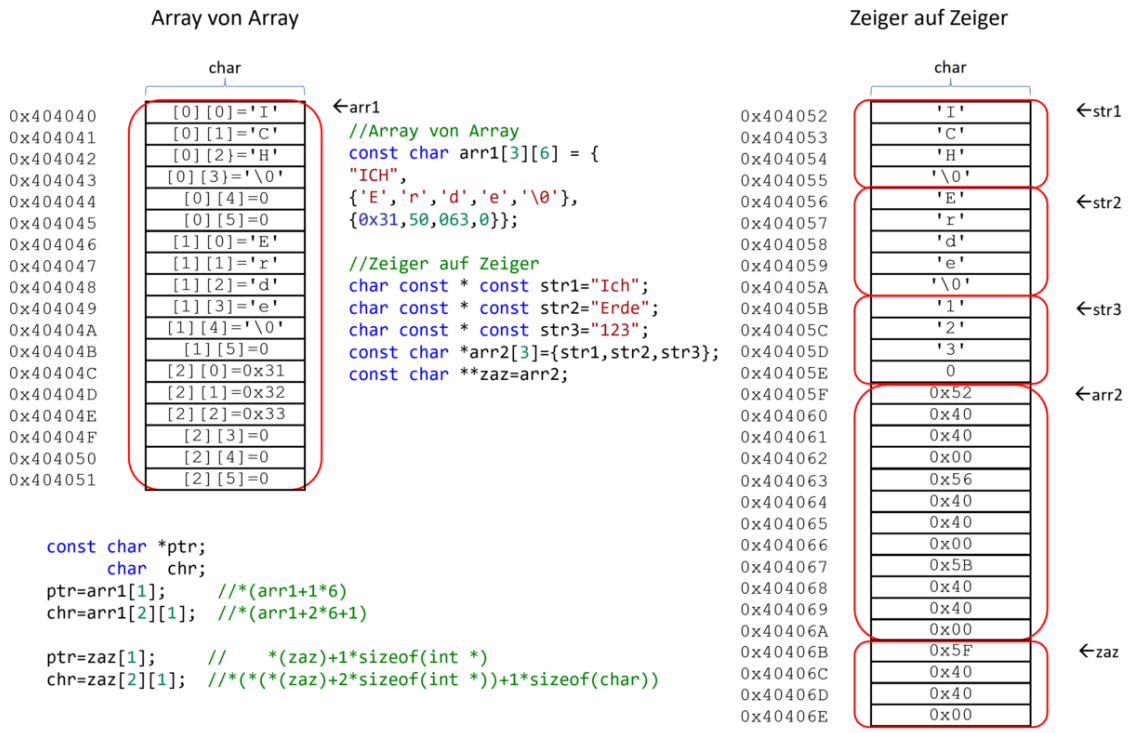

Abb. 15 Vergleich von Arrays und Zeiger auf Zeiger auf Basis eines Speicherabzuges

8.6 Speicherplatzreservierung

Der Speicherbedarf eines Arrays ergibt sich aus der Multiplikation der Dimensionen und des zugrundeliegenden Datentypes.

Bei globalen und static lokalen Arrays erfolgt die Speicherplatzreservierung für Arrays durch den Compiler, d.h. die Dimensionsangabe der Arrays muss über Konstanten/Konstanteausdrücke erfolgen.

Bei lokalen Arrays erfolgt die Speicherplatzreservierung zur Laufzeit, wenn die Definition des Arrays zur 'Ausführung' kommt. Die Dimension des Arrays kann dementsprechend über Konstanten/Kontantenausdrücke und bei C (nicht C++) über Variablen und Ausdrücke (Variable Length Arrays VLA) (siehe Kap. 5.15 Variable Length Array (VLA)) angegeben werden:

```
int var1=7;
short arr[var1];   //Speicherplatzreservierung entsprechend dem aktuellen
                   //Wert von var1
var1=8;            //Nachträgliches ändern von var1 ändert nicht die
                   //Dimension des Arrays!
arr[7]=4711;       //KO Bufferoverflow, da mit Hochsetzen von var1
                   //nicht das Array vergrößert wird
```

Die Dimension von in Strukturen/Unions eingebetteten Arrays muss ebenfalls über Konstanten/Literale definiert werden. GNU-C erlaubt ergänzend, dass die Dimension von Arrays innerhalb von in Funktionen definierten Strukturen ebenfalls variabel sein darf:

```
struct xyz{
  int x;
  int y[10];    //OK
  int z[x];     //KO, Dimension muss sich aus einer Konstanten ergeben
};

void foo(int n) {
  int arr[n];     //OK, VLA als lokale Variable möglich
  struct xyz{     //Datentypdefinition innerhalb einer Funktion
    int x;
    int y[10];    //OK
    int z[n];     //KO, Dimension muss sich aus einer Konstanten ergeben
                  //Ausnahme: GNU-C
  } var1;    //GNU-C Dimension des Arrays z ergibt such aus den Wert n
             //      zu diesem Ausführungszeitpunkt
}
```

Alternativ kann Speicherplatz für Arrays auf dem Heap reserviert werden:

```
short (*ptr1)[3][2]=(short(*)[3][2])malloc(sizeof(short)*2*3*4711);
if(ptr1!=NULL)
  ptr1[4700][2][1]=11;
```

Damit das Array mit 0 gefüllt ist, empfiehlt sich die Nutzung von calloc:

```
short (*ptr2)[3][2]=(short(*)[3][2])calloc(sizeof(sort),2*3*4711);
```

8.7 Sizeof

Der sizeof-Operator liefert bei Arrays den tatsächlichen Speicherbedarf des Arrays in Bytes zurück:

- Bei Arrays mit konstanter Dimensionsangabe wird sizeof durch den Compiler ersetzt, sizeof entspricht folglich einem Konstantenausdruck
- Bei VLA kann der sizeof-Operator nicht durch den Compiler ersetzt werden. Hier wird der sizeof-Operator zur Laufzeit ausgewertet (d.h. die Größe des Arrays wird bei VLA's ergänzend gespeichert)

Durch Ausnutzung der Tatsache, dass der sizeof-Operator auch auf teildereferenzierte Arrays angewendet werden kann, kann über diesen Weg die Dimension des Arrays ermittelt werden:

```
int arr1[7];
size_t dim1_arr1=sizeof(arr1)/sizeof(arr1[0]);

short arr2[3][4];
size_t dim1_arr2=sizeof(arr2)/sizeof(arr2[0]);
size_t dim2_arr2=sizeof(arr2[0])/sizeof(arr2[0][0]);

double arr3[3][4][5];
size_t dim1_arr3=sizeof(arr3)/sizeof(arr3[0]);
size_t dim2_arr3=sizeof(arr3[0])/sizeof(arr3[0][0]);
size_t dim3_arr3=sizeof(arr3[0][0])/sizeof(arr3[0][0][0]);
```

Für einen kürzeren und besser lesbaren Code empfiehlt sich, die Dimension über Makros zu setzen:

```
#define ARR1_DIM1 10
#define ARR1_DIM2 12
float arr1[ARR1_DIM1][ARR1_DIM2];
```

Bei Anwendung des sizeof-Operators auf Zeiger auf Arrays wird nicht der Speicherbedarf des Arrays, sondern wie bei Zeigern üblich der Speicherbedarf des Zeigers zurückgegeben:

```
short arr[4][6];
short (*ptr)[6]=arr;
printf("%zd\n",sizeof(arr));  //48=4*6*sizeof(short)
printf("%zd\n",sizeof(ptr));  //4/8

//Vorsicht, wird der Zeiger auf ein Array teildereferenziert, so
//wird auch hier die Größe des teildereferenzierten Arrays zurückgegeben
printf("%zd\n",sizeof(ptr[0]));     //12 = 6*sizeof(short)
printf("%zd\n",sizeof(ptr[0][0]));  //2  =   sizeof(short)
```

8.8 Initialisierung

Ein Array wird über ein Initialisierungsliste (siehe Grundlagen:Initialisierungsliste / Compound Literal[5]) initialisiert, d.h. die Initialisierungswerte stehen in geschweiften Klammern. Für jede Dimension bei einem mehrdimensionalen Array ist eine innere Initialisierungsliste zu nutzen:

```
int arr1[3]={        //Initialisierungsliste
    1,2,3            //Initialisierungswerte
};
int arr2[3][5][3]= { //Initialisierungsliste
  {                  //Initialisierungsliste für linke Array-Dimension
    {                //Initialisierungsliste für mittlere Array-Dimension
      1,2,3          //Initialisierungswerte
    },
    {                //Initialisierungsliste für mittlere Array-Dimension
      ...            //Initialisierungswerte
    },
    ...
  },
  {                  //Initialisierungsliste für linke Array-Dimension
  },
  ...
};                   //Ende der Initialisierungsliste

struct xyz {int x,y[3],z;};
struct xyz arr[2][2]= {  //Initialisierungsliste
  {              //Initialisierungsliste für linke Array-Dimension
    {            //Initialisierungsliste für Struktur
      1,         //Strukturelement x
      {          //Strukturelement y Initialisierungsliste
        2,3,4
      },
      5          //Strukturelement z
    },
    {            //Initialisierungsliste für Struktur
    }
  },
  {              //Initialisierungsliste für linke Array-Dimension
  }
};
```

5 Kapitel 4.11 auf Seite 58

Ergibt sich aus den Initialisierungswerten eine Dimensionsangabe, so kann die Angabe der Dimension bei der Definition des Arrays entfallen:

```
long long vector1[4]={1LL,2LL,3LL,4LL};
long long vector2[] ={7LL,8LL,9LL,10LL};

short arr1[4]={1,2,3};   //Nicht initialisierte Elemente werden mit 0 gefüllt
short arr2[2]={1,2,3};   //KO Mehr Initialisierungswerte als Dimension
short arr3[4] = {0};     //Alle Elemente mit 0 füllen
```

