

PROGRAMMIEREN IN C UND C++

BEGLEITSRIPT ZUR VORLESUNG IM WINTERSEMESTER
2021/22

Stand 23. Oktober 2021



Universität Regensburg
Fakultät Physik

Vorwort

Dieses Script soll Sie in der Vorlesung *Programmieren in C und C++* im Wintersemester 2021/22 begleiten. Alle wichtigen Kursinhalte werden hier behandelt und anhand von Beispielen verdeutlicht. Es versteht sich von selbst, dass ein optimaler Lernerfolg nur bei Besuch der Vorlesung gegeben ist. Vorkenntnisse in anderen Programmiersprachen sind zur Arbeit mit diesem Script nicht notwendig.

Hier wird von einer Linux-Arbeitsumgebung ausgegangen und minimale Grundkenntnisse dieser Arbeitsumgebung vorausgesetzt (Starten einer Kommandozeilen-Umgebung, Wechsel des Arbeitsverzeichnisses, Aufrufen von Programmen aus der Kommandozeile, Übergabe von Parametern an Programme). Die DozentInnen des Kurses können bei Bedarf erklären, wie die Arbeitsumgebung bedient wird.

Sie werden primär die Grundlagen der Programmiersprache C (Standardisierung der Sprache von 2011) erlernen. Die Sprache C erschien im Jahr 1972 und wird seither kontinuierlich weiterentwickelt. C erlaubt sehr systemnahe und daher effiziente und vielseitige Programmierung, was die lange Lebensdauer der Sprache erklärt. Anwendungsgebiete umfassen u. a. wissenschaftliches Rechnen, Kernels von Betriebssystemen oder Gerätetreiber.

C inspirierte die Entwicklung vieler anderer Programmiersprachen (darunter C++, C#, Java, D, Go, ...). Die hier erworbenen Kenntnisse sind daher leicht in andere Arbeitsfelder übertragbar und sind daher für Sie ein Sprungbrett in die Arbeit als ProgrammiererIn.

Die Sprache C++ versteht sich als Weiterentwicklung der Sprache C und soll hier nur in Ausblicken behandelt werden. Während C und C++ syntaktisch sehr ähnlich sind, handelt es sich um eigenständige Sprachen, die getrennt voneinander behandelt werden sollten. In diesem Sinne bereitet Sie dieser Kurs darauf vor, einen „vollen“ C++-Kurs zu besuchen, wie er an der Universität Regensburg zum Ende jedes Semesters als Blockkurs angeboten wird.

C++-spezifische Inhalte

Inhalte, die sich auf C++ beziehen sind optisch durch Boxen vom restlichen Kurs abgehoben.

Dieses Dokument ist keine vollständige Referenz der Sprachen C oder C++. Hier sollen nur die Grundlagen der Sprache C sowie Ausblicke auf die Arbeit mit C++ gegeben werden. Als Befehlsreferenz beider Sprachen empfehle ich auf die Seiten <https://en.cppreference.com/w/> (englische, ausführlichere Version) und bzw. <https://de.cppreference.com/w/> (deutsche Version mit eingeschränkter Verfügbarkeit der Artikel), auf die auch im Rahmen dieses Scriptes gelegentlich verwiesen wird.

Dieses Script wurde nach bestem Wissen und Gewissen zusammengestellt; Code-Beispiele wurden auf wenigstens einer Maschine getestet. Dennoch können menschliche Fehler nicht ausgeschlossen werden. Wem Unstimmigkeiten auffallen oder wer Vorschläge und Anregungen zu diesem Text einbringen will, möge mir dies sehr gerne mitteilen. Ich bin erreichbar unter der Email-Adresse:
`stefan.hartinger@stud.uni-regensburg.de`.

Stefan Hartinger, Juni 2021

Inhaltsverzeichnis

| | |
|---|-----------|
| 1. Wie arbeitet ein Computer? | 1 |
| 1.1. Binärdarstellung von positiven Ganzzahlen und Schriftzeichen | 1 |
| 1.2. Von Zahlen zu Befehlen | 3 |
| 1.3. Aufruf des Compilers | 3 |
| 2. Ein erstes Programm | 7 |
| 2.1. Hello World | 7 |
| 2.2. Variablen, Datentypen, Operatoren | 8 |
| 2.2.1. Variablen | 8 |
| 2.2.2. Variablendeklaration | 9 |
| 2.2.3. Wertezuweisung | 9 |
| 2.2.4. Rechnen mit Variablen – Operatoren | 11 |
| 2.2.5. Shorthands | 12 |
| 2.3. Formatierte Ausgabe | 14 |
| 2.4. Kommentare | 17 |
| 2.5. Hello World in C++ | 19 |
| 3. Von der Information zum Bit-muster | 20 |
| 3.1. Daten im Speicher | 20 |
| 3.1.1. Wertemenge und Vorzeichen bei Zahlen | 20 |
| 3.1.2. Adressierung – Pointer | 22 |
| 3.2. Bitweise Logik und Bit-Shifting | 24 |
| 3.3. Typecasting | 26 |
| 3.4. Hierarchie der Operatoren | 27 |
| 3.5. Zahlenformate – Hexadezimalsystem | 27 |
| 4. Dateneingabe | 29 |
| 4.1. Dateneingabe mit <code>scanf</code> | 29 |
| 4.2. Ausblick: Eingabe in C++ | 34 |
| 5. Bedingungen | 35 |
| 5.1. Wahrheitswerte | 35 |
| 5.2. Bedingte Ausführung von Code: <code>if</code> | 37 |
| 5.3. Logische Operatoren | 43 |
| 5.4. Fallunterscheidungen: <code>switch</code> | 47 |
| 5.5. Kombinierte Fallunterscheidung und Wertzuweisung – der Ternäre Operator <code>?</code> | 50 |
| 6. Die CPP-Referenz am Beispiel der math-library | 52 |
| 6.1. Überblick über die Seite <code>cppreference.com</code> | 52 |
| 6.2. Funktionen zu einer Aufgabe finden | 54 |
| 6.3. Der Artikel zu <code>sqrt</code> | 55 |
| 6.4. Weitere nützliche Funktionen der math-library | 57 |
| 6.5. Die C++-Referenz | 58 |
| 7. Schleifen | 59 |
| 7.1. Programmsprünge: <code>goto</code> | 59 |

| | |
|--|------------|
| 7.2. Schleife mit Bedingung: while | 61 |
| 7.3. Schleife mit Bedingung und einem garantierten Durchlauf: do-while | 62 |
| 7.4. Zählschleifen: for | 63 |
| 7.5. Eingriffe in den Kontrollfluss: break und continue | 64 |
| 8. Arrays und dynamische Programmierung | 66 |
| 8.1. Arrays | 66 |
| 8.1.1. Syntax-Elemente | 66 |
| 8.1.2. Initialisierung | 68 |
| 8.1.3. Pointer-Arithmetik | 70 |
| 8.1.4. Mehrdimensionale Arrays | 70 |
| 8.2. Dynamische Speicherverwaltung | 73 |
| 8.2.1. Speicher allozieren und wieder freigeben: malloc , calloc und free | 73 |
| 8.2.2. Feldgrößen ändern: realloc | 77 |
| 8.2.3. Heap und Stack | 79 |
| 8.2.4. C++ | 80 |
| 8.2.5. Mehrdimensionale dynamische Arrays | 80 |
| 8.3. C-Strings | 87 |
| 8.3.1. Spezielle Syntax-Elemente | 87 |
| 8.3.2. User-Eingaben | 89 |
| 8.3.3. Formatierte Strings erzeugen: sprintf und snprintf | 91 |
| 8.3.4. Nützliche Funktionen aus der String-Library | 93 |
| 8.4. Speicherbedarf von Datenstrukturen ermitteln: sizeof | 93 |
| 8.5. Debugging-Tool valgrind | 95 |
| 8.6. Variable Length Arrays | 98 |
| 9. Strukturierte Programmierung | 99 |
| 9.1. Scopes | 100 |
| 9.2. Funktionen | 102 |
| 9.2.1. Scopes in bisher bekannten Strukturen | 105 |
| 9.2.2. Forward Declaration | 105 |
| 9.2.3. Übergabe „By Reference“ und „By Value“ | 107 |
| 9.2.4. Datentyp void | 107 |
| 9.2.5. Globale Variablen | 111 |
| 9.2.6. Was als Funktion auslagern | 112 |
| 9.2.7. Rückgabe mehrerer Werte | 113 |
| 9.2.8. Speicherklasse static | 115 |
| 9.3. Optimierung: inline | 116 |
| 9.4. Funktionszeiger | 117 |
| 10. Speicher-Strukturen | 121 |
| 10.1. Sammlungen von Werten: structs | 121 |
| 10.1.1. Direkter Zugriff auf struct -Elemente | 122 |
| 10.1.2. Aliase für Datentypen: typedef | 125 |
| 10.1.3. structs und Pointer | 128 |
| 10.1.4. structs im Speicher | 131 |
| 10.1.5. Operationen mit structs | 132 |
| 10.1.6. Initializer-Syntax | 132 |
| 10.2. Casting-Interface: unions | 134 |
| 10.3. Automatisch numerierte Symbole: enums | 136 |
| 10.4. C++ | 137 |

| | |
|--|------------|
| 11. Modulare Programmierung | 138 |
| 11.1. Trennung von Header- und Modul-Code | 138 |
| 11.2. Speicherklasse extern | 140 |
| 11.3. Funktionen mit eingeschränkter Sichtbarkeit: static | 142 |
| 11.4. Weitere Speicherklassen und Modifier | 143 |
| 12. Anbindung an das Betriebssystem | 145 |
| 12.1. Errorcodes | 145 |
| 12.1.1. Rückgabewert der main | 145 |
| 12.1.2. Programm instantan beenden: exit | 147 |
| 12.1.3. Befehle an die Kommandozeile: system | 147 |
| 12.1.4. Fehlerbeschreibung ermitteln: strerror | 148 |
| 12.2. Kommandozeilenparameter | 148 |
| 12.3. Spezielle Befehle für unixoide Konsolen | 150 |
| 12.3.1. Konsolenbildschirm leeren | 150 |
| 12.3.2. Cursor auf der Konsole platzieren | 151 |
| 12.3.3. Farben setzen | 151 |
| 13. Dateizugriff | 153 |
| 13.1. File-Handle und Zugriffsmodi | 153 |
| 13.2. Text-Zugriff – Lesen und Schreiben in menschlichem Format | 156 |
| 13.3. Binär-Zugriff – Lesen und Schreiben in Binärformat | 158 |
| 13.4. Cursorposition in Dateien: fseek und ftell | 162 |
| 14. Der Präprozessor | 164 |
| 14.1. Konstante Ausdrücke und vordefinierte Symbole | 165 |
| 14.1.1. Konstante Ausdrücke | 165 |
| 14.1.2. Mehrzeilige Symbole | 171 |
| 14.1.3. Vordefinierte Symbole | 171 |
| 14.2. Bedingte Kompilierung | 171 |
| 14.3. Parametrisierte Macros | 173 |
| 14.4. Stringify-Operator # | 176 |
| 14.5. Concatenation-Operator ## | 177 |
| 15. Miscellaneous | 179 |
| 15.1. Zeitpunkte | 179 |
| 15.1.1. UNIX Epoch Time | 179 |
| 15.1.2. Zeitwerte finden und umrechnen | 180 |
| 15.1.3. Genaue Zeitmessung und Ticks | 181 |
| 15.1.4. Kurze Wartezeiten | 182 |
| 15.2. Zufallszahlen | 182 |
| 15.3. nan – not a number | 185 |
| 15.4. atexit | 186 |
| 15.5. Variadische Funktionen | 187 |
| 15.6. Komplexe Zahlen | 188 |
| 15.7. Debugger | 190 |
| 16. Rekursion | 192 |
| 16.1. Funktionen, die sich selbst aufrufen | 192 |
| 16.2. Kommunikation über Rekursionsebenen hinweg | 195 |
| 16.3. Laufzeitverhalten von rekursiven Methoden | 196 |
| 16.4. Beispiel: Rekursives Auflisten der Ordnerstruktur | 197 |
| 16.4.1. Code | 198 |

| | |
|---|------------|
| 16.4.2. Anmerkungen zu <code>showtree</code> | 202 |
| 16.4.3. Anmerkungen zur <code>stringlist</code> | 204 |
| 16.4.4. Anmerkungen zu <code>get_directory_entries</code> | 205 |
| 17. Linked Lists | 206 |
| 17.1. Ausgangslage: Klassische Arrays | 206 |
| 17.2. Verknüpfung mit seinen Nachbarn: Linked Lists | 208 |
| 17.3. Aufbau einer Bibliothek zur Verwaltung von Linked Lists | 210 |
| 17.3.1. Die Datentypen | 210 |
| 17.3.2. Aufbau und Bereinigung der Liste – Constructor und Destructor | 211 |
| 17.3.3. Sprünge durch die Liste | 212 |
| 17.3.4. Löschen und Einfügen in die Liste | 213 |
| 17.3.5. Ausgeben der gesamten Liste | 216 |
| 17.3.6. Der Komplette Header | 216 |
| 17.3.7. Anwendungsbeispiel | 217 |
| 17.4. Einsatzbereiche der Linked List: Vor- und Nachteile | 219 |
| 18. Ausblicke: C++ | 220 |
| 18.1. Default Arguments und Function Overloading | 221 |
| 18.2. Templates | 224 |
| 18.3. Klassen und Objektorientierung | 225 |
| 18.4. Überladene Operatoren | 226 |
| 18.5. Strings in C++ | 227 |
| 18.6. Die Container-Library | 227 |
| Appendices | 229 |
| A. Begriffe | 230 |
| B. Tabellen | 239 |
| B.1. Escape-Sequenzen | 239 |
| B.2. Formatierte Textausgabe | 239 |
| B.3. Formatierte Texteingabe | 242 |
| B.4. UNIX/bash-Kommandos zum Formatieren | 242 |
| B.5. Operator-Hierarchie | 243 |
| B.6. Übersicht Datentypen | 244 |
| B.7. Vordefinierte Präprozessor-Symbole | 245 |
| C. Geschichte | 247 |

1. Wie arbeitet ein Computer?

The good news about computers is that they do what you tell them to do. The bad news is that they do what you tell them to do.

Ted Nelson

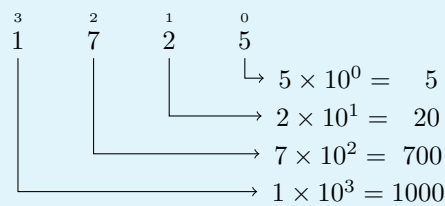
Es ist weitgehend bekannt, dass in Computern nur zwei Zustände existieren – Strom an und Strom aus. Diese Information wird gerne als 1 und 0 ausgedrückt. Aber wie lassen sich mit diesen elementaren Zuständen Programme konstruieren?

1.1. Binärdarstellung von positiven Ganzzahlen und Schriftzeichen

Viele solche an/aus-Informationen können zu größeren Einheiten zusammen geschlossen werden, um so andere Zahlen darzustellen:

Betrachten wir das Dezimalsystem, also unsere „normalen Zahlen“, z. B. die Zahl 1725.

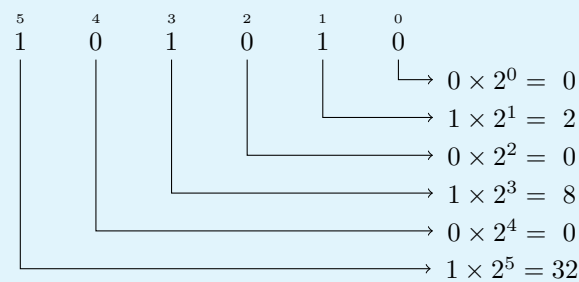
Zerlegung einer Dezimalzahl



Wir beginnen dabei, die einzelnen Ziffern unserer Zahl *von rechts* zu nummerieren und beginnen dabei mit der null. Die „Wertigkeit“ jeder einzelnen Ziffer ergibt sich aus dem Wert der Ziffer und ihrer Position: Sie ist jeweils das Produkt aus Ziffer und einer *Zehnerpotenz*. Die gesamte Zahl ist dann die Summe der einzelnen Wertigkeiten.

Nach demselben Muster funktioniert die Interpretation einer *Binärzahl*. Ziffern (*Bits*) können allerdings nur die Werte 0 oder 1 annehmen. Die Wertigkeit ergibt sich aus der Multiplikation mit einer *Zweierpotenz*. Betrachten wir zum Beispiel die Darstellung der Dezimalzahl $42 = 32 + 8 + 2$ als Binärzahl:

Interpretation der Dezimalzahl 42 als Binärzahl



Texte werden intern als Folge von Zahlen behandelt. Jedem Zeichen wird eine Zahl zugeordnet und mittels einer Tabelle „übersetzt“. Aus technischen Gründen ist die Zahl der darstellbaren Zeichen für einfache Programme auf 256 begrenzt¹. Davon sind die ersten 128 Zuordnungen international genormt; die zweite Hälfte der Tabelle hängt von den Ländereinstellungen am jeweiligen Rechner ab. Siehe Abb. 1.1 für die Zuordnung.

Damit können wir mit den Mitteln der Elektronik bereits (nahezu) beliebig große positive Ganzzahlen

¹In der Vorlesung *Algorithmen und Datenstrukturen* an der Universität Regensburg erfahren Sie, wie Sie diese Schranke umgehen und so z. B. internationale Schriftzeichen in ihren Programmierprojekten verwerten. Sie können sich hierzu auch selbst Wissen zum Stichwort *Unicode* aneignen.

ASCII TABLE

| Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char | Decimal | Hex | Char |
|---------|-----|------------------------|---------|-----|---------|---------|-----|------|---------|-----|-------|
| 0 | 0 | [NULL] | 32 | 20 | [SPACE] | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 1 | [START OF HEADING] | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 2 | [START OF TEXT] | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 3 | [END OF TEXT] | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 4 | [END OF TRANSMISSION] | 36 | 24 | \$ | 68 | 44 | D | 100 | 64 | d |
| 5 | 5 | [ENQUIRY] | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | [ACKNOWLEDGE] | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | [BELL] | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 8 | [BACKSPACE] | 40 | 28 | (| 72 | 48 | H | 104 | 68 | h |
| 9 | 9 | [HORIZONTAL TAB] | 41 | 29 |) | 73 | 49 | I | 105 | 69 | i |
| 10 | A | [LINE FEED] | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | B | [VERTICAL TAB] | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | C | [FORM FEED] | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | D | [CARRIAGE RETURN] | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | E | [SHIFT OUT] | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | F | [SHIFT IN] | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | [DATA LINK ESCAPE] | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | [DEVICE CONTROL 1] | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | [DEVICE CONTROL 2] | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | [DEVICE CONTROL 3] | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | [DEVICE CONTROL 4] | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | [NEGATIVE ACKNOWLEDGE] | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | [SYNCHRONOUS IDLE] | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | [ENG OF TRANS. BLOCK] | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | [CANCEL] | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | [END OF MEDIUM] | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | [SUBSTITUTE] | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | [ESCAPE] | 59 | 3B | ; | 91 | 5B | [| 123 | 7B | { |
| 28 | 1C | [FILE SEPARATOR] | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | |
| 29 | 1D | [GROUP SEPARATOR] | 61 | 3D | = | 93 | 5D |] | 125 | 7D | } |
| 30 | 1E | [RECORD SEPARATOR] | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | [UNIT SEPARATOR] | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | [DEL] |

Abbildung 1.1.: ASCII: American Standard Code for Information Interchange

Quelle: <https://commons.wikimedia.org/wiki/File:ASCII-Table-wide.svg>

und Schriftzeichen darstellen². Wie aber schafft man es, aus Zahlen komplexe Programme zu bauen?

1.2. Von Zahlen zu Befehlen

Die *CPU* (Central Processing Unit) eines Computers ist ein elektronisches Bauteil, in dem eine gewisse Anzahl sehr einfacher Arbeitsschritte „mechanisch“ im Chip umgesetzt sind. Solche Arbeitsschritte sind etwa „addiere zwei Zahlen“ oder „lade einen Wert vom Speicher“. Diese Befehle kann man sich als „durchnummeriert“ denken. Ist 17 beispielsweise die Nummer des Befehls **multipliziere**, so drückt die Zahlenkette 17 21 2 also den Auftrag **multipliziere die Zahlen 21 und 2 miteinander** aus. Genau aus solchen Zahlenketten besteht jedes Computerprogramm. Die Zahlen werden im *Binärformat* gespeichert, also als Kette von 0en und 1en, die auch als Schriftzeichen interpretiert werden könnten. Öffnet man mit einem Textbearbeitungsprogramm eine ausführbare Datei, so zeigt dieses nur einen unverständlichen Zeichensalat an. Für einen Menschen wäre es beschwerlich, eine solche Datei zu lesen; theoretisch ist es aber möglich, mithilfe einer geeigneten Tabelle ausführbare Dateien zu „lesen“.

In den Pioniertagen der Computertechnik musste man tatsächlich die Zuordnung der Zahlen zu den Befehlen kennen, um ein Computerprogramm zu schreiben. Diese Arbeitsweise ist selbstverständlich sehr umständlich und fehleranfällig. Erste Abhilfe schuf hier die Programmiersprache *Assembler*. Jedem CPU-Befehl wurde ein kurzer Text (ein sog. *Mnemonic*) zugeordnet. Befehle wie `mul 21 2` erlauben bereits bedeutend komfortableres Schreiben und Warten von Programmen.

Die Programmierung in Assembler ist anspruchsvoll und erlaubt es kaum, komplexe Projekte umzusetzen³. Aus diesem Grund wurden *Hochsprachen* wie C entwickelt, in denen komplette „Arbeitspakete“ wie „Gib einen Text auf dem Bildschirm aus“ in einem einzelnen Befehl zusammengefasst sind. Damit diese C-Befehle vom Computer verstanden werden, müssen diese natürlich erst wieder in Maschinensprache zurückübersetzt werden, also in eine Folge von Zahlen, die die CPU interpretieren kann. Diese Aufgabe benutzt ein sogenannter *Compiler*, also ein spezielles Computerprogramm⁴.

Ein Vorteil von Hochsprachen ist, dass man eigene Funktionen für komplexe Aufgaben realisieren kann. Häufig wiederkehrende Aufgaben werden in sogenannten *Bibliotheken* oder *Libraries* gesammelt und – oft schon *vorkompiliert* (also bereits in Maschinensprache übersetzt) – anderen Programmierern zur Verfügung gestellt. So existiert in C beispielsweise die *math-library*, in der Operationen wie *finde die Quadratwurzel einer Zahl* umgesetzt werden. Ein sogenannter *Linker* sorgt dafür, dass Code aus eigenen Projekten und vordefinierten Libraries korrekt verknüpft werden.

1.3. Aufruf des Compilers

Für diesen Kurs benutzen wir den *GNU C-Compiler* (`gcc`). Dieses Programm ist sowohl ein Compiler als auch ein Linker. Gehen wir davon aus, im aktuellen Arbeitsverzeichnis liegt eine Datei mit dem Namen `myCode.c` vor. Diese Datei kann kompiliert werden mit dem Kommandozeilen-Befehl:

Kommandozeilen-Befehl: Kompilieren

```
gcc myCode.c
```

²negative Zahlen und Dezimalbrüche werden in Abschnitt 3.1.1 behandelt

³Die erste Generation der Gameboy-Spielereihe *Pokémon* wurde in Assembler programmiert. Obgleich dies eine beachtliche Leistung darstellt sind diese Spiele heute bekannt für die Vielzahl von in diesen enthaltenen *Glitches* (Fehlern im Code, die unbeabsichtigte Effekte auslösen).

⁴Tatsächlich wird häufig erst von der Hochsprache in Assembler übersetzt. Der letzte Schritt – die Übersetzung von Assembler-Code in Maschinensprache – übernimmt ein *Assembler*. In der Praxis sagt man aber einfach „kompilieren“ zur Gesamtheit aller Übersetzungsschritte bis hin zur Maschinensprache.

Der gcc kompiliert mit diesem Befehl die Datei `myCode.c` und erstellt eine ausführbare Datei mit dem Dateinamen `a.out`. Diese kann nun über den Kommandozeilen-Befehl:

Kommandozeilen-Befehl: Kompiliertes Programm ausführen

```
./a.out
```

ausgeführt werden. Besteht der Code aus mehreren *Modulen*⁵, so können diese einfach beim Aufruf des gcc hintereinander durch Leerzeichen getrennt aufgelistet werden. Neben Quellcode-Dateien (Erweiterung `.c`) können auch vorkompilierte „Object Codes“ (Erweiterung `.o`) aufgelistet werden.

Kommandozeilen-Befehl: Kompilieren mehrerer Module

```
gcc myCode.c otherCode.c moreCode.c objectFile.o otherObject.o
```

In jedem Beispiel wird eine Datei `a.out` erstellt (bzw. überschrieben, falls diese bereits existiert). Möchte man, dass das kompilierte Programm einen anderen Namen erhält, so kann die Option `-o [OutputName]` angegeben werden; `[OutputName]` ist dabei der neue Dateiname. Beispiel: die zu erstellende ausführbare Datei soll den Namen `myProgram` tragen:

Kompilieren mit vorgegebenem Dateinamen für das kompilierte Programm

```
gcc -o myProgram myCode.c
```

Man startet dieses Programm dann mit dem Kommando

Kommandozeilen-Befehl: Kompiliertes Programm ausführen

```
./myProgram
```

Während der Entwicklung eines Programmes wird man häufig Fehler machen. Manche dieser Fehler sorgen für Code, der vom Compiler nicht interpretiert werden kann; man spricht u. a. von *Syntaxfehlern*. Dies können beispielsweise fehlende Klammern sein. Enthält der Code solche Fehler, wird das Kompilieren abgebrochen und keine ausführbare Datei erzeugt. Stattdessen zeigt der Compiler Informationen zum Fehler in der Kommandozeile an. (Wir gehen in den folgenden Kapiteln genauer auf diese Rückmeldungen ein).

Andere Code-Elemente können zwar in „validen Maschinencode“ umgesetzt werden, enthalten aber evtl. logische Fehler (sog. *semantische Fehler*). Standardmäßig gibt der Compiler hierzu keine Rückmeldung. Wir können aber mit den Kommandozeilenoptionen⁶ `-Wall`, `-Wimplicit-fallthrough` und `-Wpedantic` einstellen, dass auf solche Strukturen in der Compiler-Ausgabe ebenfalls hingewiesen werden soll. In der Compiler-Ausgabe werden diese dann als *warnings* vermerkt (daher das **W** in `Wall`, `-Wimplicit-fallthrough` und `Wpedantic`). Ich empfehle sehr, diese Optionen zu nutzen und den Code zu verbessern, bis alle Warnungen beseitigt sind; in meinen Projekten haben diese Optionen einige Fehler aufgedeckt und mir viel Zeit erspart.

Ein Aufruf mit diesen Optionen sieht also so aus:

Kompilieren mit Ausgabe aller Warnungen

```
gcc -Wall -Wimplicit-fallthrough -Wpedantic -o myProgram myCode.c
```

⁵d. h. mehrere Dateien; im Detail steckt mehr hinter diesem Begriff, worauf wir in Kapitel 11 näher eingehen werden.

⁶Compiler Optionen werden oftmals auch *Compiler Flags* genannt.