Werden bei expliziter Dimensionsangabe:

- weniger Initialisierungswerte angegeben, so werden die verbleibenden Arrayelemente mit 0 gefüllt
- mehr Initialisierungswerte angegeben, so meldet der Compiler einen Fehler

Erstere Variante bietet die Möglichkeit unter Angabe von { 0 } das gesamte Array auf 0 zu setzen. In C++, bei GNU-C und ab C23 kann die Angabe der 0 in der geschweiften Klammer entfallen:

```
char arr1[5] = {0};   //Gesamtes Array wird mit 0 initialisiert
char arr2[5] = { };   //Alternative Möglichkeit
struct xyz1{int x,y,z;} arr3[10]={0};  //Dito
struct xyz2{int x,y,z;} arr4[10]={ };  //Dito
```

Character Arrays können alternativ mit einem String " " initialisiert werden:

```
char string0[5]={'4','7','1','1','\0'};
char string1[5]="4711";
wchar_t arr2g[]=L"hallo";   //Im Falle eines WideChar Strings
int string2[]="hallo";      //KO, String bedingt char als Datentyp
```

Ist bei expliziter Dimensionsangabe des Arrays kein Speicherplatz für das Stringendezeichen vorgesehen, so wird dieses auch nicht gespeichert:

```
char string2[4]="4711";   //Kein Stringendezeichen in string2 enthalten
```

Eine Initialisierung ist nur bei Arrays möglich, deren Dimension über eine Konstante gesetzt wurde. VLA's können nicht initialisiert werden. Außerhalb der C-Spec bieten einige Compiler die Möglichkeit an, VLA's über eine leere Initialisieurngsliste auf 0 zu setzen:

```
void foo(int n) {
   int arr2[n]={0};  //KO, VLA können nicht initialisiert werden!
   int arr1[n]={};   //KO, kein C konformer Syntax
                     //OK bei einigen Compilern
```

Ähnlich wie bei der Initialisierung von Strukturen über designated initializers können seit C99 (bedingt bei C++) einzelne Array-Elemente über ihren Index initialisiert werden:

```
char arr1[20] = {
   [0]='a',
   [8]=33,34,         //Index 8 und 9 werden Initialisiert
   [3]=35,36,         //Index 3 und 4 werden Initialisiert
                      //Rückwärtsspringen bei C++ nicht möglich
   [10 ... 12]=3      //KO, laut C-Spec
};                    //OK bei einigen Compilern
char arr2[4][3]=  {
  [1][1]=7,
    [2] ={1,2,3},
    [3] ={ [1]=4 },
 };
```

Folgendes gilt es hier zu berücksichtigen:

- Bei C (nicht C++) sind doppelte Indexe erlaubt, die letzte Beschreibung in der Initialisierungsliste überschreibt vorherige Initialisierungen
- Bei C++ müssen die Indexe aufsteigend sortiert sein

Weitere Beispiele:

```
char arr1a[4]={1,2,3,4};    //OK
char arr1b[4]={1,2,3,4,5};  //KO Compilerfehler
char arr1c[4]={1,2,3};      //OK, Fehlende Elemente werden auf 0 gesetzt
char arr1d[] ={1,2,3,4};    //OK, Dimension ergibt sich aus der Anzahl der
                            //    Initialisierungselemente
char arr1e[100]={0};        //OK, Fehlende Elemente werden auf 0 gesetzt

char arr2a[6]="hello";      //OK
char arr2b[5]="hello";      //OK, jedoch kein Stringendezeichen vorhanden
char arr2c[4]="hello";      //KO  Mehr Initialisierungswerte als Dimension
char arr2d[8]="hello";      //???
char arr2e[] ="hello" "world" "ich\
  fortsetztung" __DATE__;   //???
int  arr2f[6]="hello";      //???

char arr4a[4][3]  =  { {1,2,3},{4,5,6},{7,8,9},{10,11,12} }; //OK
char arr4b[4][3]  =  { 1,2,3,4,5,6,7,8,9,10,11,12 };         //Außerhalb
                      //der Spec von einigen Compilern unterstützt
char arr4c[3][10] =  {  "hallo","du","da" };                 //???
```

8.9 Typedef

Mittels Typedef können auch Aliase auf Arrays und Zeiger auf Arrays angelegt werden. Der 'Variablenname' ist dabei der Alias:

```
//Typedefs für Arrays
typedef char arr1_t[7];
arr1_t  arr1;                //Array der Dimension 7
typedef char arr2_t[];       //OK (Incomplete Array Definition)
arr2_t arr2;                 //KO, aufgrund der fehlenden Dimension
arr2_t arr3={1,2};           //OK, Dimension ergibt sich aus der
                             //    Dimension der Initialisierung
typedef char arr4_t[5][4][3];
arr4_t arr4;                 //OK dreidimensionales Aray
arr4_t arr5[6];              //OK vierdimensionales Array

//Typedefs für Zeiger auf Arrays
typedef char (*parr3_t)[4][3];   //Definition des Alias arr3_t
parr3_t ptr4;
```

Über Typedef können auch VLA's definiert werden, diese müssen dann im Funktionskontext definiert werden. Die Dimension des Arrays ergibt sich zum Ausführungszeitpunkt der Typedef-Anweisung und nicht zum Ausführungszeitpunkt der Definition der Variablen!

```
void func(int m) {
  typedef int arr_t[m];   //Dimensionsgröße ergibt sich aus m zu diesen
                          //Zeitpunkt der Ausführung der typedef Anweisung
  m++;                    //Änderung von m ändert nichts an der Dimension
  arr_t arr;              //der hier definierten Variable
}
```

8.10 Array-Literale

Über Compound Literal kann auch eine anonymous array definiert werden (nur C nicht C++):

```
//Compound Literal gibt entsprechend der Nutzung eines
//Arraynamens die Startadresse des anonymous Array zurück
int (*arr)[3];
arr=(int [2][3]){{1,2,3},{1,2,3}};

//Übergabe eines Compound Literals an eine Funktion
void foo(int (*arr)[3]) {
}
foo( (int [2][3]){{1,2,3},{4,5,6}} );

//Übergabe eines Compound Literals zur Array Initialisierung
arr=malloc(sizeof(int)*3*2);
memcpy(arr,                          //Destination-Address
       (int [2][3]){{1,2,3},{4,5,6}}, //Source-Address
       2*3*sizeof(int)               //Anzahl zu kopierender Bytes
       );
```

Entsprechend Arrays, bei welchen der Arrayname einem Zeiger auf einem Array entspricht, ist ein Compound Literal ebenfalls ein Zeiger auf ein Array.

Compound Literale sind ungeachtet des Names keine Konstanten, so dass deren Inhalt geändert werden kann. Zur Darstellung von Konstanten muss das Compound Literal explizit als const definiert werden:

```
int (*arr2)[3];
arr2=( int [2][3]){{1,2,3},{1,2,3}};
arr2[1][1]=7;  //Änderung möglich

//Explizite Definition als const
const int (*arr3)[3];
arr3=(const int [2][3]){{1,2,3},{1,2,3}};
arr3[1][1]=7;    //KO, compilerfehler
```

8.11 Array mit 0 füllen

Mittels der memset() Funktion besteht die Möglichkeit, einen Speicherbereich mit einem Byte-Wert zu füllen. Bei Angabe der Startadresse und der Größe des Arrays bietet sich hierüber die Möglichkeit, ein Array mit 0 zu füllen:

```
char arr0[5][4];
memset(arr0,0,sizeof(arr0));         //Speicherbereich/Array mit 0 Füllen

//Alternative Füllwerte füllen das Array nicht mit dem
//erwarteten Wert!
float  arr1[8][9];
memset(arr1,1,sizeof(float)*8*9);
printf("%g",(double)arr1[1][1]);     //Inhalt ist ungleich 1.0
```

8.12 Vergleichen von Arrays

Der Arrayname ist ein Zeiger auf ein Array, so dass hierüber kein Inhaltsvergleich, sondern einzig ein Adressvergleich stattfindet:

```
char arr1[4],arr2[4];
if(arr1 == arr2)                    //Für Compiler OK, jedoch erfolgt hier
  printf("Startadresse identisch"); //ein Vergleich der Startadressen
```

Ein Inhaltsvergleich muss entweder händisch oder kann alternativ über die memcmp() Funktion durchgeführt werden:

```
char arr1[4]={1,2,3,4};
char arr2[4]={1,2,3,4};
//Händischer Vergleich
int lauf;
for(lauf=0;lauf<4;lauf++)
  if(arr1[lauf]!=arr2[lauf])
    break;
if(lauf == 4)
   printf("Arrays sind identisch");
//Vergleich mittels memcmp()
if(memcmp(arr1,arr2,sizeof(arr1)) == 0)
   printf("Arrays sind identisch");
```

Vorsicht: Wenn aufgrund von alignment der Compiler die zugrundeliegende Datenstruktur vergrößert (bspw. struct xy {int x; char y;}; von 5-Bytes auf 8-Bytes) kann memcmp nicht genutzt werden!

8.13 Kopieren von Arrays

Der Arrayname ist ein Zeiger auf ein Array, so dass ein Kopieren entweder händisch oder mittels der memcpy() Funktion erfolgen muss:

```
char arr1[4];
char (*arr2);
arr2=arr1;           //Syntax OK, jedoch nur Kopie der Startadresse
                     //und nicht des Inhaltes!
arr2=malloc(sizeof(arr1));
//Händisches Kopieren
for(int lauf=0;lauf<4;lauf++)
  arr2[lauf]=arr1[lauf];

//Kopieren mittels memcpy(), Größenangabe ermittelt aus 'Quellarray'
memcpy(arr2,arr1,sizeof(arr1));
memcpy(arr2,arr1,sizeof(arr2));  //Hier werden nur 4/8 Bytes kopiert
```

8.14 Array bei Funktionsaufrufen

8.14.1 Parameterübergabe

Der Arrayname ist ein Zeiger auf ein Array. Dies bedeutet, man kann das Array nicht als Ganzes (d.h. mit seinem gesamten Inhalt) ansprechen. Folglich kann ein Array nicht als Call by Value übergeben werden, sondern immer nur als Call by Reference.