Bibliotheken (z. B. `.o`-Dateien) liegen in der Regel nicht im aktuellen Arbeitsverzeichnis, sondern in einem Standard-Verzeichnis, so dass alle Codes auf der Festplatte diese gleichermaßen nutzen können. Um dem Compiler mitzuteilen, dass eine solche Bibliothek mit in den Linking-Prozess einbezogen werden soll, benutzen wir die Kommandozeilen-Option `-l[library]`, wobei `[library]` der Name der Bibliothek ist, die einbezogen werden soll. Im Kurs wollen wir z. B. desöfteren mathematische Funktionen aus der Mathe-Bibliothek `libm` benutzen. Entsprechend ergänzen wir den Compiler-Aufruf um die Option `-lm`. Das voranstehende `lib` wird vom Compiler automatisch entfernt:

Kompilieren und Linken mit der math-library

```
gcc -Wall -Wimplicit-fallthrough -Wpedantic -o myProgram -lm myCode.c
```

Im Vorwort wurde erwähnt, dass C laufend fortentwickelt wird. Dies hat Auswirkungen auf die Interpretation des Codes; derselbe Code wird von älteren Versionen des Compilers nicht zwangsweise in derselben Art umgesetzt wie von neueren. Der gcc kann sowohl nach älteren als auch neueren Versionen der Sprache C kompilieren. Welche Version der Sprache benutzt werden soll, wird mit der Kommandozeilenoption `-std=[standard]` angegeben. Im Kurs halten wir uns an den Standard aus dem Jahr 2011, und ergänzen den Kommandozeilen-Aufruf entsprechend um `-std=c11`:

Kompilieren mit nach Standard C11

```
gcc -std=c11 -Wall -Wimplicit-fallthrough -Wpedantic -o myProgram myCode.c -lm
```

Neben den gezeigten Optionen existieren noch viele weitere Schalter, über die die Arbeitsweise des Compilers beeinflusst werden können; auf diese soll hier jedoch nicht weiter eingegangen werden. Interessierte KursteilnehmerInnen können mit dem Befehl `man` eine Übersicht aller Optionen mit Erklärungen anzeigen lassen.

```
man-page des gcc
```

```
man gcc
```

Ausflug: C++

Die hier gegebenen Erklärungen und Optionen gelten so auch für C++. Jedoch sollte der Compiler `g++` benutzt werden. Das heißt, an allen Stellen, in denen oben `gcc` stand, muss durch `g++` ersetzt werden. Außerdem muss als Standard dann `-std=c++11` anstelle von `-std=c11` angegeben werden.

Tipp

Da es lästig werden kann, wiederholt dieselben Optionen zu tippen, arbeite ich für kleinere Projekte gerne mit einem kleinen Script wie dem folgenden^a:

Mini-Script `compile` zum komfortablen kompilieren

```
1  OPTIONS="-std=c11 -Wall -Wimplicit-fallthrough -Wpedantic"
2  LIBS="-lm -lcurses"
3
4  INFILE=$1
5  OUTFILE="${INFILE%%.*}.o"
6
7  clear
8
9  rm -f $OUTFILE
10 echo =====
11 echo attempting to compile
12 echo gcc $OPTIONS -o $OUTFILE $INFILE $LIBS
13 echo .....
14 echo ""
15
16 gcc $OPTIONS -o $OUTFILE $INFILE $LIBS
17
18 echo ""
19 echo .....
20 echo Compilation successfull.
21 echo .....
22 echo ""
23 ./OUTFILE
```

Dieses Script kann z. B. in einer Textdatei mit Namen `compile` gespeichert und als ausführbares Script markiert werden. Letzteres geschieht über einen Kommandozeilen-Befehl:

Kommandozeile: Script ausführbar machen

```
chmod +x ./compile
```

Mit diesen Schritten reicht es, nur noch

Kommandozeile: Script ausführen

```
./compile myCode.c
```

eingzugeben. Der Code automatisch mit den besprochenen Optionen kompiliert und direkt ausgeführt, sofern der Code erfolgreich kompiliert werden konnte.

^aFür größere Projekte gibt es Mechanismen, die hier nicht angesprochen werden können. Interessierte KursteilnehmerInnen können die DozentInnen bei Interesse auf *makefiles* ansprechen.

2. Ein erstes Programm

The secret to getting ahead is getting started.

Mark Twain

In diesem Kapitel werden wir zunächst ein sogenanntes *Hello World* besprechen: Dieses Programm gibt einen kurzen Text auf dem Bildschirm aus und enthält sonst „nur“ den Grundaufbau eines C-Programms¹. Im Weiteren betrachten wir auch Möglichkeiten, die Sprache C für einfache Rechnungen zu verwenden.

2.1. Hello World

Die folgenden Zeilen zeigen ein Programm, das den Text **Hello World!** auf dem Bildschirm ausgibt.

Code

```
1  #include <stdio.h>
2
3  int main() {
4      printf("Hello World!\n");
5  }
```

Betrachten wir den Code Zeile für Zeile.

Es wurde bereits angesprochen, dass häufig wiederkehrende Aufgaben in Bibliotheken ausgelagert werden können und so in vielen Projekten zur Verfügung stehen. Eine solche Bibliothek stellt Routinen zur Ein- und Ausgabe bereit und trägt den Namen `stdio` (*standard in/output*). Um die Routinen im Code nutzbar zu machen, müssen diese einmalig *deklariert* werden. Mit dem Befehl `#include` wird die Datei `stdio.h` in den Code mit eingebunden². Diese Datei wird ein *Header* genannt und enthält alle notwendigen Deklarationen für die Textein- und Ausgabe. In Kapitel 11 werden wir hierauf zurück kommen. Für den Moment reicht es zu wissen, dass die `#include`-Zeile notwendig ist, um Ein/Ausgabe-Befehle benutzen zu können.

Programme sind in funktionelle Einheiten („Funktionen“) gegliedert, die von {geschweiften Klammern} eingefasst werden. Zeile 3 deklariert eine solche Funktion mit Namen `main`, die bis Zeile 5 reicht. Ein Programm kann beliebig viele solcher Funktionen besitzen, die sich jeweils im Namen unterscheiden müssen. Groß- und Kleinschreibung wird beachtet, `main` ist also von `Main` verschieden. Die Funktion `main` ist das Hauptprogramm; nur dieses wird automatisch ausgeführt. Auch hierzu werden wir in Kapitel 9 vertiefen (insbesondere die Bedeutung von `int` und den Klammern). Für unsere ersten Programme können wir diese Zeile einfach kopieren.

Zeile 4 enthält den Befehl `printf`. Dieser Befehl gibt Text auf dem Bildschirm aus. Das `f` in `printf` steht dabei für *formatted*. In Abschnitt 2.3 werden wir einige Details hierzu kennenlernen. Der auszugebende Text wird als *Parameter* übergeben, d. h. in Klammern hinter den Befehl selbst gestellt. Hier ist also der

¹Der Begriff *Hello World* wird auch im Kontext anderer Sprachen benutzt und bedeutet dort dasselbe.

²Die Datei befindet sich in einem Standard-Verzeichnis, das bei der Installation des gcc angelegt und voreingestellt wird.

Parameter zu `printf` der Ausdruck `"Hello World!\n"`. Um Texte von restlichem Code abzugrenzen muss der Text in "doppelte Anführungszeichen" gesetzt werden. Dieser Text enthält auch einen abschließenden Zeilenumbruch, codiert als `\n` (n für *new line*).

Wichtig ist das Semikolon (;) am Ende der Zeile – hiermit wird ein Befehl abgeschlossen. Jeder Befehl muss mit einem Semikolon abgeschlossen werden. Fehlt dieses, gibt der Compiler eine Fehlermeldung aus und bricht die Kompilation ab. Mehrere Befehle dürfen in derselben Zeile stehen, solange die Befehle von einem Semikolon voneinander abgetrennt sind.

Wir können beliebig Leerzeichen, Tabulatoren oder Zeilenumbrüche in den Code einfügen, um diesem eine optische Struktur zu geben. Nur die *Schlüsselwörter* (z. B. Befehle, bzw. die „Worte“ des Codes) dürfen nicht zertrennt werden. Einrückungen werden verwendet, um „Hierarchieebenen“ zu verdeutlichen und machen den Code besser les- und wartbar. Ich empfehle, gleich von Anfang an Einrückungen wie in diesem Script gezeigt zu verwenden. Vom Compiler werden Leerzeilen komplett ignoriert und können zur Abgrenzung von Codeblöcken beliebig eingefügt werden.

Der folgende Code erzeugt dasselbe Ergebnis wie oben und soll nur verdeutlichen, wo Leerzeichen eingefügt werden können.

```
Code
1  #include    <stdio.h>
2  int    main    (    )    {
3      printf    (    "Hello World!\n"    )    ;
4  }
```

Code wird vom Compiler von oben nach unten gelesen und bearbeitet. Die Reihenfolge der einzelnen Anweisungen ist also von großer Bedeutung. Verschiebt man beispielsweise die `#include`-Zeile an das Ende des Codes, so erhält man Fehlermeldungen, da der Befehl `printf` deklariert werden muss (durch Einbinden der Datei `stdio.h`) *bevor* er zum ersten Mal aufgerufen wird.

2.2. Variablen, Datentypen, Operatoren

Nützliche Programme entwickeln sich über die Zeit, berechnen also z. B. ein Ergebnis oder reagieren auf Benutzereingaben. Um dies zu ermöglichen existieren in C *Variablen*, also Symbole, die einen Wert beschreiben. Dieser kann sich über die *Laufzeit* eines Programmes ändern.

2.2.1. Variablen

Eine Variable hat einen Namen, einen *Datentyp* und einen Wert.

Der Name einer Variablen ist eine beliebige Kombination aus *alphanumerischen Zeichen*³. Er muss mit einem Buchstaben beginnen, darf keine Leerzeichen enthalten und kann bis zu 40 Zeichen lang sein. Auch hier wird zwischen Groß- und Kleinschreibung unterschieden. Variablennamen müssen *eindeutig* sein, d. h. es darf keine zwei Variablen mit demselben Namen geben. Auch Funktionsnamen gelten als „verbraucht“; wir können keine Variable `main` nennen.

Anfänger neigen dazu kurze, nichtssagende Variablennamen zu vergeben. Um Programme gut les- und wartbar zu machen (und um somit Fehler zu vermeiden) sollte der Variablenname bereits ihre Funktion im Code angeben. Statt `a`, `b`, `c` also lieber `elementCount`, `averageNote`, `currentIndex`.

³Die Buchstaben a-z, A-Z, die Ziffern 0-9 sowie der Unterstrich (_), jedoch keine Umlaute, Sonderzeichen oder das scharfe ß.

Unter *Datentyp* versteht man die Art der Information, die in der Variablen gespeichert wird, z. B. Ganzzahl, Kommazahl oder Text. Dies ist notwendig, da der Computer intern nur Einsen und Nullen speichern kann. Ohne Kontext ist es also nicht möglich, zu erkennen, wie eine Bit-Folge zu interpretieren ist. Dieser Kontext ist durch Zuweisung eines Datentyps gegeben.

Der Wert einer Variablen kann schließlich alles sein, was ihrem Datentypen entspricht; eine Variable kann z. B. eine einzelne Zahl oder ein ganzes Bild speichern.

2.2.2. Variablendeklaration

Um Variablen zu benutzen müssen diese zuerst *deklariert* werden. Dies geschieht in der Form

Syntax: Variablen Deklarieren

```
Datentyp VariablenName;
```

Mehrere Variablen können in einem Schritt deklariert werden, indem man sie durch Kommata voneinander abtrennt.

Für den Moment wollen wir uns auf die Datentypen **int** (Ganzzahlen) und **double** (Kommazahlen) beschränken. Mit diesen können wir also folgende Variablen deklarieren:

Beispiel: Deklaration von mehreren Variablen

```
1  int main () {
2      int    Ganzzahl;
3      double Kommazahl, andereZahl;
4  }
```

Variablen werden also *innerhalb von Funktionen* deklariert⁴.

Ihnen mag die Ähnlichkeit zwischen der Deklaration von Ganzzahl-Variablen und der Funktion **main** aufgefallen sein. Dies ist kein Zufall. Tatsächlich ist formell die Funktion **main** die Vorschrift einen **int**-Wert zu berechnen. Das Symbol **main** ist eine Variable, die auf diese Vorschrift verweist. Hierzu mehr in Kapitel 12.

2.2.3. Wertezuweisung

Werte können über den *Operator* = zugewiesen werden:

Beispiel: Wertzuweisung

```
1  int main () {
2      int    Ganzzahl;
3      double Kommazahl, andereZahl;
4
5      Ganzzahl    = 7;
6      Ganzzahl    = 8;
```

⁴In Abschnitt 9.2.5 werden wir hierzu mehr hören; für den Moment beschränken wir uns auf den Fall, dass alle Variablen am Anfang der Funktion **main** deklariert werden.

```

7      Kommazahl  = 4.3;
8      andereZahl = 5.0;
9  }

```

In Zeile 5 wird also der Variablen **Ganzzahl** der Wert 7 zugewiesen. In der folgenden Zeile 6 überschreiben wir diesen Wert mit dem Wert 8.

In den Zeilen 7 und 8 erhalten die beiden **double**-Variablen ebenfalls Werte. Wir bemerken, dass hier ein *Dezimalpunkt* gesetzt wird, also kein Komma.

Deklaration und Wertzuweisung können auch in einem einzigen Schritt geschehen:

Beispiel: Kombinierte Deklaration und Wertzuweisung

```

1  int main () {
2      int    Ganzzahl    = 8;
3      double Kommazahl  = 4.3,
4          andereZahl = 5.0;
5  }

```

Nicht-Initialisierte Variablen

Wenn eine Variable deklariert wird, sorgt der Compiler lediglich dafür, dass genug Platz im Speicher reserviert wird, um diesen Wert zu erfassen. Im Allgemeinen wird aber kein bestimmter „Startwert“ festgelegt. Die neue Variable startet mit dem Wert, den das Bitmuster an der Stelle festlegt, an der Platz für die Variable reserviert wurde. Dies ist ein zufälliger Wert! Programme, die mit einem solchen Zufallswert weiterarbeiten haben generell *undefiniertes* Verhalten und können abstürzen oder sogar Schaden anrichten. Es ist daher also ratsam, allen Variablen bei der Deklaration gleich einen sinnvollen oder sicheren Startwert zuzuweisen.

Fließkommazahlen ohne Nachkommastelle

In der Mathematik gilt $5 = 5.0$. Ein Computer arbeitet zwar nach mathematischen Prinzipien; streng genommen gilt diese Gleichheit für den Prozessor aber nicht. Wie wir in Abschnitt 3.1.1 sehen werden kann dieselbe (mathematische) Zahl auf verschiedene Weisen binär dargestellt werden. Insbesondere zwischen Ganzzahlen und Kommazahlen bestehen große Unterschiede in der Binär-Darstellung. Der gcc kann zwar Umwandlungen automatisch durchführen, und würde auch

Beispiel: Implizite Typumwandlung von Ganzzahl zu Fließkommazahl

```

1  int main () {
2      double Kommazahl = 5;
3  }

```

„korrekt“ umsetzen. Hier wird versucht, die *Ganzzahl* 5 in der **double**-Variablen **Kommazahl** zu speichern. Da dies direkt nicht möglich ist, wandelt der gcc diese Ganzzahl zuerst in die Fließkommazahl 5.0 um und speichert dann diese. Eine solche *implizite Typumwandlung* kann aber unbeabsichtigte Nebeneffekte haben und sollte daher vermieden werden. Als Beispiel für solche Nebeneffekte möchte ich Ihnen das folgende Beispiel geben:

Beispiel: Implizite Typumwandlung von Fließkommazahl zu Ganzzahl

```
1  int main () {  
2      int Ganzzahl = 5.5;  
3  }
```

Analog zum Beispiel oben wird nun implizit die Zahl 5.5 zu einer Ganzzahl konvertiert. Natürlich ist Ihnen bewusst, dass es keine ganze Zahl gibt, die gleich 5.5 ist. Es wird also gerundet. Im Gegensatz zur menschlichen Intuition rundet der gcc hier aber *ab*. Außerdem können die „fehlenden“ 0.5 das Verhalten des Programms in der Laufzeit gegenüber der Erwartung verändern. Damit der Datentyp (und damit das Verhalten des Programms) immer ersichtlich ist, sollten daher „ganzzahlige Fließkommawerte“ als Kommawert (z. B. 5.0) gesetzt werden.

2.2.4. Rechnen mit Variablen – Operatoren

Mit Variablen kann auch gerechnet werden. Hierzu stehen 6 „Grundrechenarten“ zur Verfügung (siehe Tabelle 2.1). Mehrere solche Rechenoperationen können hintereinander geschaltet werden. Es gilt wie üblich Punkt-vor-Strich; Klammern können gesetzt werden um eine andere Reihenfolge festzulegen.

| Operation | Zeichen | Operation | Zeichen |
|-------------|---------|------------------------------|---------|
| Addition | + | Multiplikation | * |
| Subtraktion | - | Division | / |
| Negation | - | Rest der Division („Modulo“) | % |

Tabelle 2.1.: Rechenoperatoren in C

Beispiel: Rechnen mit Variablen

```
1  int main () {  
2      int    Ganzzahl;  
3      double Kommazahl, andereZahl;  
4  
5      Ganzzahl  = 7;  
6      Ganzzahl  = Ganzzahl + 8;  
7  
8      Kommazahl = 4.3;  
9      andereZahl = 3.0 * (-Kommazahl + 7.9) / 2.0;  
10 }
```

Wichtig ist, dass Variablen bereits deklariert sind *bevor* auf sie verwiesen wird, d. h. bevor sie in anderen Ausdrücken vorkommen. Folgendes Beispiel funktioniert daher nicht:

Beispiel: Fehlerhafter Code, Variablen zu spät deklariert

```
1  int main () {  
2      Ganzzahl = 7;  
3      int Ganzzahl;  
4  }
```

und erzeugt die folgende Fehlermeldung:

Fehlermeldungen des gcc

```
myProgram.c: In function 'main':
myProgram.c:2:4: error: 'Ganzzahl' undeclared (first use in this
function)
    Ganzzahl = 7;
    ~~~~~
myProgram.c:2:4: note: each undeclared identifier is reported only once
for each function it appears in
```

Jede Fehlermeldung beginnt mit einem Verweis auf die fehlerhafte Datei und die Funktion darin, in der das Problem aufgetreten ist (myProgram.c: In function 'main:'). In der folgenden Zeile der Fehlerausgabe wird genauer spezifiziert in welcher Zeile, und an welcher Stelle der Fehler auftrat (myProgram.c:2:4, also Zeile 2, Spalte 4) und welcher Fehler gefunden wurde ('Ganzzahl' undeclared). Schließlich gibt der Compiler noch die relevante Stelle aus und markiert diese erfreulicherweise.

Zur Division mit Ganzzahl- und Fließkomma-Werten

Der Datentyp einer Division hängt vom Datentyp von Divisor und Dividend ab. Ist einer der beiden ein Fließkommawert, so wird auch das Ergebnis als Fließkommawert berechnet. Sind beide Werte Ganzzahlen, wird auch das Ergebnis als Ganzzahl berechnet und gegebenenfalls abgerundet. Betrachten wir folgendes Beispiel:

Beispiel: Division von Ganzzahlen und Fließkommazahlen

```
1  double x = 3 / 2 ,
2      y = 3.0 / 2.0,
3      z = 3.0 / 2 ,
4      w = 3 / 2.0;
```

Die Variable `x` geht aus der Division zweier Ganzzahlen hervor, und wird damit zu 1.0 abgerundet. Bei der Berechnung von `y` hingegen gehen Fließkommazahlen in die Division ein; somit wird das Ergebnis nicht gerundet und als 1.5 gespeichert. Dasselbe gilt für `z` und `w`, da zumindest *eine* der an der Division beteiligten Zahlen eine Fließkommazahl war.

2.2.5. Shorthands

Häufig soll sich der Wert einer Variablen *inkrementiert* („hochgezählt“) werden. Dies kann codiert werden als:

Beispiel: Inkrementieren einer Variablen i

```
1  int main () {
2      int i = 5;
3      i = i + 1;  // i hat jetzt den Wert 6
4  }
```

Daneben gibt es auch den *Shorthand* (Kurzform) `variable++`:

Beispiel: Inkrementieren einer Variablen i mit Shorthand ++

```
1  int main () {
2      int i = 5;
3      i++;          // i hat jetzt den Wert 6
4  }
```

In beiden Beispielen wird der Wert von `i` in Zeile 3 um 1 erhöht.

Analog dazu existiert der *Dekrement*-Operator `variable--`, der den Wert von `variable` um 1 verringert.

Weiter existieren der Operator-Shorthand `variable += expression`. Zu `variable` wird der Wert von `expression` hinzugezählt und anschließend in `variable` gespeichert. `expression` darf dabei ein beliebig komplexer Ausdruck sein:

Beispiel: Inkrementieren einer Variablen i mit Shorthand +=

```
1  int main () {
2      int i = 5,
3          j = 2;
4      i += 3 * j + 1;    // i = i + 3 * j + 1
5  }
```

Hier wird zunächst der Ausdruck `3 * j + 1` ausgewertet zu 7; als nächster Schritt wird die Summe `i + 7` berechnet, und diese dann in `i` gespeichert. Die Variable `i` hat also nach Zeile 4 den Wert 12.

Analog existieren auch die Shorthands `--`, `*=`, `/=` und `%=`.

Inkrement und Dekrement: Prefix und Postfix

Die Operatoren `++` und `--` können sowohl *vor* als auch *hinter* einen Ausdruck gesetzt werden (wir sprechen von *Prefix* und *Postfix*). Für sich alleine ist in beiden Fällen der Effekt derselbe – der Wert des Ausdrucks wird um 1 erhöht bzw. verringert. Kombiniert mit anderen Operationen ergeben sich aber Unterschiede. Betrachten wir folgendes Beispiel:

Beispiel: Unterschiede bei Prefix- und Postfix-Inkrement

```
1  int main () {
2      int i, j;
3
4      // Postfix-Form
5      i = 5;
6      j = i++;          // i = 6, j = 5.
7
8      // Prefix-Form
9      i = 5;
10     j = ++i;           // i = 6, j = 6.
11 }
```

In der Postfix-Form (Zeile 6) wird zuerst der Wert von `i` gelesen und der Variablen `j` zugewiesen; danach wird `i` inkrementiert.

Bei der Prefix-Form (Zeile 10) dagegen wird zuerst `i` inkrementiert und danach der Wert von `i` in `j` gespeichert.

Um den Code leicht les- und wartbar zu halten, empfehle ich, alle gedanklichen Schritte in abgeschlossene Befehle zu setzen. Äquivalent zum obigen Code ist der folgende, intuitiv leichter verständliche Code:

Beispiel: Äquivalente Codes

```
1  int main () {
2      int i, j;
3
4      // zur Postfix-Form
5      i = 5;
6      j = i;
7      i++;                // i = 6, j = 5.
8
9      // zur Prefix-Form
10     i = 5;
11     i++;
12     j = i;                // i = 6, j = 6.
13 }
```

2.3. Formatierte Ausgabe

Wir wollen nun auch die berechneten Ergebnisse in Erfahrung bringen. Dazu können wir den bereits bekannten Befehl `printf` benutzen. Betrachten wir folgendes Beispiel:

Beispiel: Ausgabe einer Berechnung mit `printf`

```
1  #include <stdio.h>
2
3  int main () {
4      double result = 5.0 / 7.0;
5
6      printf("5.0 / 7.0 = %lf\n", result);
7  }
```

Die Ausgabe lautet:

Ausgabe einer Berechnung mit `printf`

```
5.0 / 7.0 = 0.714286
```

Wie oben besprochen wird zuerst in Zeile 4 das Ergebnis der Division zweier Fließkommazahlen berechnet und in der Variablen `result` gespeichert. Zur Ausgabe auf dem Bildschirm übergeben wir dem Befehl `printf` jetzt *zwei* Argumente:

Das erste Argument wird *Format-String* genannt und ist in unserem Beispiel der Anteil:

```
"5.0 / 7.0 = %lf\n".
```

Ein Format-String ist eine Zeichenkette die *Platzhalter* enthalten kann. Diese beginnen mit dem Zeichen `%` gefolgt von einem oder mehreren Zeichen, die beschreiben, wofür Platz freigehalten werden soll. In

diesem Fall steht der Platzhalter `%lf`, was für eine **double**-Zahl steht. Diese Zahl wird hinter dem Format-String genannt und von diesem durch ein Komma abgetrennt. Weitere Platzhalter-Symbole sind in Tabelle B.2 aufgelistet.

In einem Format-String können beliebig viele Platzhalter stehen; für jeden Platzhalter muss eine entsprechende Einsetzung hinter dem Formatstring selbst genannt werden.

Beispiel: Ausgabe mehrerer Werte mit `printf`

```
1  #include <stdio.h>
2
3  int main () {
4      double f = 3.14;
5      int    g = 42;
6
7      printf(
8          "Ganzzahl i = %d, Fließkommazahl d = %lf, weitere Zahl = %d",
9              g,                      f,                      99);
10 }
```

Werden hinter dem Formatstring zu viele oder zu wenige Werte angegeben, so meldet der Compiler eine Warnung:

Warnung bei zu wenigen Argumenten für den Formatstring

```
myProgram.c: In function 'main':
myProgram.c:4:12: warning: format '%i' expects a matching 'int' argument
[-Wformat=]
    printf("%i\n");
           ~^
```

Das Programm wird jedoch „erfolgreich“ kompiliert und ausgeführt. Anstelle des Platzhalters wird ein zufälliger Wert vom Speicher eingesetzt⁵.

Ähnliches stellen wir fest, wenn wir das folgende Beispiel betrachten:

Beispiel: Ausgabe mehrerer Werte mit `printf`

```
1  #include <stdio.h>
2
3  int main () {
4      printf("%d\n", 1.5);
5  }
```

Der Formatstring `%d` beschreibt die Ausgabe einer *Ganzzahl*; als Wert für diesen Platzhalter wird aber eine Fließkommazahl angeboten. Die Compiler-Warnung:

⁵Bugs dieser Art sind leider häufig. Der im Jahr 2014 entdeckte *Heartbleed-Bug* in der OpenSSL-Library geht auf einen ähnlichen Mechanismus zurück. Hacker konnten „fehlerhafte Anfragen“ an einen Server schicken und so den Speicherinhalt des Servers auslesen – und damit z. B. Zugangsdaten der User auslesen. Compiler-Warnungen sollten also sehr ernst genommen werden.

Warnung bei falscher Zuordnung Platzhalter/Datensatz

```
myProgram.c: In function 'main':
myProgram.c:4:12: warning: format '%d' expects argument of type 'int', but
argument 2 has type 'double' [-Wformat=]
    printf("%d\n", 1.5);
           ~^
           %f
```

weist darauf hin, dass die Ausgabe mit `%d` für Fließkommawerte ungeeignet ist (und weist auf die Möglichkeit hin, mit `%f` auszugeben). Trotz dieses Fehlers kann das Programm umgesetzt und erfolgreich ausgeführt werden. Angezeigt wird aber nicht 1.5, sondern -2102427368. Diese Zahl ergibt sich, wenn das Bitmuster der Zahl 1.5 als Ganzzahl interpretiert wird⁶.

Wir hatten zuvor schon `\n` als Umschreibung des Zeilenumbruchs kennengelernt. Diese Umschreibung bestimmter Zeichen nennen wir *Escape-Sequence*. Solche Escape-Sequenzen werden benutzt um Zeichen einzufügen, die Teil der C-Syntax sind und sonst nur schwer auf dem Bildschirm ausgegeben werden können. In diesem Kurs werden wir `\n` (Zeilenumbruch), `\\` (Backslash) und `\"` (Anführungszeichen) häufiger sehen.

In den Tabellen B.2 und B.3 im Anhang sind die wichtigsten Codes für Platzhalter und Escape-Sequenzen aufgelistet.