Für die Datentypdefinition im Funktionskopf stehen 3 Varianten zur Verfügung:

- Zeiger auf ein Array
- Datentypdefinition entsprechend eines Arrays unter Angabe aller Dimension
- Datentypdefinition entsprechend eines Arrays mit fehlender Angabe der linken Dimension

Letztere beiden lassen vermuten, dass hier eine Call By Value Übergabe stattfinden. Dies ist ein Trugschluss. Alle 3 Datentypdefinition erzeugen, angewendet im Funktionskopf, einen Zeiger auf ein Array:

```
char arr1[3];
char arr2[5][3];
char arr3[5][4][3];

//Datenübergabe bei eindimensionalem Array
//und 3 unterschieden Datentypedefinitionen
//(alle 3 erzeugen ein Zeiger auf ein Array)
void  func11(char *ar);
void  func13(char ar[3]);
void  func12(char ar[]);

//Datenübergabe bei mehrdimensionalen Arrays
//und 3 unterschiedlichen Datentypedefinitionen
void func23(char (*ar)[3]);
void func21(char ar[5][3]);
void func22(char ar[][3]);

void func33(char (*ar)[4][3]);
void func31(char ar[5][4][3]);
void func32(char ar[][4][3]);
```

Ergänzend zu der Dimensionsangabe des Arrays über Konstante kann die Dimension auch entsprechend VLA's über Variablen (und Ausdrücke) definiert werden (nicht C++). In diesem Fall muss die Dimensionsvariable zuvor definiert worden sein. Entweder als globale Variable oder im Funktionskopf ergänzend als eigenständiger Parameter übergeben werden. Die tatsächliche Dimension des Arrays wird zum Aufrufzeitpunkt der Funktion ausgewertet:

```
//Definition der Definitionsvariable als globale Variable
int dim1,dim2;
void funcxy(char arr[dim1][dim2]);

//Übergabe der 'Dimensionvariablen' vor der Arrayübergabe
void funcxy(int dim1, int dim2, char arr[dim1][dim2]);

//Unter Angabe der fehlenden linken Dimension
void funcxy(int dim2, char arr[][dim2]);

//Über Forward Parameter Definitionen im Funktionskopf
//(nur GNU-C) können die 'Dimensionvariablen'
//auch nach dem Array übergeben werden:
//https://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html#Variable-Length
void funcxy(int dim1, int dim2; char arr[dim1][dim2], int dim1, int dim2);
//          --------------                             ------------------
//          Prototypen                                 Übergabeparameter
//Hinweis:
//Semikolon trennt im Funktionskopf beim GCC Compiler zwischen
//Prototypenbereich und dem Variablenbereich!
```

Egal, über welche Art das Array im Funktionskopf definiert wurde, der Datentyp ist ein Zeiger auf ein Array. Folglich gibt der sizeof-Operator angewendet auf den Arraynamen die Größe des Zeigers zurück:

```
void foo(int n, int arr[n][n]) {
  printf("sizeof=%zd",sizeof(arr));  //4/8
```

Hinweis:

- Bei Arrayübergabe über VLA's kann alternativ zur Dimensionsangabe über eine Variable auch ein beliebiger Ausdruck stehen, der zum Aufrufzeitpunkt der Funktion ausgewertet wird:

```
//Quelle leider nicht notiert
int main(int argc, char *argv[printf("Hello")]) {
  printf(" world!\n");
  return 0;
}

void f(char _[(
               printf("Press enter to confirm: "),
               getchar(),
               1
               )]) {
  printf("thanks.\n");
}

void imax(int *out, int a, int b, char _[(
                                          *out = a > b ? a : b,
                                          1
                                          )]) {}
```

8.14.2 Werterückgabe

Ergänzend zur Übergabe eines Zeigers auf ein Array an eine Funktion kann auch ein Zeiger auf ein Array zurückgegeben werden. Anstatt der Klammerung des Variablennamens muss hier die Funktionsname inkl. Funktionskopf geklammert werden!

Entsprechend der Rückgabe von 'normalen' Zeiger (siehe Zeiger:Werterückgabe[6]) muss auch hier Achtsamkeit auf den Gültigkeitsbereich gelegt werden, auf den der Zeiger zeigt:

```
int (*foo(void))[3] {
  int arr[3][3];
  printf("hallo");
  return arr;  //KO Array arr ist nach Funktionsende nicht mehr gültig
}
```

Die bei der Parameterübergabe alternativen Schreibweisen für den Datentyp sind hier nicht möglich:

```
char [] function(void)          //KO
char [3] function (void)        //KO
char (*arr)[3] function(void)   //KO
char *function(void)            //Zeiger auf Char!
```

Alternativ bietet es sich an, ein Alias für ein Array zu erstellen und diesen als Rückgabewerte zu nutzen. Dies verbessert zudem die Lesbarkeit des Codes:

```
typedef int (*alias_t)[3];
alias_t foo(void) {
   return (alias_t) NULL;
}
```

6 Kapitel 7.12.3 auf Seite 164

8.14.3 Call by Value

Strukturen können wahlweise als Call-By-Value und Call-by-Reference übergeben werden. Über den Weg, ein Array in eine Struktur zu packen, kann ein Array auch als Call-by-Value übergeben werden:

```
struct arr {
    int arr[4][3];    //sizeof(int)*3*4=48-Bytes
};
struct arr foo(struct arr par) {
  struct arr lok=par;
  //Kopieren von 48-Byte von par nach lok;
  return lok;
  //Kopieren von 48-Byte aus der lokalen Variable par nach ret
}
struct arr ret=foo((struct arr){.arr={{1,2,3},{4}} });
//Kopieren von 48 Bytes vom anonymous Array in die lokale variable par
```

8.14.4 Const Matrizen

Entsprechend Const-Zeiger sollten Zeiger auf Arrays als const übergeben werden. Dies verhindert, dass der Callee Änderungen am Array durchführen kann:

```
void lut1(const int (*arr)[10]) {
  arr[1][1]=3;   //KO Const Array kann nicht beschrieben werden
}
void lut2(const int arr[][9]      ) {
  arr[1][1]=3;   //KO Const Array kann nicht beschrieben werden
}
```

8.15 Sonstiges

- In einer Struktur kann das letzte Element ein Array ohne Größenangabe sein (siehe Datentypen:Incomplete Array[7])
- Bei einem Prototyp für ein Array kann wie bei Funktionsaufrufen die linke Dimension entfallen. Bei fehlender linken Dimension versagt sizeof auf den Prototyp

```
//Prototypen
extern int vect[];
extern int arr1[][]; //KO keine Dimensionsangabe
extern int arr2[][3];
extern int arr3[4][3];

void foo(void) {
  vect[1]=3;     //OK da linke Dimensionsangabe nicht notwendig
  arr1[1][2]=3;  //KO da keine Dimeniensangabe im Prototyp vorhanden
  arr2[1][2]=3;  //OK da linke Dimensionsangabe nicht notwendig

  size_t vect_size=sizeof(vect);  //KO aufgrund fehlender Dimensionsangabe
  size_t arr2_size=sizeof(arr2);  //KO aufgrund fehlender Dimensionsangabe
  size_t arr3_size=sizeof(arr3);
}
//Definitionen
int vect[3];
int arr1[4][3];
```

7 Kapitel 5.7.10 auf Seite 113

```
int arr2[4][3];
int arr2[4][3];
```

- Bei Arrayzugriffen können entsprechend dem w:Assoziativgesetz[8] Arrayname und Index 'ausgetauscht' werden: `array[index] → index[array]`

```
//Quelle: https://gist.github.com/fay59/5ccbe684e6e56a7df8815c3486568f01
int foo(int* ptr, int index) {
  //When indexing, the pointer and integer parts
  //of the subscript expression are interchangeable.
  return ptr[index] + index[ptr];
  //It works this way, according to the standard (§6.5.2.1:2),
  //because A[B] is the same as *(A + B), and addition
  //is commutative.}
}
```

- Der Dereferenzierungoperator, ergänzt um Zeigerarithmetik und der Array Elementzugriff sind identisch:

```
int arr[4][5][6];
arr[1][1][1]=1;        //identisch zu
*(arr[1][1]+1)=1;
arr[0][1][2]=2;        //identisch zu
*(*(arr[0]+1)+2)=1;
arr[3][2][1]=3;        //identisch zu
*(*(*(arr+3)+2)+1)=3;
```

8.16 C++

Wie in C können auch in C++ von jedem Datentyp Arrays angelegt werden. Dies gilt auch für Klassen:

```
int arr[10];
class ABC {
  public:
    int a[10];
    ABC(void) {}
} obj1[10];
ABC  obj2[20];
ABC *pobj=new ABC[30]();
std::string arr_of_strings[4];
```

Die Dimension dieser Arrays wird mit der Definition angegeben und kann zur Laufzeit nicht geändert werden. Der Arrayname ist wie in C nur ein Zeiger auf das erste Element.

VLA sind in GNU-C++ enthalten, jedoch kein Bestandteil der C++ Spezifikation.

8 `https://de.wikipedia.org/wiki/Assoziativgesetz`

Mit der Klasse `std:array` wird ein erweiterter Array-Datentyp bereitgestellt, der ergänzende Methoden für den Vergleich, das Sortieren, usw. zur Verfügung stellt:

```
//Quelle: https://en.cppreference.com/w/cpp/container/array
std::array<int, 3> a1{{1, 2, 3}};
std::array<int, 3> a2 = {1, 2, 3};
// Container operations are supported
std::sort(a1.begin(), a1.end());
std::ranges::reverse_copy(a2, std::ostream_iterator<int>(std::cout, " "));
std::cout << '\n';

// Ranged for loop is supported
std::array<std::string, 2> a3{"E", "\u018E"};
for (const auto& s : a3)
  std::cout << s << ' ';
```

Auf Basis der Klassen `std::vectoren` und `std::list` aus der Standard Template Library (STL) stehen 'Arrays' zur Verfügung, deren Größe während der Laufzeit dynamisch geändert werden können. Zugriffe auf Objekte dieser Klasse sind folglich langsamer als reine Array Zugriffe.