Formatstrings legen nicht nur den *Datentyp* der Ausgabe fest. Zusätzlich kann eine Information hinterlegt werden, wie viel Platz auf dem Bildschirm benutzt werden soll. Auf diese Weise können tabellarische Ausgaben stattfinden. Zu diesem Zweck setzt man zwischen das `%` und dem Platzhalter-Symbol (z. B. `d` für Ganzzahl) eine Zahl:

Beispiel: Format-String mit Platzangabe (1)

```
1  #include <stdio.h>
2
3  int main () {
4      printf("Text %3d\n", 1);
5      printf("Text %3d\n", 10);
6      printf("Text %3d\n", 100);
7      printf("Text %3d\n", 1000);
8  }
```

Ausgabebeispiel: Format-String mit Platzangabe (1)

```
Text  1
Text 10
Text 100
Text 1000
```

Sie sehen also dass die ersten drei Zeilen rechtsbündig ausgegeben. Dies liegt daran, dass mit `%3d` drei Textzeichen für einen Ganzzahl-Wert vorgesehen werden. Die vierte Zeile enthält eine Zahl, die länger ist als die vorgesehenen drei Zeichen. Daher erfolgt die Ausgabe, als wäre keine Zahl eingefügt.

Es gibt auch die Möglichkeit, Platz linksbündig zu reservieren. In diesem Fall gibt man negative Zahlen an:

⁶Genauer: Die ersten 32 bit des Bitmusters; 1.5 ist eine double-Zahl und damit 64 bit breit. In jedem Fall ist diese Ausgabe „unsinnig“.

Beispiel: Format-String mit Platzangabe (2)

```
1  #include <stdio.h>
2
3  int main () {
4      printf("Text %-3d Text\n", 1);
5      printf("Text %-3d Text\n", 10);
6      printf("Text %-3d Text\n", 100);
7      printf("Text %-3d Text\n", 1000);
8  }
```

Ausgabebeispiel: Format-String mit Platzangabe (2)

```
Text 1   Text
Text 10  Text
Text 100 Text
Text 1000 Text
```

Bei Fließkommazahlen können wir zusätzlich zum Gesamtplatz für die Ausgabe auch angeben, wie viele Stellen hinter dem Komma angegeben werden sollen. Die Ausgabe wird dabei automatisch gerundet, wenn weniger Nachkommastellen angegeben werden als die auszugebende Zahl:

Beispiel: Format-String mit Platzangabe (3)

```
1  #include <stdio.h>
2
3  int main () {
4      printf("Text %5.3lf Text\n", 1.5);
5      printf("Text %5.3lf Text\n", 1.555);
6      printf("Text %5.3lf Text\n", 1.5555);
7      printf("Text %5.3lf Text\n", 15.0);
8  }
```

Ausgabebeispiel: Format-String mit Platzangabe (3)

```
Text 1.500 Text
Text 1.555 Text
Text 1.556 Text
Text 15.000 Text
```

2.4. Kommentare

Im Optimalfall ist Code selbsterklärend. „Sprechende“ Variablennamen und gute Struktur erlauben, viele Zusammenhänge intuitiv zu erfassen. Dennoch ist es oft notwendig (und dann auch guter Stil), Programme mit Kommentaren zu versehen.

Kommentare können auf zwei Arten gesetzt werden:

- Mehrzeilige Kommentare beginnen mit `/*` und enden mit `*/`. Achtung: Kommentare können nicht „verschachtelt“ werden!

- Einzeilige Kommentare werden durch zwei *forward slashes* `//` eingeleitet. Sie enden mit dem nächsten Zeilenumbruch⁷.

Ich empfehle, auch bei Kommentaren keine länderspezifischen Zeichen zu verwenden, also z.B. keine Umlaute (äöü). Am besten verwenden Sie nur Zeichen nach ASCII Tabelle Abb. 1.1.

Beispiel: Kommentare

```
1  int main () {
2      int i = 0;
3      i++; // Inkrement-Shorthand
4      // Das ist noch ein einzeiliger Kommentar.
5
6      /* Kommentar der sich über mehrere Zeilen erstreckt.
7       * Oft setzt man weitere Sternchen an den Zeilenanfang, dies
8       * dient aber nur optischen Zwecken und kann
9       * nach belieben ausgelassen werden.
10     */
11
12     /* "Stern-Form" auch für einzeilige Kommentare */
13
14     /* Ein weiterer Kommentar
15      * /* FEHLER -- keine verschachtelten Kommentare */
16      */
17 }
```

⁷Diese Form der Kommentare ist erst seit dem Standard C99 aus dem Jahr 1999 zulässig; davor existierte nur die „Sternchen-Form“.

2.5. Hello World in C++

Ausflug: C++

Die hier gegebenen Erklärungen und Optionen gelten so auch für C++. Das bedeutet, dass der oben gezeigte Code auch in C++ so kompiliert werden kann und dasselbe Ergebnis erzeugt. Jedoch steht hier für die Textausgabe auch das Konzept *Streams* zur Verfügung. Betrachten wir den folgenden Code:

Beispiel: Hello World in C++

```
1  #include <iostream>
2
3  int main() {
4      std::cout << "Hello world!" << std::endl;
5  }
```

Wie schon zuvor wird mit `#include` ein Header geladen (Achtung: Hier keine Erweiterung `.h!`). In diesem Fall ist es der Header, der Input- und Output-Streams definiert. Ein *Stream* ist ein Programm-Interface, das Daten in verschiedenen Formaten annehmen und verarbeiten kann.

Die Deklaration des Hauptprogramms `main` verläuft völlig analog zur Form in C.

`std::cout` ist der „Ausgabe-Stream“. Mit dem Operator `<<` können Datenobjekte an den Stream übergeben werden. Mehrere Datenobjekte können „hintereinander“ in den Stream geschickt werden. So wird hier nach dem Text `Hello World!` ein Zeilenumbruch `std::endl` an den Stream übergeben.

Platzhalter sind bei der Arbeit mit `std::cout` nicht notwendig.

Um weitere Details für die Ausgabe festzulegen siehe:

- <https://en.cppreference.com/w/cpp/io/manip/setw> – Gesamtbreite für die Ausgabe einer Zahl
- <https://en.cppreference.com/w/cpp/io/manip/setprecision> – Anzahl der Nachkommastellen
- <https://en.cppreference.com/w/cpp/io/manip/setiosflags> – Ausgabeformat (exponentialschreibweise, hexadezimal, ...)

Siehe außerdem <https://en.cppreference.com/w/cpp/io/cout>

3. Von der Information zum Bit-muster

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

James Whitcomb Riley

Damit ein Prozessor eine Information verarbeiten kann, muss diese in Form von Ketten aus 1 und 0 vorliegen – also in *binärer* Form. Aus einer Reihe von Bits geht jedoch nicht hervor, ob diese nun als Zahl, Text, Bild, ... interpretiert werden sollen¹. In C (und den damit verwandten Sprachen) muss daher zu jeder Information ein *Datentyp* mit angegeben werden².

Wir wollen uns hier vertieft mit der Datenstruktur von Zahlen im Speicher beschäftigen, um im weiteren die Arbeitsweise vieler Befehle genau zu verstehen.

3.1. Daten im Speicher

3.1.1. Wertemenge und Vorzeichen bei Zahlen

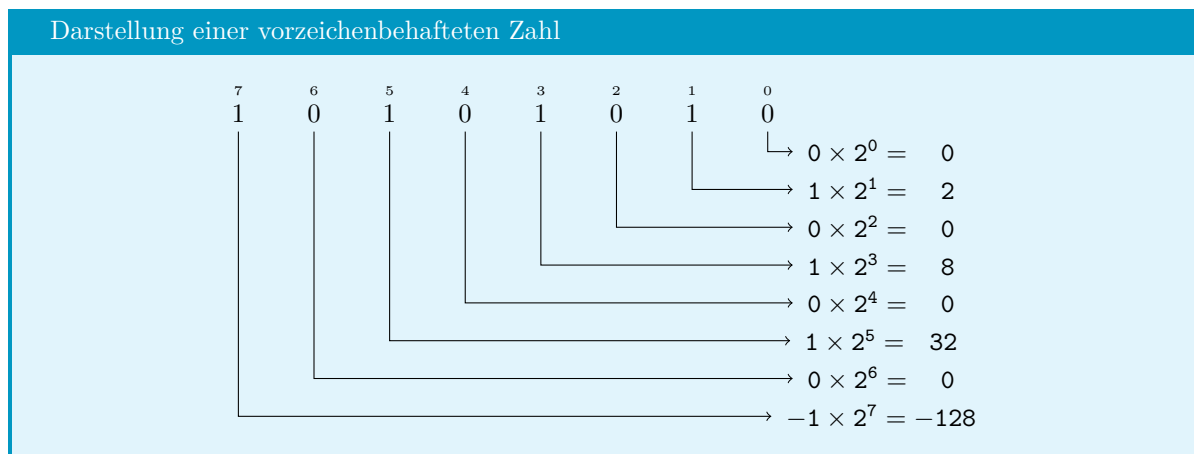
In Abschnitt 1.1 haben wir bereits gesehen wie positive Ganzzahlen und Schriftzeichen als binäre Werte dargestellt werden können. Ein Prozessor kann i.d.R. jedoch nur Einheiten von je 8 Bits (also ganze *Bytes*) bzw. $2^n \cdot 8$ Bits behandeln. Eine Gruppe aus 8 Bits kann einen von $2^8 = 256$ Werten darstellen, also beispielsweise den Zahlenbereich von 0 bis 255. Soll mit größeren Werte gearbeitet werden, müssen mehrere Bytes zu einer Einheit zusammengefasst werden. Zwei Bytes (16 Bits) erlauben also $2^{16} = 65\,536$ verschiedene Werte, bei 4 Bytes (32 Bits) sind dies bereits $2^{32} = 4\,294\,967\,296$ verschiedene Werte. Natürlich muss dem Compiler mitgeteilt werden, wie viele Bytes zu einer Einheit zusammengefasst werden. Dies geschieht über den *Datentyp*. Wir haben bereits den Datentyp `int` kennengelernt. Eine Variable vom Typ `int` ist eine Ganzzahl, deren genaue Größe von der *Architektur des Zielsystems* abhängt, also von der Art des Rechners, auf dem der Compiler läuft. Der Compiler entscheidet hier also, um ein möglichst effizient laufendes Programm zu erstellen. Üblicherweise werden `ints` als 32-bit-Variablen umgesetzt; für bestimmte Microcontroller oder ältere Prozessoren kann dies aber auch als 16-bit-Wert interpretiert werden. Neben `int` gibt es außerdem noch `short int` (16 bit), `char` (8 bit), `long int` (32 bit, auch auf älteren Prozessoren), und `long long int` (64 bit). Wo es wichtig ist, die *Registerbreite* einer Ganzzahl-Variable exakt zu kennen, stehen noch andere Datentypen zur Verfügung – Siehe die Tabellen B.8 und B.9 im Anhang.

Negative Zahlen werden durch die Behandlung des Bits mit der höchsten Wertigkeit als *Vorzeichen-Bits* dargestellt. Das bedeutet, dass ein Bit die Information positiv/negativ enthält anstatt zum normalen Teil der Zahl zu gehören. Um zu vermeiden, dass es eine „positive Null“ gibt, die verschieden von der

¹Da jede digitalisierbare Information auch als Zahl interpretiert werden kann, kommt es zu der bizarren Situation, dass manche (sehr große) Zahlen urheberrechtlich geschützte Werke beschreiben. Insbesondere ist es möglich, solche Zahlen als Primzahlen zu konstruieren. Diese Zahlen werden in Fachkreisen *Illegale Primzahlen* genannt. Ob die gefundenen Primzahlen tatsächlich als illegal gelten, wurde bislang nicht vor Gericht verhandelt.

²Andere Sprachen wie etwa Python versuchen, aus dem Kontext zu „erraten“, welche Art Information vorliegt. Bezug nehmend auf das Zitat zu Eingang des Kapitels nennt man solche Sprachen *duck typed*. C dagegen ist *strong typed*. Die Meinungen zu *Duck-Typing* gehen teils weit auseinander und werden bisweilen recht emotional diskutiert.

„negativen Null“ ist, wird die Wertigkeit des Vorzeichenbits von der gesamten Zahl abgezogen. Dieses schwierig in Worte zu fassende Konzept wird klarer, wenn wir es anhand eines Beispiels betrachten:



Für die Bits 0 bis 6 ändert sich nichts an der Interpretation, die bereits in Abschnitt 1.1 besprochen wurde. Lediglich das Bit 7 wird nun als Vorzeichenbit interpretiert und erhält somit seine *negative Wertigkeit*. Es ergibt sich ein Wert von $-128 + 32 + 8 + 2 = -86$.

ints sind *vorzeichenbehaftet*. Will man nur positive Zahlen speichern, kann man bei der Deklaration der Variablen den *Modifier* **unsigned** benutzen:

Beispiel: Deklaration einer vorzeichenlosen Ganzzahl

```

1  int main () {
2      unsigned int positiv = 9;
3  }
```

Was geschieht nun, wenn man einer **unsigned**-Variable einen negativen Wert zuweisen will? Betrachten wir das Beispiel:

Beispiel: Zuweisung einer negativen Zahl zu einer unsigned-Variable

```

1  int main () {
2      unsigned char positiv = -1;
3  }
```

Hier wird der Compiler zuerst das Bitmuster der Zahl -1 mit einer Breite von 8 Bit erzeugen, da ein **char** geschrieben werden soll. Dieses Bitmuster lautet 11111111. Da die Variable **positiv** aber ein **unsigned char** ist, wird dieses Bitmuster nun als positive Zahl interpretiert und im Folgenden als 255 gelesen.

Für die Darstellung von Kommazahlen (in der Literatur meist *Fließkommazahlen* genannt) stehen die Datentypen **float** (32 bit), **double** (64 bit) und **long double** (80 bit) zur Verfügung, die im Detail selten verstanden werden müssen. Durch die größere Datenmenge bei **doubles** und **long doubles** fallen Rundungsfehler bei der Arbeit mit diesen Typen weniger ins Gewicht, jedoch sind Berechnungen hier zeitaufwändiger. In den meisten Fällen ist **double** der am besten geeignete Datentyp beim Umgang mit Fließkommazahlen.

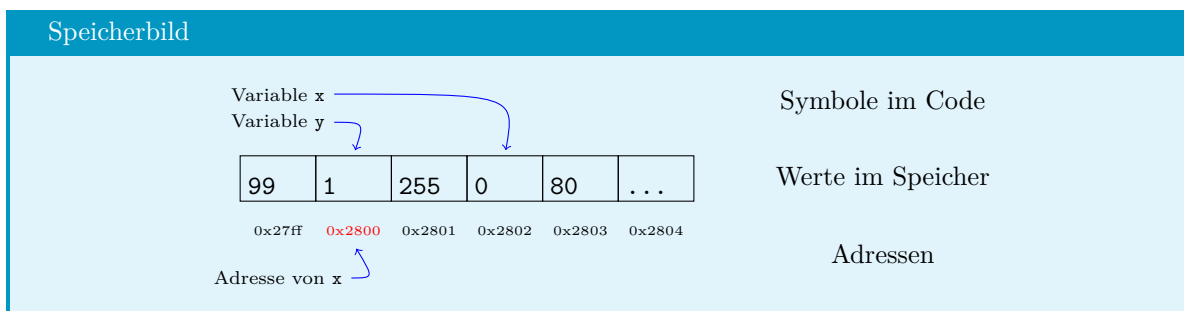
Es gibt keine vorzeichenlosen Fließkommazahlen.

KursteilnehmerInnen, die sich weiter für den Aufbau von Fließkommawerten interessieren, können nach dem Stichwort *Standard IEEE 754* suchen. Insbesondere finden Sie unter der Adresse <https://>

steve.hollasch.net/cgindex/coding/ieeefloat.html eine einsteigerfreundliche Erklärung. Im Kurs *Numerik* der Universität Regensburg wird die Darstellung von Fließkommazahlen ebenfalls behandelt.

3.1.2. Adressierung – Pointer

Man kann sich den Arbeitsspeicher als langes Band von kleinen, nummerierten Speicherzellen vorstellen. Jede Zelle fasst genau ein Byte. Um einen Wert zu lesen oder zu schreiben muss dem Prozessor die Nummer der Zelle mitgeteilt werden, die verändert wird. Diese Nummer wird *Adresse* oder *Pointer* genannt. Wenn wir im Code Variablen benutzen, übersetzt der Compiler diese in Adressen. Für die einfachsten Aufgaben kann der Compiler die Speicherverwaltung komplett übernehmen. Wir werden jedoch schon in Kapitel 4 Situationen kennen lernen, wo wir diese Speicherstruktur kennen müssen.



Adressen sind Zahlen und können damit auch im Speicher als binäre Werte abgebildet werden. Abhängig von der *Prozessorarchitektur* handelt es sich dabei um 32-bit oder 64-bit Ganzzahlen. Meist werden diese nicht als *Dezimalzahlen*, sondern als *Hexadezimalzahlen* angegeben (d. h. mit den „Ziffern“ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Siehe Abschnitt 3.5 für weitere Details hierzu.

Neben der Information, wo im Speicher eine bestimmte Information liegt, muss natürlich auch bekannt sein, wie diese zu interpretieren ist. Daher wird an dieser Stelle zu jedem Datentyp ein zugehöriger *Pointer* eingeführt, also ein Datentyp, der die Adresse einer Information enthält.

Pointer können ebenfalls in Variablen gespeichert werden. Die Deklaration erfolgt wie schon in Abschnitt 2.2.2. Der einzige Unterschied ist, dass ein Pointer-Typ gegenüber seinem *Basistyp* durch ein Sternchen * ausgezeichnet wird.

Beispiel: Deklaration von Pointer-Variablen

```
1 int main () {  
2     int normaleVariable;  
3     int * pointer;  
4 }
```

Um die Adresse anderer Variablen in Erfahrung zu bringen bedienen wir uns des Adress-Operators &:

Beispiel: Deklaration von Pointer-Variablen

```
1 int main () {  
2     int normaleVariable;  
3     int * pointer = &normaleVariable;  
4 }
```

Wir sagen also, *pointer* „zeigt auf“ *normaleVariable*, d. h. in der Variablen *pointer* ist die Information gespeichert, wo *normaleVariable* im Speicher zu finden ist.

Pointer können *dereferenziert* werden; das heißt, anstatt mit der Adresse selbst zu arbeiten, behandeln wir den Wert, auf den der Pointer zeigt. Dies kann sowohl lesend als auch schreibend stattfinden. Wir benutzen den *Dereferenzierungs-Operator* `*`:

Beispiel: Dereferenzierung eines Pointers

```
1  #include <stdio.h>
2
3  int main () {
4      int    normaleVariable = 5;
5      int * pointer = &normaleVariable;
6
7      printf("normaleVariable = %d\n", normaleVariable);
8      printf("*pointer          = %d\n", *pointer);
9      printf("Adresse von normaleVariable: %p\n", pointer);
10
11     *pointer = 8;
12     printf("normaleVariable = %d\n", normaleVariable);
13 }
```

Wir legen zuerst eine normale Variable mit Wert 5 an, sowie einen Pointer, der auf diese Variable zeigt. Wenn über den *dereferenzierten Pointer* `*pointer` in Zeile 8 ein lesender Zugriff stattfindet, passiert genau dasselbe, als würde direkt mit der Variablen `normaleVariable` selbst gearbeitet werden: `printf` liest den Wert 5 aus dem Speicher. Ebenso ändert auch der Zugriff über `*pointer` in Zeile 11 tatsächlich den Wert von `normaleVariable`. Die Ausgabe des Wertes von `pointer` selbst (also ohne Dereferenzierungs-Operator `*`) in Zeile 9 liefert eine große Zahl in Hexadezimal-Darstellung, die mit dem Wert von `normaleVariable` nichts zu tun hat.

Folglich lautet die Ausgabe:

Ausführungsbeispiel: Dereferenzierung eines Pointers

```
normaleVariable = 5
*pointer        = 5
Adresse von normaleVariable: 0x7ffdea635c5c
normaleVariable = 8
```

Große Bedeutung von Pointern in C

In der Sprache C ist der Umgang mit Pointern üblich, aber zugleich auch eine der größten Fehlerquellen. Verinnerlichen Sie daher die soeben gezeigten Konzepte; wir werden hierauf immer wieder aufbauen.



Abbildung 3.1.: Pointer im echten Leben. Quelle: <https://xkcd.com/138/>

Ausflug: C++

Während C++ ebenfalls Pointer zur Verfügung stellt, die sich dort genauso verhalten wie in der Sprache C, gilt das Paradigma, in C++ *keine* Pointer zu verwenden. C++ kennt andere Konzepte, die Pointer ersetzen (bzw. verstecken) und die helfen, häufige Fehler bei der Arbeit mit Pointern zu vermeiden. Im Rahmen dieses Kurses können leider nur Stichworte genannt werden; sehen Sie dies als Motivation, einen C++-Kurs zu besuchen.

Auch wenn C++-Code i. d. R. keine Verwendung von Pointern macht ist die Kenntnis des Konzepts auch dort sehr wichtig, da der gesamte „Unterbau“ der Sprache auf den hier präsentierten Techniken beruht.

3.2. Bitweise Logik und Bit-Shifting

Für Elektronik-Anwendungen ist es oft nötig, einzelne Schalter an- oder auszuschalten. Meist geschieht dies über eine Zustandsvariable, die an einen *Port* (ein Anschluss, über den Daten fließen können) gesendet wird. Diese Zustandsvariable ist dann i. d. R. eine Ganzzahl-Variable, in der jedes Bit für einen eigenen Schalter steht. Wir benötigen also Mittel, um einzelne Bits zu manipulieren.

Dazu stehen uns die Operatoren `~`, `&`, `|`, `^`, `<<` und `>>` zur Verfügung.

Die *bitweise Negation* (NOT, auch *1er Komplement* genannt) `~` kehrt jedes Bit um. Betrachten wir die vorzeichenlose 8-Bit-Zahl 42 (00101010). Das Ergebnis der Negation ist der Wert 213 (Binärzahl 11010101). Als Code lässt sich das wie folgt umsetzen:

Beispiel: Bitweise Negation

```
1 int main () {
2     unsigned char source = 42;
3     unsigned char result = ~source;
4 }
```


Das *bitweise und* (AND) `&` vergleicht die Bits von zwei Werten und setzt das Bit im Ergebnis nur, wenn beide Bits in den Vergleichswerten gesetzt waren (d. h. den Wert 1 hatten).

Beim *bitweisen oder* (OR) `|` werden ebenfalls die Bits von zwei Werten verglichen. Das entsprechende Bit im Ergebnis wird jedoch bereits gesetzt, wenn mindestens eines der Vergleichsbits gesetzt war.

Das *bitweise ausschließliche oder* (XOR) `^` arbeitet genauso wie das OR, setzt das Ergebnis-Bit jedoch nur, wenn *genau* eines der Vergleichsbits gesetzt war.

All dies lässt sich in den folgenden Wahrheits-Tabellen ausdrücken:

| Wahrheitstabellen: Bitweise Operatoren | | | | | |
|--|-----|---|----|----|----|
| 1 | 1 | 0 | 0 | 12 | |
| 1 | 0 | 1 | 0 | 10 | |
| - | AND | - | -- | -- | OR |
| 1 | 0 | 0 | 0 | 8 | 14 |
| 1 | 1 | 0 | 0 | 12 | |
| 1 | 0 | 1 | 0 | 10 | |
| - | XOR | - | -- | -- | |
| 0 | 1 | 1 | 0 | 6 | |

Als Code lässt sich das wie folgt umsetzen:

```

Beispiel: Bitweise Operatoren

1  int main () {
2      unsigned char lhs = 12, rhs = 10,
3          and, or, xor;
4      and = lhs & rhs;    // = 8
5      or  = lhs | rhs;    // = 14
6      xor = lhs ^ rhs;    // = 6
7  }

```

Die Negation benötigt nur ein *Argument* (Wert, mit dem gearbeitet wird), und wird daher *unärer Operator* genannt. Für AND, OR und XOR sind zwei Werte nötig, weswegen diese als *binäre Operatoren* bezeichnet werden. Die Argumente können auch komplexe Ausdrücke sein, also Ergebnisse von Berechnungen:

```

Beispiel: Bitweise Operatoren

1  int main () {
2      unsigned char result = ~(2 + 7 * (18 & 5));
3  }

```

Bitmuster können als ganzes nach links oder rechts verschoben werden. Hierzu dienen die Operatoren `<<` und `>>`.

```

Beispiel: Bitweise Operatoren

1  int main () {
2      unsigned char toLeft  = 170 << 1;
3      unsigned char toRight = 85 >> 1;
4  }

```

Dies löst folgende Veränderung aus:

Bitshift nach links bzw. nach rechts um jeweils eine Stelle

| | | | |
|-----------------|-----|-----------------|-----|
| 1 0 1 0 1 0 1 0 | 170 | 0 1 0 1 0 1 0 1 | 85 |
| ----- << 1 ---- | --- | ----- >> 1 ---- | --- |
| 0 1 0 1 0 1 0 0 | 84 | 0 0 1 0 1 0 1 0 | 42 |

Bei einem Bitshift nach links werden die Bits am rechten Ende des Bitmusters mit nullen aufgefüllt. Stellen, die über das linke Ende hinaus verschoben werden, gehen verloren. Alles eben gesagte gilt analog für den Bitshift nach rechts.

Multiplikation mit Zweierpotenzen

Ein Bitshift um n Stellen nach links entspricht der Multiplikation mit der Zahl 2^n . Ein Bitshift um n Stellen nach rechts entspricht der Division durch die Zahl 2^n (Bei Ergebnissen mit Nachkommastelle wird abgerundet). Bitshifts werden schneller durchgeführt als Multiplikationen mit $*$.

Dies gilt nur für Ganzzahlen, nicht aber für Fließkommazahlen.

3.3. Typecasting

Es ist gelegentlich notwendig, einen Wert von einem Datentyp in einen anderen umzuwandeln. Bei einer solchen Umwandlung kann sich das Bitmuster ändern, beispielsweise wenn die Ganzzahl 3 in die Fließkommazahl 3.0 umgewandelt wird.

Eine solche Umwandlung wird als *Typecasting* bezeichnet. Die Syntax hierzu ist simpel:

Syntax: Typecasting

(Ziel-Datentyp) Ausdruck

Ziel-Datentyp ist ein beliebiger Datentyp, wie beispielsweise in Anhang B.6 beschrieben. Wie üblich kann *Ausdruck* ein einfacher Wert, eine Variable oder ein komplexer Ausdruck sein.

Ein Grund, den Datentyp zu ändern wurde zum Ende von Abschnitt 2.2.4 bereits angesprochen: Operationen wie die Division verhalten sich unterschiedlich, abhängig davon, ob eine Fließkommazahl oder eine Ganzzahl an der Operation beteiligt ist. Solche *Datentyp-spezifischen* Szenarios sind in C sehr häufig.

Beispiel: Typecasting int zu double

```

1  int main () {
2      int x = 7, y = 5;
3      double z = (double) x / y;
4  }
```

Reihenfolge der Operationen

Der Compiler arbeitet „von links nach rechts“, beachtet aber, dass manche Operationen „Vorrang“ vor anderen haben (z. B. Punkt-vor-Strich). Im obigen Beispiel wird also zuerst der Typecast der Variablen `x` durchgeführt, und dann der zum **double** konvertierte Wert von `x` durch `y` geteilt. Damit erhält `z` den Wert 1.2.

Im folgenden Beispiel dagegen erhält `z` den Wert 1.0:

Beispiel: Fehlerhaftes Typecasting int zu double

```
1  int main () {  
2      int x = 7, y = 5;  
3      double z = (double) (x / y);  
4  }
```

Durch die Klammern interpretiert der Compiler Zeile 3 als: „Caste das Ergebnis von `x / y` zu **double**“. Da das Ergebnis von `x / y` aus der Division zweier **int**-Werte hervorgeht, wird hier gerundet und `z` erhält den (i. d. R. falschen) Wert 1.0.