Die Klasse `std::vector` entspricht dem klassischen Array, d.h. die Elemente sind nacheinander im Speicher angeordnet. Beim Anlegen eines `std::vector` wird mehr Speicher reserviert, als notwendig, so dass ein Einfügen eines Elementes nicht jedesmal zu einem Realloc führt. Ein Einfügen in der Mitte des Vectors bedingt das 'hochschieben' aller Elemente über der Einschiebeposition:

```
//Quelle: https://en.cppreference.com/w/cpp/container/vector
// Create a vector containing integers
std::vector<int> v = {8, 4, 5, 9};

// Add two more integers to vector
v.push_back(6);
v.push_back(9);

// Overwrite element at position 2
v[2] = -1;
v[10] = -1;   //KO Programmabsturz

// Print out the vector
for (int n : v)
  std::cout << n << ' ';
std::cout << '\n';
```

Die Klasse `std::list` basiert auf einer doppelt verketten Liste. Beim Einfügen wird Speicher für die Daten und für die Verkettungselemente vom Heap reserviert. Im Anschluss werden einzig die Zeiger des vorliegenden und nachfolgenden Elementes umgebogen. Der Zugriff auf ein Element bedingt das Durchlaufen der Liste vom Anfang oder Ende, bis das Zielobjekt gefunden wurde:

```
//Quelle: https://en.cppreference.com/w/cpp/container/list
// Create a list containing integers
std::list<int> l = {7, 5, 16, 8};

// Add an integer to the front of the list
l.push_front(25);
// Add an integer to the back of the list
l.push_back(13);

// Insert an integer before 16 by searching
auto it = std::find(l.begin(), l.end(), 16);
if (it != l.end())
   l.insert(it, 42);
```

```
// Print out the list
for (int n : l)
   std::cout << n << ", ";
std::cout << "\n";
```

9 Präprozessor

Der Präprozessor ist ein von C-Syntax unabhängiger Sprachsyntax. Beim Übersetzungsprozess einer C-Datei wird der Präprozessor als zweiter Prozess (nach dem Entfernen der Kommentare) ausgeführt und das Ergebnis dieser Übersetzung dem eigentlichen C-Compiler vorgelegt. Der Präprozessor erzeugt dabei keinen ausführbaren Code (also keine Maschinensprachebefehle wie der C-Compiler) sondern entspricht im Wesentlichen einer Textersetzung. Prinzipiell könnte der C-Präprozessor auch als Präprozessor für andere Sprachen genutzt werden.

Das Resultat des Übersetzungsprozesses kann mittels des Compilerschalters 'gcc -E' dargestellt werden (im Compiler Explorer dazu '-E' im Feld 'Compiler options' eintragen). Dieser stoppt den Übersetzungsprozess nach Durchlauf des Präprozessors und gibt den erzeugten C-Code auf der Standardausgabe aus. Vorsicht, bei vielen Include-Anweisungen ist die Ausgabe schnell mal mehrere tausend Zeilen lang. Im Compiler Explorer kann alternativ im Compiler-Fenster unter 'Add new...' über '# Preprocessor' ein weiteres Fenster mit der Preprocessor Ausgabe geöffnet werden.

Der Syntax des Präprozessors weicht deutlich vom C-Syntax ab. Wesentliche Kennzeichen des Syntax sind:

- Kein Semikolon zum 'Abschluss' eines Befehls
- Die meisten Befehle starten mit einem # und Enden am Zeilenende
- Zur Fortführung eines Befehls in der nächsten Zeile muss die fortzuführende Befehlszeile mit einem Backslash '\' abgeschlossen werden

Die Befehle lassen sich in folgende Kategorien unterteilen:

- Include-Anweisung
- Object-Like Makros
- Function-Like Makros
- Bedingte Übersetzung
- Sonstiges

9.1 Include

Entspricht einer Textersetzung, wobei die Textzeile mit der Include-Anweisung durch den Inhalt der adressierten Datei ersetzt wird.

```
Syntax: #include <h-char-sequence >
Syntax: #include "y-char-sequence"
Syntax: #include Object-Like-Makro //Standard-C
```

Die Include-Anweisung kann jede Datei inkludieren. Damit der folgende Compilerlauf keine Fehler meldet, sollte diese Datei C-Syntax enthalten. Typischerweise werden die inkludierten Files Header-Files genannt und haben die Dateiendung .h oder .hpp.

Die inkludierten Files werden ebenfalls vom Präprozessor geparst, so dass z.B. hier enthaltene Include-Anweisungen ebenfalls aufgelöst werden (Vorsicht, Gefahr der doppelten Includierung von identischen Dateien und damit ggf. eine Endlosschleife)

Beispiele:

```
#include <stdio.h>
#include "class.h"
#define includefile1 <stdint.h>
#define includefile2 "main.h"
#include includefile1
//Der Include-Anweisungen folgende Anweisungen werden nicht übersetzt!
#include "test.h" int var=3;
int main(int argc,char *argv[]) {
  var++; //KO   Anweisung 'int var=3;' wurde nicht übersetzt
  return 0;
}
```

9.1.1 Suchpfade

"y-char-sequence"

Suchpfad für die einzubindende Datei ist das aktuelle Arbeitsverzeichnis, von dem der Compiler gestartet wurde. Über relative Pfadangaben können Dateien in 'Nachbarschaft' und über absolute Pfadangabe Dateien von 'überall' inkludiert werden:

```
#include "test.h"
#include "verzeichnis/hallo.h"
#include "\home\xyz\test.h"
#include "../lib/test.h"
//Vorsicht, Windows nutzt '\' und Unix '/' zur Pfadtrennung,
```

<h-char-sequence>

Nutzung des vom Compiler vorgegebenen Suchpfades zum Suchen nach der angegebenen Datei. Im Compiler-Suchpfade sind unter anderem die Verzeichnisse der Header-Dateien der Standard-C-Library enthalten. Über relative Pfadangaben können Dateien in 'Nachbarschaft' zum Suchpfad inkludiert werden:

```
#include <stdio.h>
#include <sys/stat.h>
```

Mittels `gcc -Ixxx` kann der Suchpfade um die Pfadangabe xxx ergänzt werden, so dass z.B. die Header-Dateien von Librarys direkt (ohne Pfadangabe) angesprochen werden können. Die Liste der vom Compiler genutzten Suchpfade kann mittels 'cpp -v' oder 'cpp

-v /dev/null' ausgegeben werden:

9.1.2 Probleme im Umgang mit Header-Dateien

Bei unachtsamer Nutzung der include Anweisung können gewollt/ungewollt:

- Identische Header-Datei mehrmals zu einer C-Datei inkludiert werden (direkt oder auch indirekt):

datei.h

```
#include <stdio.h> //doppelt inkludiert
int var; //*1)
typedef unsigned int UI;
#define HALLO 1
```

main.c

```
#include <stdio.h>
#include "datei.h"
#include "datei.h" //doppelt!
```

- Eine Header-Datei in unterschiedlichen Dateien inkludiert werden:

datei1.h

```
int var; //*2)
typedef unsigned int UI;
#define HALLO 1
```

func1.c

```
#include "datei.h"
...
```

func2.c

```
#include "datei.h"
...
```

Damit in beiden Fällen ein fehlerfreier C/C++-Code resultiert, unterliegt der Inhalt von Header-Dateien einigen Restriktionen:

- Keine Definitionen von Variablen und Funktionen in Header-Dateien (da Header-Datei in unterschiedlichen Dateien inkludiert werden, die dann zu eigenständigen Variablen/Funktionen in jeder erzeugten Objektdatei führt, siehe *1) und *2))
- Beliebig viele Deklarationen möglich (sofern diese vom identischen Datentyp sind)
- Keine doppelte Definition von Datentypen (aufgrund der Gefahr bei doppelter Einbindung der identischen Header-Datei)
- Keine doppelte Definition von Markos (Gefahr bei doppelter Einbindung der identischen Header-Datei)

Datentypen und Makros sind fester Bestandteil von Header-Dateien. Zur Sicherstellung, dass diese bei doppelter Einbindung einer Header-Datei nicht doppelt ausgeführt werden, sind Include-Wächter notwendig:

```
#ifndef _Projektname_Dateiname_Dateiendung_INCLUDED_
#define _Projektname_Dateiname_Dateiendung_INCLUDED_
//Eigentlicher Inhalt der Header-Datei
#endif
```

Alternativ kann nachfolgende, von vielen Compiler unterstützte Anweisung, am Anfang der Header-Datei genutzt werden (siehe auch pragma once[1])

```
#pragma once
//Eigentlicher Inhalt der Header-Datei
```

9.1.3 Anwendung

Einzubindende-Dateien werden u.A. für folgende Anwendungsfälle genutzt:

- Deklarationen von Funktionen, die in einer C-Datei geschrieben sind und von anderen C-Datei genutzt werden sollen (entspricht der Public Zugriffsmethode in OOP):

class.h

```
int class_set(int val,int attr);
int class_init(void);
```

class.c

```
int class_set(int val,int attr)
{
   ...
}
int class_init(void)
{
   ...
}
static void class_priv(void)
{
   ...
}
```

func1.c

```
#include "class.h"
void func1_init(void)
{
   ...
   class_init();
   ...
   class_set(4,1);
}
//Für class_priv()
//kein Prototyp
//vorhanden, somit
//nicht aufrufbar!
```