3.4. Hierarchie der Operatoren

Ausdrücke werden vom Compiler „von links nach rechts“ ausgewertet, wobei die Vorrangigkeit mancher Operatoren beachtet wird. Die Vorrangigkeit kann Tabelle B.7 im Anhang entnommen werden. Wo immer Unsicherheiten bestehen, kann durch Setzen von (runden Klammern) Sicherheit geschaffen werden.

3.5. Zahlenformate – Hexadezimalsystem

Gleich zu Beginn dieses Kurses in Abschnitt 1.1 haben wir das *Binärsystem* (oder auch *Dualsystem*) kennen gelernt, das neben dem uns vertrauten *Dezimalsystem* steht³. Nach der dort gezeigten Idee lassen sich Zahlen in beliebigen *Zahlensystemen* bzw. mit beliebiger *Basis* (also Anzahl von verschiedenen Ziffern im Zahlensystem) darstellen. Von besonderer Bedeutung ist das *Hexadezimalsystem* mit 16 Ziffern:

0 1 2 3 4 5 6 7 8 9 A B C D E F

Um eine Zahl als *hexadezimal* zu kennzeichnen wird häufig der Präfix `0x` vorangestellt. Der Ausdruck⁴ `0x10` entspricht also der *Dezimalzahl* 16. Groß- und Kleinschreibung wird hier nicht unterschieden, es gilt `0xab` = `0XAB`.

Die weiteren Informationen in diesem Abschnitt sind für besonders interessierte KursteilnehmerInnen gedacht und müssen für den Kurserfolg nicht im Detail verstanden werden. Sie werden diese Idee aber in der IT-Welt immer wieder antreffen und Nutzen aus dem Verständnis ziehen.

Die Darstellung von Zahlen bzw. allgemeiner von Bitfolgen im Hexadezimalsystem ist mitunter von Vorteil⁵. Wir haben bereits gehört, dass ein Computer nur Gruppen von Bits behandeln kann. Ein

³Übrigens: Das Dezimalsystem wäre völlig unbrauchbar, wenn der Mensch nicht zufällig zehn Finger hätte. (Kalenderspruch)

⁴Andere verbreitete Schreibweisen sind `10h`, `1016` oder `10hex`

⁵Historische Bedeutung hat in diesem Kontext auch noch das *Oktalsystem*, das die Ziffern 0..7 verwendet. Heute lebt es fast nur noch in dem Witz fort: *Why do programmers confuse Halloween and Christmas? Because Oct 31 = Dec 25...*

Byte, also eine Gruppe von 8 Bit erlaubt, wie in Abschnitt 3.1.1 erwähnt, 256 verschiedene Werte. Im Hexadezimalsystem wird daraus der „runde“ Wert 0x100. Ähnlich ist die Zahl der mit 16 Bit darstellbaren Zahlen gleich 0x10000 und für 32 bit erhalten wir 0x100000000 verschiedene Möglichkeiten. Dies ist kein Zufall – Die Zahl 16 (also die Basis des Hexadezimalsystems) ist eine Potenz der Zahl 2 (also der Basis des Binärsystems). Es besteht eine „direkte Verwandtschaft“ zwischen Binärzahlen und Hexadezimalzahlen.

Eine Zahl, die mit n Bit dargestellt wird, kann also mit $\frac{n}{4}$ Hexadezimal-Ziffern geschrieben werden. Dies ist offensichtlich praktischer als lange Ketten von 1en und 0en zu schreiben.

Farben als Hexadezimal-Werte

In der Computertechnik werden Farben häufig mit *Hex-Codes* beschrieben. Dies sind sechsstellige Kombinationen aus Ziffern und den Buchstaben A bis F. Tatsächlich sind diese Codes drei aneinander gereihte Hexadezimalzahlen. Diese beschreiben jeweils den Rot-, Grün- und Blau-Anteil einer Farbe. Der Dezimalwert 255 (bzw. die Hexadezimalzahl 0xFF) beschreibt dabei „volle Intensität“, 0 (oder 0x00 dagegen sagt aus, von der Teilfarbe kommt kein Beitrag. Damit kann die Farbe Orange dann ausgedrückt werden als 0xFF7F00: Volle Intensität Rot (FF), halbe Intensität Grün (7F) und Blau-Anteil 0 (00).

Siehe auch https://de.wikipedia.org/wiki/Additive_Farbmischung

Zweier-Potenzen in der Computerwelt

Alles in der Computertechnik ist um Potenzen der Zahl 2 aufgebaut. Diese Zahlen kommen derartig häufig vor, dass sie von TechnikerInnen bereits als „runde Zahlen“ wahrgenommen werden. So feierte beispielsweise Randall Munroe, Author des Webcomics xkcd seinen eintausendsten Strip mit dieser Veröffentlichung:

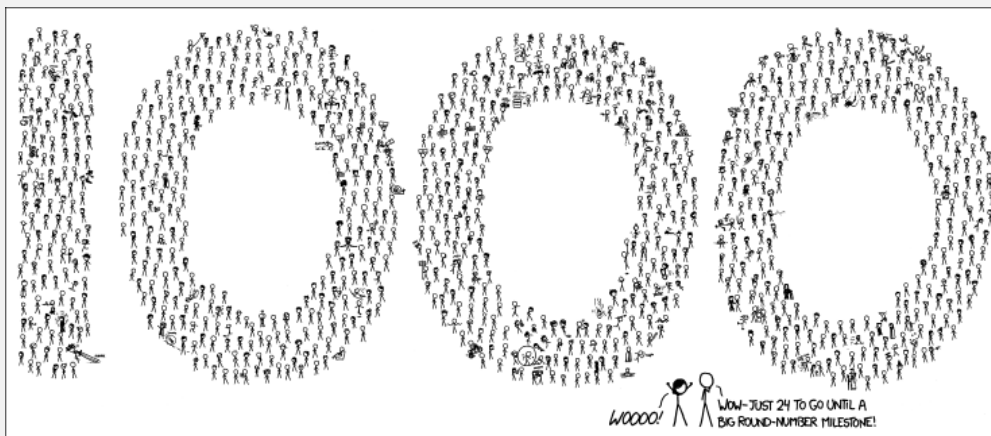


Abbildung 3.2.: Dezimale und binäre runde Zahlen. Quelle: <https://www.xkcd.com/1000/>

4. Dateneingabe

On two occasions I have been asked, „Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?“ ... I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.

Charles Babbage, Passages from the Life of a Philosopher

In den letzten Kapiteln haben wir gesehen, wie wir in der Sprache C einfache Berechnungen realisieren können. Die Daten, mit denen gerechnet wurde, standen dabei jedoch schon vor Start des Programmes fest. Im folgenden sollen einfache Mittel vorgestellt werden, die Usereingaben und damit Interaktivität ermöglichen.

4.1. Dateneingabe mit `scanf`

Die Funktion `scanf` dient dazu, Tastatureingaben von der Konsole einzulesen, in verschiedene Datenformate umzurechnen und dann als solche im Speicher abzulegen. Die Funktion ist deklariert im Header `stdio.h` (wie auch `printf`). Betrachten wir das folgende Beispiel:

Beispiel: Eine Ganzzahl mit `scanf` einlesen

```
1  #include <stdio.h>
2
3  int main () {
4      int Ganzzahl = -1;
5
6      printf("Bitte geben Sie eine Ganzzahl ein:\n");
7      scanf("%d", &Ganzzahl);
8
9      printf("Ihre Eingabe war: %d\n", Ganzzahl);
10 }
```

Wie gewohnt haben wir den Header `stdio.h` in Zeile 1 geladen und können nun die Funktionen `printf` und `scanf` verwenden. In Zeile 4 deklarieren wir eine Variable `Ganzzahl` und initialisieren sie mit dem Startwert `-1`. Nach der Aufforderung an den Benutzer unseres Programms (Zeile 6) lesen wir jetzt mit `scanf` einen Wert in den Speicher:

Ähnlich wie `printf` nimmt `scanf` als erstes Argument einen *Formatstring* an. Dieser beschreibt, welche Art von Werten erwartet wird, und benutzt dieselben Formatierungszeichen wie `printf` (siehe Tabelle B.4 im Anhang für Details). Das folgende Argument ist die *Adresse, an die der eingelesene Wert geschrieben werden soll*. Wir benutzen den Adress-Operator `&`, um dem Compiler mitzuteilen, dass der eingelesene Wert an der Adresse der Variable `Ganzzahl` abgelegt werden soll. Hier überschreiben wir also den Wert von `Ganzzahl` mit der Eingabe des Users, interpretiert als `int` (vorgegeben durch das Formatierungszeichen `%d`).

In der Ausführung kann dies so aussehen:

Ausführungsbeispiel: Eine Ganzzahl mit `scanf` einlesen

```
Bitte geben Sie eine Ganzzahl ein:  
42  
Ihre Eingabe war: 42
```

Unterschied von Zahlenwerten und Text mit Ziffern

Bitte beachten Sie: Die User-Eingabe war ein Text, bestehend aus zwei *Zeichen* (eben der 4 und der 2). Diese beiden Zeichen werden von `scanf` erst zu dem Zahlenwert 42 zusammengesetzt und schließlich als **int**-Wert im Speicher abgelegt. Das *Bitmuster* des „Textes“ 42 ist ein anderes als das der *Zahl* 42. Wir werden bald Fälle sehen, in denen dieser Unterschied von Bedeutung ist. Um diese Denkweise bereits jetzt einzuüben, mache ich Sie hier auf den Gedanken aufmerksam.

Beim Einlesen versucht `scanf`, die User-Eingaben im Kontext des angegebenen Datentyps zu interpretieren. Passen die Eingaben nicht zum Typ, so werden sie i. d. R. ignoriert, können aber auch unvorhergesehene Ergebnisse liefern. Hierzu zwei Laufzeit-Beispiele zum obigen Code:

Ausführungsbeispiel: Eine Ganzzahl mit `scanf` einlesen

```
Bitte geben Sie eine Ganzzahl ein:  
12.3  
Ihre Eingabe war: 12
```

Während der Code einen Ganzzahl-Wert erwartet (`%d`), hat der User eine Fließkommazahl eingegeben. `scanf` ignoriert den Nachkomma-Anteil und speichert nur den Wert 12.

Ausführungsbeispiel: Eine Ganzzahl mit `scanf` einlesen

```
Bitte geben Sie eine Ganzzahl ein:  
a  
Ihre Eingabe war: -1
```

Der Buchstabe `a` kann nicht als Ganzzahl interpretiert werden. Der Startwert `-1` wird nicht überschrieben.

Fehlerhafte Usereingaben erwarten

Als ProgrammiererIn sollten Sie immer davon ausgehen, dass die Anwender Ihrer Programme Fehler machen werden. Diese Fehler sollen von Ihrem Programm *abgefangen* werden können. Wir werden in Kapitel 5 lernen, wie wir Teile unseres Codes nur unter bestimmten Bedingungen ausführen lassen. Um solche Bedingungen zu formulieren müssen jedoch geeignete (definierte) Startbedingungen vorliegen. Im Beispiel oben könnte geprüft werden, ob der Wert von `Ganzzahl` nach dem Eingabeversuch immer noch `-1` ist und eine entsprechende *Fehlerbehandlung* für diesen Fall nachgeschaltet werden. Natürlich ist es möglich, dass ein Anwender genau den Wert `-1` eingeben will. Welche Startwerte sinnvoll sind und wie die Fehlerbehandlung aussehen kann, hängt vom jeweiligen Kontext ab. Behalten Sie in jedem Fall den Gedanken, dass fehlerhafte Usereingaben häufig sind und von Ihnen „vorhergesehen“ werden sollten. Aus diesem Grund ist es sinnvoll hier einen Startwert für die Variable `Ganzzahl` festzulegen, auch wenn dieser quasi sofort überschrieben wird.

Betrachten wir weiter das folgende Beispiel:

Beispiel: Fehler bei `scanf`: Falsche Typisierung

```
1  #include <stdio.h>
2
3  int main () {
4      int Ganzzahl = -1;
5
6      printf("Bitte geben Sie eine Ganzzahl ein:\n");
7      scanf("%f", &Ganzzahl);
8
9      printf("Ihre Eingabe war: %d\n", Ganzzahl);
10 }
```

Gegenüber dem Beispiel zu Eingang des Kapitels hat sich lediglich Zeile 7 geändert: Statt mit `%d` einen `int` zu lesen, fordern wir nun mit `%f` dazu auf, eine `float`-Variable zu lesen. Diese wird wieder an den Speicherort der Variablen `Ganzzahl` geschrieben, die jedoch vom Typ `int` ist!

Der Compiler kann diesen (fehlerhaften) Code umsetzen, gibt aber eine entsprechende Warnung aus:

Compiler-Warnung: Fehlerhafte Typisierung mit `scanf`

```
myProgram.c: In function 'main':
myProgram.c:7:12: warning: format '%f' expects argument of type 'float *', but
argument 2 has type 'int *' [-Wformat=]
    scanf("%f", &Ganzzahl);
           ~^  ~~~~~
           %d
```

Ähnlich wie bei `printf` markiert der Compiler die relevante Stelle und schlägt vor, mit `%d` einzulesen. Es handelt sich aber nur um eine Warnung; der Compiler kann „funktionierenden“ Maschinencode erzeugen. Ebenso wie auch bereits zu Ende von Abschnitt 2.3 erklärt, ist das Ergebnis dieser Ausführung aber „unsinnig“:

Ausführungsbeispiel: Fehlerhafte Typisierung mit `scanf`

```
Bitte geben Sie eine Ganzzahl ein:
1
Ihre Eingabe war: 1065353216
```

Noch schwerwiegender wird der Fehler, wenn wir statt dem Formatierungscode `%f` für `float` mit `%lf` einen `double`-Wert einlesen. Variablen des Typs `int` und `float` sind 32 Bit breit¹. Ein `double`-Wert dagegen ist 64 Bit breit. Beim Einlesen wird also „über die Grenzen des reservierten Speicherbereichs hinaus geschrieben“. Der Effekt hiervon ist absolut *undefiniertes Verhalten*. Im „besten Fall“ wird der „zu viel beschriebene“ Speicherplatz nicht benötigt und das Programm läuft ohne größere Nebeneffekte weiter. Es ist aber auch möglich, dass in andere reservierte Bereiche hinein geschrieben wird (*bleeding*) und so andere Variablen geändert werden:

¹zumindest auf den heute üblichen Rechnern

Beispiel: Fehler bei `scanf`: bleeding

```
1  #include <stdio.h>
2
3  int main () {
4      int x = 10;
5      int y = 10; // Adresse von y entspricht Adresse von x + 4 !
6
7      printf("Bitte geben Sie eine Zahl ein:\n");
8      scanf("%lf", &x);
9
10     printf("(x, y) = (%d, %d)\n", x, y);
11 }
```

Die beiden Variablen `x` und `y` werden im Speicher direkt aufeinander folgend angelegt. Schreiben wir über die Grenzen von `x` hinaus, so ändern wir auch `y`:

Ausführungsbeispiel: Fehler bei `scanf`: bleeding

```
Bitte geben Sie eine Zahl ein:
1
(x, y) = (0, 1072693248)
```

Im schlimmsten Fall resultiert unvorhersagbares Laufzeit-Verhalten oder Absturz des Programms wegen Speicherzugriffsfehler (*Segmentation fault* bzw. kurz *Segfault*).