- Definition von projektweiten Konstanten und Datentypen, welche von allen C-Dateien genutzt werden können:

common.h

1 https://en.wikipedia.org/wiki/pragma%20once

```
#define DNS_ADDR "141.41.1.150"

#define ERROR(text, arg... ) \
        (fflush(stdout),\
         fprintf(stderr,\
         "\e[31m%s() Error:\e[30m "\
         text"'\n",__func__,##arg) )

#define MODE_TemperaturOnly  1
#define MODE_DruckOnly       2
#define MODE_AttitudeOnly    3
#define MODE_TemperaturDruck 4
#define MODE  MODE_AttidueOnly

typedef struct {
  int mode;
  int debuglevel;
  ...
} global_t;
```

- Zur Nutzung von Librarys muss einerseits die Library als solches zur ausführbaren Datei dazu gebunden werden (Compilerschalter gcc -lxxx). Zum Aufruf der in der Library enthaltenen Funktionen sind ergänzend die Prototypen der Funktionen, die notwendigen Datentypen und die Konstanten über include in die Source-Datei einzubinden:

stdio.h

```
//Auszug aus stdio.h
...
extern int fprintf (FILE *__restrict __stream,
                    const char *__restrict __format, ...);
extern int printf (const char *__restrict __format, ...);
extern int sprintf (char *__restrict __s,
               const char *__restrict __format, ...)
                    __attribute__ ((__nothrow__));
extern int vfprintf (FILE *__restrict __s,
                     const char *__restrict __format,
                     __gnuc_va_list __arg);
  ...
```

Die Prototypen, die Datentypen und die Konstanten der Standard-C Library sind in diverse Header-Dateien verteilt: C-Standard-Bibliothek[2]

9.1.4 Sonstiges

- C89: Max Rekursionstief 8

2 https://de.wikipedia.org/wiki/C-Standard-Bibliothek

- C99: Max Rekursionstief 15
- Bei #include <> muss der Compiler die Datei nicht laden, sondern kann eine eigene (gespeicherte Header-Datei) nutzen (zwecks schnellerer Übersetzung)
- Include lädt einzig die Deklarationen in die aktuelle C-Datei ein. Die dazugehörigen Definitionen sind beim Linker Durchlauf, z.B. durch Librarys separat beizufügen (Compilerschalter gcc -lxxx)
- C++ benutzt ein Name-Mangling System (siehe Name mangling[3]), so dass auf deren Funktionen und Variablen nicht direkt zugegriffen werden kann. Wird aus einem C++-Projekt eine Library erstellt, so kann das Name-Mangling wie folgt 'deaktiviert' werden:

```
extern "c" void test(void);

#ifdef __cplusplus
extern "C" {
#endif
  void test(void);
#ifdef __cplusplus
}
#endif
```

- Zum Inkludieren von Standard-C Header Dateien in C++ muss dem Dateienamen ein c vorangestellt werden. Die Endung .h entfällt:

```
#include <cstring>   //zum Einbinden der Datei string.h unter c++
```

- Mit jeder Include Anweisung erhöht sich die Zeitdauer für einen Compiledurchlauf. Einerseits muss jede zu inkludierende Datei von der Festplatte geladen werden und andererseits vergrößert sich mit jeder Datei der zu übersetzende Code. In diesen Sinn gilt 'weniger ist mehr'. Also gerne mal regelmäßig die Include-Anweisungen durchgehen und überlegen, ob diese in der Tat benötigt werden. Tipp: Hinter der Include-Anweisung im Kommentar vermerken, für welche Funktionen diese Header-Datei benötigt wird

9.2 Object-Like Makros

Entspricht einer Textersetzung auf Wortbasis, wobei das Wort `name` im folgenden Code durch `sequence-of-tokens` ersetzt wird. Innerhalb von Kommentaren und Strings erfolgt keine Textersetzung.

`Syntax: #define name sequence-of-tokens`$_{\text{opt}}$

`'Sequence-of-tokens'` endet am Zeilenende. Soll das Makro in der Folgezeile fortgesetzt werden, so muss die Zeilenfortsetzung (siehe Grundlagen:Zeilenfortsetzung[4]) genutzt werden. Wenn `'sequence-of-tokens'` leer/nicht angegeben ist, wird `'name'` durch nichts ersetzt, d.h. `'name'` wird aus dem Source-Code entnommen und durch ein 'Leerzeichen'

3 https://en.wikipedia.org/wiki/Name%20mangling
4 Kapitel 4.4 auf Seite 37

ersetzt.

Die bedingte Übersetzung prüft oftmals nur ab, ob ein Makro definiert ist (mittels `#ifdef name` oder `#if defined(name)`). Der 'zugewiesene' Wert `sequence-of-tokens` ist dabei nicht von Interesse, so dass dessen Angabe in diesem Anwendungsfall ebenfalls nicht nötig ist.

Wie Variablen und Funktionsnamen dürfen Makros nicht doppelt definiert werden. Soll ein vorhandenes Makro gelöscht oder neu gesetzt werden, so kann das vorhandene Makro mit `#undef` gelöscht werden. Im nachfolgenden Source-Code erfolgt dann keine Textersetzung für dieses Makro mehr.

`Syntax: #undef name`

Die resultierenden Fehlermeldungen nach der Textersetzung sind Fehlermeldungen aufgrund des ungültigen C-Syntax, hervorgerufen durch die Textersetzung. Auf den ersten Blick sind diese Fehlermeldung nur schwer nachvollziehbar, da gedanklich die Textersetzung vor Interpretation der Fehlermeldung durchzuführen ist. Daher empfiehlt sich in solchen Fällen, den resultierenden Source Code nach der Textersetzung mittels `gcc -E` sich darstellen zu lassen.

Beispiele:

```
#define   ROT1 var
int ROT1=2,ROT11=3;   //Nur Wortweise ersetzung!

#define   ROT2
int ROT2 var1=1;      //ROT2 wird durch nichts ersetzt
    #   define ROT3     8  //Leerzeichen werden eleminiert
#define   ROT4 /*Test */ "Hello World"
#define  ROT5 5;
int a=ROT5,b=ROT5+6;  //KO Compilerfehler, da ';' bestandteil der Ersetzung

#define  ROT6 1,"hallo",7
char str[]="ROT6";  //Keine Ersetzung von ROT6, da innerhalb eines Strings
struct {int a; char str[7]; int b;} var2={ROT6};

#define  else        //C-Befehl else durch nichts ersetzen
if(1==2) printf("hallo"); else printf("welt");

#define ADDAB a+b
int  d,e,f=ADDAB;

#define  ROT1  //KO da Mehrfachdefinition verboten
#undef   ROT1  //für begrenzte Gültigkeit oder Überschreibung
int ROT1=2;    //Definition einer Variablen ROT1
#define  ROT1
ROT1=2;        //ROT1 wird durch nichts ersetzt

#define ROT7 BACKSLASH \
#define ROT8  8                     //Befehlsfortführung

#define XYZ  9
enum {XYZ=9};             //KO XYZ wird durch nichts ersetzt
```

Ergänzend zu den selbst definierten Makros gibt die C-Spezifikation folgende vordefinierten Makros vor, welche vom Compiler durch die entsprechende Gegebenheiten ausgetauscht werden:

name	Datentyp	Bedeutung
__LINE__	Konstante vom Type int	Wird ersetzt durch die aktuelle Zeilennummer, in welcher das Makro steht
__FILE__	const char *	Wird ersetzt durch den Dateinamen der C-Datei
__DATE__	const char *	Wird ersetzt durch das aktuelle Datum zum Compilezeitpunkt
__TIME__	const char *	Wird ersetzt durch die aktuelle Uhrzeit zum Compilezeitpunkt
__STDC__	Konstante vom Type int	Wert=1, wenn der Compiler auf Standard-C konform eingestellt ist
__STDC_VERSION__	Konstante vom Type int	Wert=YYYYMM Jahr und Monat der C-Version Bsp: 199409 für C89 199901 für C99
__func__ (ab C99)	char *	wird ersetzt durch einen Zeiger auf einen String, in welchem der aktuelle Funktionsname enthalten ist

Beispiel für Nutzung dieser Makros

```
printf("File:"__FILE__ " erstellt am "__DATE__" um "__TIME__ " gestartet\n");
#define MELDUNG(text) fprintf( stderr, "Datei [%s], Zeile %d: %s\n" \
                              ,__FILE__, __LINE__, text )
```

Weiterhin gibt es vom Compiler vordefinierte Makros, die in Abhängigkeit des Betriebssystems, der Rechenbreite, des Prozessors und vielen anderen Bedingungen gesetzt werden. Neben der reinen Informationen dienen diese Makros auch dazu, plattformunabhängigen Code zu schreiben. Auszug aus Compilerinternen Makros:

name	Datenty	Bedeutung
__GNUC__	—	Defined, wenn der Compiler die GNU-C Spezifikation unterstützt
__INCLUDE_LEVEL__	int	Zahl, welche den aktuellen Include-Level angibt
__CHAR_UNSIGNED__	—	Defined, wenn Char vom Datentyp unsigned ist
__TIMESTAMP__	const char *	String mit dem letzten Änderungsdatum der Datei

und viele weitere:

```
__WIN32 __unix__ __linux__ __i386__ __CYGWIN32__
```

Die vom Compiler vordefinierten Makros können mit nachfolgender Anweisung ausgegeben/angezeigt werden:

`cpp -dM -E` oder `cpp -dM -E -xc /dev/null`

Ergänzend zur `#define` Anweisung im Source-Code können Makros mit dem Compileraufruf gesetzt werden:

```
gcc -Dxxxx[=definition] //für Object-Like Makros
gcc -Dxxx[(arglist)=Definition] //für Function-Like Makros
```

Diese Makro-Definition wird gerne von den Build-Tool genutzt, über welche globale Einstellungen getätigt werden. Ein typisches hier gesetztes Makro ist NDEBUG das im Auslieferungszustand (RELEASE) der Software gesetzt wird. Mittels bedingter Übersetzung können auf Grundlage dieses Makros z.B. Debug-Ausgaben unterbunden werden.

9.2.1 Anwendung

- Objekt-like-Makros werden oft in Verbindung mit bedingter Übersetzung zur Erzeugung von:
  - Host-System (Compiler) unabhängigen Code
  - Target-System (Rechnerarchitektur) unabhängigen Code
  - Debug/Release Konfiguration
  - Softwarevarianten Verwaltung
  - Derivatensteuerung

genutzt. Entsprechende Beispiele siehe Bedingte Übersetzung[5]

- Objekt-like-Makros stellen eine Alternative zu const Variablen dar. Da erste keinen Speicherplatz beanspruchen, resultiert hieraus kleiner und schneller Code:

```
const int val=4711;
#define VAL 4711
//Für die Größenfestsetzung von Arays
#define ARR_SIZE 16
int arr[ARR_SIZE];
```

- Zur Beschreibung von projektweit gültigen Konstanten:

```
#define DNS "192.168.0.1"
#define TASK_MAX  32
```

- In der Header-Datei 'inttypes.h' sind Konstanten als Alternative für die printf Formatstringanweisungen enthalten:

5 Kapitel 9.4 auf Seite 216

```
#include <inttypes.h>
//Auszug aus inttypes.h
#define PRId8   "d"
#define PRId64  "lld"
#define SCNi8   "hhi"
//so dass ein Formatstring wie folgt zusammengesetzt werden kann:
char var=47;
long lo=11;
printf("var=%" SCNi8 " lo=%" PRId64 "\n",var,lo);
```

9.3 Function-Like Makros

Function-like Makros entsprechen einer doppelten Textersetzung. Einerseits wird der Name durch `sequence-of-tokens` und ergänzend die in `sequence-of-tokens` enthaltenden `Identifier` (aus der `identifier-List`) durch den beim Aufruf des Makros enthaltenen Text ersetzt:

`Syntax: #define name( identifier-List`$_{\text{opt}}$) sequence-of-tokens$_{\text{opt}}$

Beispiel für die doppelte Textersetzung:

```
#define ADD(a,b) a+b
int var=ADD(7,8);
   //1. Ersetzung von sequence-of-tokens  → int var=a+b(7,8);
   //2. Identifier a wird durch 7 ersetzt → int var=7+b(8);
   //3. Identifier b wird durch 8 ersetzt → int var=7+8;

//Function-Like Makros entsprechend generischen inline Funktionen
int   a=1  ,b=2  ,c=ADD(a,b);  //Integer Addition
float x=1.0,y=2.0,z=ADD(x,y);  //Float Addition
```

Hinweise:

- `Sequence-of-tokens` endet am Zeilenende. Soll sich das Makro über mehrere Zeilen erstrecken, so ist die Zeilenfortsetzung (siehe Grundagen:Zeilenfortsetzung[6]) zu nutzen
- Zwischen name und '(' ist kein Leerzeichen erlaubt (andernfalls handelt es sich dann um ein Objekt-like-Makro und die öffnende Klammer gehört zu sequence-of-tokens)

```
#define ADD  (a,b) a+b  //Mit Leerzeichen
int var=ADD(7,8);   //→ int var=(a,b) a+b(7,8)
```

- wird ein Identifier beim Aufruf ausgelasen, so wird der Identifier durch nichts ersetzt

```
#define ADD(a,b) a+b
q=ADD( ,8);   //Erster Identifier wird durch nichts ersetzt
              //Erzeugt ungültigen C-Syntax → Compilerfehler
q=ADD( , );   //Präprozessorfehler
              //Beide Identifier werden durch nichts ersetzt
```

6 Kapitel 4.4 auf Seite 37

```
              //Erzeugt ungültigen C-Syntax → Compilerfehler
 q=ADD( );    //Präprozessorfehler
```

Makros basieren auf reine Textersetzung. Wird der Rückgabewert eines Makros in einem Ausdruck weiterverarbeitet, so kann der resultierende C-Code eine andere Abarbeitungsreihenfolge bedingen, als erwartet:

```
#define ADD(a,b) a+b
int   a=ADD(1,2)*3;        //a=1+2*3 → 7
int   b=5  ,c=8;
int   d=ADD(b,c)*ADD(b,c); //d=b+c*b+c; → 53
```

Zur Lösung dieser Problematik empfiehlt sich, 'sequence-of-tokens' zu klammern:

```
#define ADD(a,b)  (a+b)
int   a=ADD(1,2)*3;        //a=(1+2)*3 → 9
int   b=5  ,c=8;
int   d=ADD(b,c)*ADD(b,c); //d=(b+c)*(b+c); → 169
```

Werden Ausdrücke beim Aufruf von Makros eingesetzt, so bleiben diese bei der Ersetzung der Identifier erhalten:

```
#define MUL(a,b)  (a*b)
float a = MUL(7+8,8-7);   //a=7+8*8-7 → 64
```

Zur Lösung dieser Problematik ist es ratsam, auch die Identifier zu klammern:

```
#define MUL(a,b)  ((a)*(b))
float a = MUL(7+8,8-7);   //a=(7+8)*(8-7) → 15
```

Wird in einem Makro eine 'lokale' Variable benötigt, so bietet sich ein Compound-Statement an (siehe Grundlagen:Block / Compound-Statement[7]):

```
//Quelle: https://gcc.gnu.org/onlinedocs/cpp/Swallowing-the-Semicolon.html
#define SKIP_SPACES(p, limit) { \
                  char *lim = (limit);         \
                  while (p < lim) {            \
                    if (*p++ != ' ') {         \
                      p--; break; }            \
                    }                          \
                  }
char str[]="  hallo   Welt";
char *ptr=str;
SKIP_SPACES(ptr,str+sizeof(str));
```

Da C-Zeilen, und damit auch Makroaufrufe mit einem Semikolon abgeschlossen werden, ergibt sich ein Problem bei Nutzung solcher Makros in Verbindung mit einer IF-Anweisung:

```
char str[]="  hallo   Welt";
char *ptr=str;
if(ptr != NULL)
  SKIP_SPACES(ptr,str+sizeof(str));  //Makro wird mit einem Semikolon
 abgeschlossen!
else
   printf("Error: ptr==NULL");
```

7 Kapitel 4.13.4 auf Seite 64

Das abschließende Semikolon erzeugt in diesem Anwendungsfall eine Leeranweisung, so dass zwischen `if` und `else` zwei Anweisungen stehen. Dies Problem kann umgangen werden, wenn anstatt des Compound-Statement eine `do {} while(0)` Anweisung genutzt wird:

```
#define SKIP_SPACES(p, limit) do{ \
                  char *lim = (limit);         \
                  while (p < lim) {            \
                    if (*p++ != ' ') {         \
                      p--; break; }            \
                    }                          \
                  }while(0)
char str[]="  hallo   Welt";
char *ptr=str;
if(ptr != NULL)
  SKIP_SPACES(ptr,str+sizeof(str));
else
  printf("Error: ptr==NULL");
```

Sind in Makros bedingte Anweisungen oder das logische UND/ODER enthalten und wird das Makro mit Argumenten aufgerufen, die einen Seiteneffekt hervorrufen (bspw. Postinkrement oder Funktionsaufrufe, die eine globale/statisch lokale Variable ändern) (siehe Seiteneffekt[8]), so kann der Seiteneffekt ebenfalls zu unerwarteten Verhalten herbeiführen:

```
#define MAX(var,a,b)   (var=a>b?a:b)
int a=7,b=8,c;
MAX(c,a++,b++);   //→(c=a++>b++?a++:b++);
                  //b wird zweimal erhöht, a nur einmal!
```

Zur Vermeidung der doppelten Ausführung (und zur Geschwindigkeitssteigerung) sollten die Makroargumente wie bei einem Funktionsaufruf in lokale Variablen zwischengespeichert werden (Call-by-Value):

```
#define MAX(var,a,b)   do { typeof(a) a_=(a);  \
                            typeof(b) b_=(b);  \
                            var=a_ > b_ ? a_ : b_;} while(0)
```

Nach der GNU-C Spezifikation bietet sich alternativ zur 'do {} while(0)' Kapselung ein Embedded Statement an (siehe Grundlagen:Embedded Statement[9]). Dies hat ergänzend den Vorteil, dass dieses einen Rückgabewert hat:

```
#define MAX(a,b)  ({ typeof(a) a_=(a);  \
                     typeof(b) b_=(b);  \
                     a_ > b_ ? a_ : b_;})
int a=7,b=8,c;
c=MAX(a++,b++);
```

Function-Like Makros sind ein mächtiges, aber auch fehleranfälliges Werkzeug. Es empfiehlt sich:

- Ausreichend Klammern (sowohl das Makro als auch die Makroparameter)
- Möglichst Makronamen in GROSSBUCHSTABEN zu schreiben, so dass beim Lesen des Source-Codes klar ist, dass hier ein Makro und keine Variable/Funktion aufgerufen wird
- Innerhalb von Makros den Komma-Operator zum Trennen von Anweisungen nutzen, sofern die Anweisung nicht in einer `do {} while(0)` oder oder Embedded Statement gekapselt sind

8 `https://de.wikipedia.org/wiki/Seiteneffekt%20`
9 Kapitel 4.13.5 auf Seite 65

- Bei Makroaufrufen sind Argumente mit Seiteneffekte (wie Pre-/Postinkrement) zu vermeiden, da diese durch eventuelle Mehrfachauswertung zu unerwarteten Verhalten führen

Hinweis:

- Eine alternative/weiterführende Beschreibung ist in der Online-Dokumentation von gnu zu finden: `https://gcc.gnu.org/onlinedocs/cpp/Macro-Arguments.html` Marco-Arguments[10]

9.3.1 Anwendung

- Funktionalitäten unabhängig vom Datentyp bereitstellen (generische Funktionen)

```
#define MAX(a,b)  ({  /* siehe oben */ })
#define FOREACH(iterator,array)   \
               for(typeof(array[0]) *iterator=&array[0];\
                   iterator<(array+sizeof(array)/sizeof(array[0]));\
                   iterator++)
int arri[]={1,5,3,9};
FOREACH(ptr1,arri)
  printf("%d\n",*ptr1);
```

- Als Alternative zu einem 'langsamen' Funktionsaufruf (entspricht einer inline Funktion)

```
#define read_SCL(port)  (*AT91C_PIOA_PDSR & i2c_mask[port][1])
#define set_SCL(port)   (*AT91C_PIOA_SODR = i2c_mask[port][1])
#define clear_SCL(port) (*AT91C_PIOA_CODR = i2c_mask[port][1])
```

- Lesbareren und dennoch schnellen Code zu erzeugen

```
struct vl {
    struct vl *next;
    char       keydata[];
};
struct vl ele3={NULL ,"key\0value"};
struct vl ele2={&ele3,"schluessel\0wert"};
struct vl ele1={&ele2,"schluessel\0wwwweeeerrrttt"};
struct vl *liste = &ele1;
//Unleserlicher Code
printf("Key='%s' Value='%s'\n",(char *)(&ele2+1),
                               (char *)(&ele2+1)+strlen((char *)(&ele2+1))+1);
//Besser lesbarer Code
#define VL_KEY(vl)    (char *)(vl+1)
#define VL_VALUE(vl)  (char *)(vl+1)+strlen(VL_KEY(vl))+1
printf("Key='%s' Value='%s'\n",VL_KEY(&ele1),VL_VALUE(&ele1));
```

- Für Debug/Release-Zwecke Debugaufrufe im Code zu belassen und diese im Bedarfsfall durch den Aufruf einer Debug-Routine oder durch nichts zu ersetzen

10 `https://de.wikibooks.org/wiki/%20Marco-Arguments`

```
#define DEBUG_PRINT(a)  printf(a)
#define DEBUG_PRINT(a)             //Makro wird durch nichts ersetzt

#define DERIVATE(a)  func1(a)
#define DERIVATE(a)  func2(a,2)
```

- Generisches Makro zum Sortieren eines Arrays (siehe `https://github.com/svpv/qsort` Quicksort as a C macro[11])

9.3.2 Sonstiges

- Über den Ellipsis Puncturator '...' kann einem Function-like Makro beliebig viele Parameter übergeben werden:

```
#define ERROR(text, arg...) (fflush(stdout), fprintf(stderr,        \
                             "\e[31m%s() Error:\e[30m " text "'\n", \
                             __func__, ##arg))
#define ERROR1(text, ...)   (fflush(stdout), fprintf(stderr, \
                             "\e[31m%s() Error:\e[30m " text "'\n", \
                             __func__, __VA_ARGS__))
ERROR("Return-Wert=%d",ret);
ERROR1("Var1=%d var2=%d",var1,var2);
```

- Einige Library- und OS-Funktionen sind als Makros implementiert. Beispielsweise:

```
assert()
pthread_cleanup_pop()
```

9.4 Bedingte Übersetzung

Die bedingte Übersetzung erlaubt es, Textbereiche ein- resp. auszublenden (d.h. Text Bereiche aus dem Source-Code zu entfernen/drinnen zu lassen).

Syntax:

```
#if const-expression-1
    group-of-lines-1  //beliebige Anzahl an Zeilen, die bei gültiger Bedingungen
                      //eingeblendet, andernfalls ausgeblendet werden
#elif const-expression-2
    group-of-lines-2
   …
#else
   group-of-lines-x
#endif
```

`Const-expression` ist eine Integerkonstante oder ein Ausdruck, welcher durch den Präprozessor zu einer Integerkonstante ausgewertet wird. Wenn dieser 0 ist, gilt die Bedingung als

11 `https://de.wikibooks.org/wiki/%20Quicksort%20as%20a%20C%20macro`

nicht erfüllt, andernfalls als erfüllt. Mögliche Operatoren für den Konstantenausdruck sind bspw. `== >= != & | && || + - <<`. Der Wertebereich der Integerkonstanten entspricht nach Standard-C dem Wertebereich von `long` und ab C99 dem Wertebereich von `intmax_t` (largests integer type found on the target).

Beispiel:

```
#define A  1
#if 1
#if A == 1
#if (A+1) == 1
#if (A>>2)&1 == 0x01

#define HALLO ich
#define HAL   "ich"
#if HALLO == ich     //OK Textvergleich
#if HAL == "ich"     //KO Stringvergleich nicht möglich!
```

Hinweise:

- Ein nicht definiertes Makro wird zu 0 ersetzt. Ein definiertes Makro ohne Wertzuweisung wird durch nichts ersetzt, so dass ein Vergleich mit einer Integerkonstanten fehlerhaft ist:

```
#define A 1
#define B
#if A==1
#if B==0    //KO Präprozessorfehler (B wird durch nichts ersetzt)
#if C==0    //OK (C wird durch 0 ersetzt)
```

- Der ergänzende Makro Operator `defined(name)` wird zu 1 aufgelöst, wenn name als Makro definiert wurde (egal, ob und welcher Inhalt zugewiesen wurde), andernfalls 0. Die Kombination von `defined` und der `#if` Anweisung lautet:
  - `#ifdef name` entspricht `#if defined name`
  - `#ifndef name` entspricht `#if !defined name`

```
#define A
#if   defined A
#if ! defined (B)
#ifdef A
#ifndef B
```

- Bedingte Übersetzungen können beliebig verschachtelt sein.

```
#define MAKRO1  2
#define MAKRO2  1
#if MAKRO1==1
  #if MAKRO2==1
  #else
  #endif
#elif MAKRO2==2
#    if MAKRO==1
#    else
#    endif
#endif
```

9.4.1 Anwendung

- Debug/Release (Im Debug-Mode zusätzliche Debug-Ausgaben aktivieren)

```
#ifdef NDEBUG
#define DBGPRINT(val)
#else
#define DBGPRINT(val) printf("%s",val);
#endif
```

- Host-System Abhängigkeiten nutzen (GCC,CLANG,Microsoft / Windows/Linux)

```
#ifdef __unix__
```

- Traget-Unabhängigkeiten (für unterschiedliche Prozessor, Ressourcenausstattung)

```
#ifdef __CHAR_UNSIGNED__
#if INT_MAX==32768
#if sizeof(int)==2    //KO Präprozessor ist sizeof nicht definiert
```

- Derivate-Steuerung (Low-Cost / High-Cost)

```
#define VERSION_CHROM_V90  90
#define VERSION_CHROM_V89  89
#if VERSION == VERSION_CHROM_V90 && defined __unix__
```

9.5 Error/Warning Anweisung

Erzeugung einer Compiler Warning/Fehler-Meldung mit dem Inhalt des `preprocessor-tokens` (muss daher kein String sein).

`Syntax: #error preprocessor-tokens`
`Syntax: #warning preprocessor-Tokens //Nur GNU-C`

9.5.1 Anwendung

- Zur Überprüfung, ob Makros 'richtig' gesetzt wurden

```
#define BUF_SIZE 511
#if (BUF_SIZE % 256) != 0
  #error   Bufsize muss ein vielfaches von 256 betragen
#endif
#if (BUF_SIZE & (BUF_SIZE -1)) != 0
  #error Bufsize muss eine 2er Potenzzahl sein
#endif
#ifndef __unix__
```

```
  #error   Windows ist doof!
#endif
```

9.6 Pragma

Anweisung an den Compiler zum Aktivieren/Deaktivieren bestimmter Compiler-Funktionalitäten.

`Syntax: #pragma preprocessor-tokens`

Die `preprocessor-tokens` sind Compiler-Abhängig!

9.6.1 Anwendung

- Compiler-Optimierung für einzelne Funktionen einschalten

```
#pragma GCC push_options
#pragma GCC optimize ("-O3")
//Einschaltung der max. Compiler Optimierung
void ws2812_send(void) {
}
//Vorherige Compiler Optimierung wieder herstellen.
#pragma GCC pop_options
```

- Anweisung an den Compiler, dass nachfolgende Schleife 'parallelisiert' werden kann

```
//Quelle: https://gcc.gnu.org/onlinedocs/gcc/Loop-Specific-Pragmas.html
void foo (int n, int *a, int *b, int *c) {
  int i, j;
#pragma GCC ivdep
  for (i = 0; i < n; ++i)
    a[i] = b[i] + c[i];
}
```

9.7 Stringizing-Operator

Wird einem Identifier im Ersatztext eines Function-like Makros ein # vorangestellt, so wird bei der Ersetzung (durch den Präprozessor) das Argument durch Einschließen in doppelte Hochkommata in eine Zeichenkette umgewandelt (stringizing).

Beispiel:

```
#define TOSTR(X) #X
char string[] = "hallo";
//Ausgabe des Inhaltes von string gefolgt vom Name der Variablen als String
printf("'%s' '%s'", string, TOSTR( string ) );
```

9.7.1 Anwendung

- Zum Aufbau einer eigenen Symboltabelle, in welcher der Name der Variablen und die Adresse der Variablen enthalten ist:

```
struct var {int *adr; char name[100];};
#define VAR(x) {&x,#x}
int a,b,c;
struct var symboltable[]={VAR(a),VAR(b),VAR(c)};
```

- Umwandlung einer Konstanten in einen String

```
#define TCP_PORT_DEFAULT         4711
int tcp_port=TCP_PORT_DEFAULT;

#define STRINGIFY(x) #x
#define STRINGIFY_RESOLVE(x) STRINGIFY(x)
//Ein Stringconcatenate funktioniert nur über Stringkonstanten
//Zum Umwandlung einer Integerzahl in eine Stringkonstante
//bietet sich der Stringify-Operator an
const char *message_help=
  "TCP-Port (Default: " STRINGIFY_RESOLVE(TCP_PORT_DEFAULT) ")\n";
```

9.8 Verkettung von Makroparametern / Token Merging / Token Concatenation

Der Verkettungsoperator ## erlaubt es, zwei Makroparameter innerhalb eines Function-like Makros zu einem zu verschmelzen.

Beispiel:

```
#define GLUE(X,Y) X ## Y
printf( "%d\n", GLUE(2, 34) );   //Gibt 234 als Zahl zurück

#define TEMP(i)  temp##i
TEMP(1) = TEMP(2 +k)   +x;     //-->temp1=temp2+k+x;

#define PRIVATE(member) private_##member
struct class {
  int PRIVATE(xyz);
```

```
};
obj.PRIVATE(xyz)=4711;
```

9.8.1 Anwendung

- Umwandlung einer Konstanten in einen String

```
#define DEBUG_STATUS_DEBUG 0
#define DEBUG_STATUS_INFO  1
#define DEBUG_STATUS_WARN  2
#define DEBUG_STATUS_ERROR 3

#define DEBUG_STATUS_STR0 "Error+Warn+Info+Debug"
#define DEBUG_STATUS_STR1 "Error+Warn+Info"
#define DEBUG_STATUS_STR2 "Error+Warn"
#define DEBUG_STATUS_STR3 "Error"

#define DEBUG_STATUS_DEFAULT     DEBUG_STATUS_WARN
int debug_status= DEBUG_STATUS_DEFAULT;

#define CAT(a,b) a ## b
#define CAT_RESOLVE(a,b) CAT(a,b)
const char *message_help=
  "Debug (Default=" CAT_RESOLVE(DEBUG_STATUS_STR,DEBUG_STATUS_DEFAULT) ")";
```

Hinweis:

- Auf Basis des Verkettungsoperators sind diverse nützliche Funktionalitäten darstellbar. Mehr dazu siehe `https://github.com/pfultz2/Cloak/wiki/C-Preprocessor-tricks,-tips,-and-idioms` C Preprocessor tricks, tips, and idioms[12]

12 `https://de.wikibooks.org/wiki/%20C%20Preprocessor%20tricks%2C%20tips%2C%20and%20idioms`

10 Autoren

Edits	User
223	D. Justen[1]
82	Dirk Hünniger[2]
1	Intruder[3]
5	Ma.Brauer[4]

1 https://de.wikibooks.org/wiki/Benutzer:D._Justen
2 https://de.wikibooks.org/wiki/Benutzer:Dirk_H%25C3%25BCnniger
3 https://de.wikibooks.org/wiki/Benutzer:Intruder
4 https://de.wikibooks.org/w/index.php%3ftitle=Benutzer:Ma.Brauer&action=edit&redlink=1

Abbildungsverzeichnis

- GFDL: Gnu Free Documentation License. http://www.gnu.org/licenses/fdl.html
- cc-by-sa-4.0: Creative Commons Attribution ShareAlike 4.0 License. https://creativecommons.org/licenses/by-sa/4.0/deed.en
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. http://creativecommons.org/licenses/by-sa/3.0/
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. http://creativecommons.org/licenses/by-sa/2.5/
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. http://creativecommons.org/licenses/by-sa/2.0/
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. http://creativecommons.org/licenses/by-sa/1.0/
- cc-by-1.0: Creative Commons Attribution 1.0 License. https://creativecommons.org/licenses/by/1.0/deed.en
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/
- cc-by-2.5: Creative Commons Attribution 2.5 License. http://creativecommons.org/licenses/by/2.5/deed.en
- cc-by-3.0: Creative Commons Attribution 3.0 License. http://creativecommons.org/licenses/by/3.0/deed.en
- cc-by-4.0: Creative Commons Attribution 4.0 License. https://creativecommons.org/licenses/by/4.0/deed.de
- GPL: GNU General Public License. http://www.gnu.org/licenses/gpl-2.0.txt
- LGPL: GNU Lesser General Public License. http://www.gnu.org/licenses/lgpl.html
- PD: This image is in the public domain.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.
- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.

- LFK: Lizenz Freie Kunst. `http://artlibre.org/licence/lal/de`
- CFR: Copyright free use.
- EPL: Eclipse Public License. `http://www.eclipse.org/org/documents/epl-v10.php`

Copies of the GPL, the LGPL as well as a GFDL are included in chapter Licenses[5]. Please note that images in the public domain do not require attribution. You may click on the image numbers in the following table to open the webpage of the images in your webbrower.

5 Kapitel 11 auf Seite 229

1	D. Justen	CC-BY-4.0
2	D. Justen	CC-BY-4.0
3	D. Justen	CC-BY-4.0
4	D. Justen	CC-BY-4.0
5	D. Justen	CC-BY-4.0
6	D. Justen[6], D. Justen[7]	CC-BY-4.0
7	D. Justen[8], D. Justen[9]	CC-BY-4.0
8	D. Justen[10], D. Justen[11]	CC-BY-4.0
9	D. Justen[12], D. Justen[13]	CC-BY-4.0
10	D. Justen[14], D. Justen[15]	CC-BY-4.0
11	D. Justen[16], D. Justen[17]	CC-BY-4.0
12	D. Justen[18], D. Justen[19]	CC-BY-4.0
13	D. Justen[20], D. Justen[21]	CC-BY-4.0
14	D. Justen[22], D. Justen[23]	CC-BY-4.0
15	D. Justen[24], D. Justen[25]	CC-BY-4.0

6 http://commons.wikimedia.org/w/index.php?title=User:D._Justen&action=edit&redlink=1
7 https:///w/index.php?title=User:D._Justen&action=edit&redlink=1
8 http://commons.wikimedia.org/w/index.php?title=User:D._Justen&action=edit&redlink=1
9 https:///w/index.php?title=User:D._Justen&action=edit&redlink=1
10 http://commons.wikimedia.org/w/index.php?title=User:D._Justen&action=edit&redlink=1
11 https:///w/index.php?title=User:D._Justen&action=edit&redlink=1
12 http://commons.wikimedia.org/w/index.php?title=User:D._Justen&action=edit&redlink=1
13 https:///w/index.php?title=User:D._Justen&action=edit&redlink=1
14 http://commons.wikimedia.org/w/index.php?title=User:D._Justen&action=edit&redlink=1
15 https:///w/index.php?title=User:D._Justen&action=edit&redlink=1
16 http://commons.wikimedia.org/w/index.php?title=User:D._Justen&action=edit&redlink=1
17 https:///w/index.php?title=User:D._Justen&action=edit&redlink=1
18 http://commons.wikimedia.org/w/index.php?title=User:D._Justen&action=edit&redlink=1
19 https:///w/index.php?title=User:D._Justen&action=edit&redlink=1
20 http://commons.wikimedia.org/w/index.php?title=User:D._Justen&action=edit&redlink=1
21 https:///w/index.php?title=User:D._Justen&action=edit&redlink=1
22 http://commons.wikimedia.org/w/index.php?title=User:D._Justen&action=edit&redlink=1
23 https:///w/index.php?title=User:D._Justen&action=edit&redlink=1
24 http://commons.wikimedia.org/w/index.php?title=User:D._Justen&action=edit&redlink=1
25 https:///w/index.php?title=User:D._Justen&action=edit&redlink=1

11 Licenses

11.1 GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS 0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion. 1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work. 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary. 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures. 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee. 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

* a) The work must carry prominent notices stating that you modified it, and giving a relevant date. * b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices". * c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it. * d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate. 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

* a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange. * b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge. * c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b. * d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements. * e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying. 7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

* a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or * b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or * c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or * d) Limiting the use for publicity purposes of names of licensors or authors of the material; or * e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or * f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way. 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10. 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so. 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it. 11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law. 12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from

conveying the Program. 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such. 14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version. 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author> This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

11.2 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free"in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Versionöf the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A SSecondary Sectionïs a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The Ïnvariant Sectionsäre certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Textsäre certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent"copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparentïs called Öpaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page"means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page"means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher"means any person or entity that distributes copies of the Document to the public.

A section Ëntitled XYZ"means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as Äcknowledgements", "Dedications", Ëndorsements", or "History".) To "Preserve the Titleöf such a section when you modify the Document means that it remains a section Ëntitled XYZäccording to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. * B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement. * C. State on the Title page the name of the publisher of the Modified Version, as the publisher. * D. Preserve all the copyright notices of the Document. * E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. * F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. * G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. * H. Include an unaltered copy of this License. * I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "Historyïn the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. * J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History&ection. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. * K. For any section Entitled Äcknowledgementsör "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. * L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. * M. Delete any section Entitled Ëndorsements". Such a section may not be included in the Modified Version. * N. Do not retitle any existing section to be Entitled Ëndorsementsör to conflict in title with any Invariant Section. * O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled Ëndorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "Historyïn the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled Äcknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled Ëndorsements". 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an ăggregateïf the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled Äcknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License ör any later versionäpplies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document. 11. RELICENSING

"Massive Multiauthor Collaboration Site"(or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration"(or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA"means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

Ïncorporate"means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is ëligible for relicensingïf it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts."line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

11.3 GNU Lesser General Public License

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below. 0. Additional Definitions.

As used herein, “this License” refers to version 3 of the GNU Lesser General Public License, and the “GNU GPL” refers to version 3 of the GNU General Public License.

“The Library” refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An “Application” is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A “Combined Work” is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the “Linked Version”.

The “Minimal Corresponding Source” for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The “Corresponding Application Code” for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work. 1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL. 2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

* a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or * b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

* a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

* a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the Combined Work with a copy of the GNU GPL and this license document. * c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document. * d) Do one of the following: o 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source. o 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user’s computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version. * e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

* a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License. * b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy’s public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.