Typisierung

Achten Sie extrem genau auf richtige Typisierung wann immer Sie mit Adressen (Operator `&`) arbeiten. Fehler dieser Art sind in großen Projekten extrem schwierig zu finden und führen zu instabilem Code. Beachten Sie insbesondere alle Warnungen, die der Compiler ausgibt.

Wie bei `printf` darf der Formatstring bei `scanf` auch mehrere Platzhalter enthalten, die dann durch Leerstellen voneinander getrennt werden. Der User hat dann die Möglichkeit, alle Werte „in einer Zeile“ einzugeben, und diese durch Leerzeichen zu trennen, oder jeden Wert in eine eigene Zeile zu setzen.

Beispiel: Eingabe mehrerer Werte mit `scanf`

```
1  #include <stdio.h>
2
3  int main () {
4      int x = 0;
5      double y = 0;
6
7      printf("Bitte geben Sie eine Ganzzahl und eine Fließkommazahl ein:\n");
8      scanf("%d %lf", &x, &y);
9
10     printf("(x, y) = (%d, %lf)\n", x, y);
11 }
```


Ausführungsbeispiel: Eingabe mehrerer Werte mit `scanf`

```
Bitte geben Sie eine Ganzzahl und eine Fließkommazahl ein:  
1 1.5  
(x, y) = (1, 1.500000)
```

Jeder Wert wird einzeln interpretiert; natürlich muss die Reihenfolge der Usereingaben der Reihenfolge der Platzhalter in der `scanf`-Anweisung entsprechen.

User-Freundliches Verhalten

Die Erfahrung zeigt, dass User bei der Eingabe mehrerer Werte in einem Block häufig Fehler machen. Ich empfehle, jeweils nur einen einzelnen Wert einlesen zu lassen, und vor jedem `scanf`-Befehl einen Hinweis mit `printf` auszugeben, welcher Wert erwartet wird.

Adressen bei `scanf`

Wichtig ist, dass `scanf` eine *Adresse* erwartet. Ein häufiger Fehler ist es, den Adress-Operator `&` zu vergessen wie im folgenden Beispiel:

Beispiel: Eine Ganzzahl mit `scanf` einlesen

```
1  #include <stdio.h>  
2  
3  int main () {  
4      int Ganzzahl = 0;  
5  
6      printf("Bitte geben Sie eine Ganzzahl ein:\n");  
7      scanf("%d", Ganzzahl);  
8  
9      printf("Ihre Eingabe war: %d\n", Ganzzahl);  
10 }
```

Hier wird versucht, an die Adresse zu schreiben, die gerade in der Variablen `Ganzzahl` gespeichert ist – also an die Adresse 0. Dies ist ein verbotener Zugriff und endet daher in einem Programmabsturz.

Der Compiler gibt eine entsprechende Warnung aus:

Compilerwarnung: Ungültige Adresse übergeben

```
myProgram.c: In function 'main':  
myProgram.c:7:12: warning: format '%d' expects argument of type 'int *',  
but argument 2 has type 'int' [-Wformat=]  
    scanf("%d", Ganzzahl);  
           ~^
```

und die Programmausführung stoppt mit einem *Segfault* bzw. Speicherzugriffsfehler:

Ausführungsbeispiel: Ungültige Adresse übergeben

```
Bitte geben Sie eine Ganzzahl ein:  
1  
Speicherzugriffsfehler (Speicherabzug geschrieben)
```

4.2. Ausblick: Eingabe in C++

Ausflug: C++

Wie schon bei der Textausgabe kann alles oben gesagte auch in C++ angewandt werden. Es bietet sich jedoch an, auch hier wieder *Streams* zu verwenden. Das folgende Beispiel zeigt, wie mit den Mitteln der `iostream`-Bibliothek Werte eingelesen werden können.

Beispiel: Eingabe mehrerer Werte mit `scanf`

```
1  #include <iostream>
2
3  int main () {
4      int    x = 0,
5            y = 0;
6      double z = 0;
7
8      std::cout << "Bitte geben Sie eine Ganzzahl ein:" << std::endl;
9      std::cin  >> x;
10
11     std::cout << "Bitte geben Sie eine Ganzzahl ";
12     std::cout << "und eine Fließkommazahl ein:" << std::endl;
13     std::cin  >> y >> z;
14
15     std::cout << "Ihre erste Eingabe war: " << x << std::endl;
16     std::cout << "Ihre zweite Eingabe war: ";
17     std::cout << y << ", " << z << std::endl;
18 }
```

Wir verwenden also den *Eingabestream* `std::cin`. Aus diesem können mit dem Operator `>>` Werte „angefordert“ werden, die nach denselben Regeln interpretiert werden, wie dies schon bei `scanf` besprochen wurde. Jedoch entfällt die Notwendigkeit eines Format-Strings; der Datentyp wird *automatisch* aus dem Typ der Variablen ermittelt, die hinter `>>` steht. Achtung: Hier werden die Variablen selbst, nicht ihre Adressen aufgeführt; der Operator `&` entfällt! Mehr zum Thema unter <https://en.cppreference.com/w/cpp/io/cin>.

5. Bedingungen

In any moment of decision, the best thing you can do is the right thing, the next best thing is the wrong thing, and the worst thing you can do is nothing.

Theodore Roosevelt

Bis hierhin haben wir Programme mit *linearem* Verlauf geschrieben: unsere Befehle werden von oben nach unten ohne Auslassungen oder Sprünge bearbeitet. Hier nun wollen wir erste Strukturen einführen, die den *Programmfluss* ändern: Wir wollen bestimmte Code-Teile nur dann ausführen, wenn bestimmte *Bedingungen* erfüllt sind.

5.1. Wahrheitswerte

Mathematischen Ausdrücken können *Wahrheitswerte* zugeordnet werden. Diese Wahrheitswerte sagen aus, ob es sich um eine „wahre“ oder „falsche“ Aussage handelt. Ein Beispiel für einen solchen Ausdruck ist:

`1 + 5 * 8 + 1 == 42`

Der Wahrheitswert dieses Ausdrucks ist offensichtlich *wahr* (bzw. *true*).

Beachten Sie bitte, dass für den *Vergleich* zweier Zahlen das *doppelte Gleichheitszeichen* `==` verwendet wird. Das einfache Gleichheitszeichen `=` dient ausschließlich der *Wertzuweisung* an Variablen.

Tabelle 5.1 listet Vergleichsoperatoren auf.

| Vergleich | Zeichen | Vergleich | Zeichen |
|---------------------|--------------------|--------------------|--------------------|
| Gleichheit | <code>==</code> | Ungleichheit | <code>!=</code> |
| Kleiner als | <code><</code> | Größer als | <code>></code> |
| Kleiner oder gleich | <code><=</code> | Größer oder gleich | <code>>=</code> |

Tabelle 5.1.: Vergleichsoperatoren in C

Da es nur zwei Wahrheitswerte gibt (*wahr* und *falsch*, bzw. *true* und *false*) wird im Prinzip nur ein einzelnes Bit benötigt, um einen solchen Wahrheitswert zu speichern. Ein Prozessor kann jedoch nur Gruppen zu mindestens 8 Bit behandeln, nicht aber einzelne Bits. Für uns sind Wahrheitswerte daher Ganzzahl-Werte, wie etwa `ints`. In der Tat lassen sich solche Wahrheitswerte speichern und ausgeben:

Beispiel: Speichern und Ausgabe von Wahrheitswerten

```
1  #include <stdio.h>
2
3  int main () {
4      int    true  = 1 + 1 == 2,
5             false = 1 + 1 != 2;
6
7      printf("Wert von 'true' : %d\n", true );
8      printf("Wert von 'false': %d\n", false);
9  }
```

Die Ausgabe hierzu lautet:

Ausführungsbeispiel: Speichern und Ausgabe von Wahrheitswerten

```
Wert von 'true' : 1
Wert von 'false': 0
```

Da eine **int**-Variable mehr Werte als 0 und 1 halten kann, interpretiert man alle von null verschiedenen Werte als *wahr*.

Selbstverständlich können Ausdrücke, die zu Wahrheitswerten ausgewertet werden, auch Variablen enthalten. Es kann sogar mit ihnen gerechnet werden (auch wenn dies nur selten sinnvoll ist).

Beispiel: Rechnen mit Wahrheitswerten

```
1  #include <stdio.h>
2
3  int main () {
4      int x      = 17,
5          truth  = (x > 5) + (x < 5) + (x == 17);
6
7      printf("%d wahre Ausdrücke.\n", truth);
8  }
```

Ausführungsbeispiel: Rechnen mit Wahrheitswerten

```
2 wahre Ausdrücke.
```

Nacheinander werden die drei Vergleiche $x > 5$, $x < 5$ und $x == 17$ ausgewertet (jeweils zu 1, 0, und 1) und dann aufsummiert. Diese Summe wird in der normalen **int**-Variable **truth** gespeichert.

Symbole **true** und **false**

Die Symbole **true** und **false** sind zwar nach keinem bisherigen C-Standard definiert; in vielen Bibliotheken sind diese Symbole aber mit den Werten 1 bzw. 0 definiert, und sollten daher auch ausschließlich im Sinne von Wahrheitswerten verwendet werden. Vermeiden Sie nach Möglichkeit die Verwendung der Symbole als Variablen ganz.

In C++ sind sowohl **true** als auch **false** fester Bestandteil der Sprache und können nicht als Variablennamen vergeben werden.

5.2. Bedingte Ausführung von Code: `if`

Mit dem Konstrukt `if` lassen sich *Wenn-Dann-Blöcke* erstellen: WENN *Bedingung erfüllt*, DANN *führe Anweisungen aus*. Bedingungen werden als mathematische Ausdrücke formuliert, die als erfüllt gelten, wenn ihr Wahrheitswert *true* ist. *Anweisungen* kann ein einzelner Befehl oder eine ganze Reihe von Befehlen sein. In der einfachsten Form sieht ein `if`-Block so aus:

Syntax: Einfacher `if`-Block

```
if (Bedingung) {  
    ... Anweisungen ...  
}
```

Im Kontext eines vollständigen Programms kann dies so aussehen:

Beispiel: Einfacher `if`-Block

```
1  #include <stdio.h>  
2  
3  int main () {  
4      int foo = 0;  
5  
6      printf("Bitte geben Sie eine Ganzzahl ein:\n");  
7      scanf("%d", &foo);  
8  
9      if (foo % 2 == 0) {  
10         printf("%d ist eine gerade Zahl.\n", foo);  
11     }  
12  
13     printf("Sie haben %d eingegeben.\n", foo);  
14 }
```

Die Anweisung in Zeile 10 wird nur ausgeführt, wenn der Rest der Division von `foo` durch 2 gleich null ist, also wenn der Wert der Variablen gerade ist. Unabhängig vom Ausgang dieser Entscheidung wird Zeile 13 in jedem Fall ausgeführt.

Wenn auf die Bedingung nur eine einzelne Anweisung folgt (wie im obigen Beispiel), so können die {geschweiften Klammern} auch entfallen. Der folgende Code führt also zum selben Ergebnis:

Beispiel: Einzeiliger `if`-Block

```
1  #include <stdio.h>  
2  
3  int main () {  
4      int foo = 0;  
5      printf("Bitte geben Sie eine Ganzzahl ein:\n");  
6      scanf("%d", &foo);  
7  
8      if (foo % 2 == 0)  
9          printf("%d ist eine gerade Zahl.\n", foo);  
10  
11     printf("Sie haben %d eingegeben.\n", foo);  
12 }
```

Fehlerquelle fehlende Klammern

Es mag praktisch erscheinen, die geschweiften Klammern bei einzeiligen `if`-Blocks nicht zu setzen. Nicht selten ergänzt man sein Programm im Laufe eines Projekts aber noch. Hängt man neue Befehle an einen `if`-Block an, so muss man diese Klammern natürlich nachsetzen. Dies wird oft vergessen. Betrachten Sie folgenden Code:

Beispiel: Fehlerhafter `if`-Block durch fehlende Klammern

```
1  #include <stdio.h>
2
3  int main () {
4      int foo = 0;
5
6      printf("Bitte geben Sie eine Ganzzahl ein:\n");
7      scanf("%d", &foo);
8
9      if (foo % 2 == 0)
10         printf("%d ist eine gerade Zahl.\n", foo);
11         printf("%d ist durch 2 teilbar.\n", foo);
12
13     printf("Sie haben %d eingegeben.\n", foo);
14 }
```

Die Ausgabe in Zeile 10 findet nur statt, wenn `foo` einen geraden Wert hält; Zeile 11 dagegen wird fälschlicherweise *immer* ausgeführt. Die Code-Einrückung – eigentlich ein Zeichen guten Stils – kaschiert diesen Fehler noch mehr. Ich rate daher dazu, *immer* Klammern zu setzen. Richtig müsste der `if`-Block also lauten:

Beispiel: Korrekter mehrzeiliger `if`-Block

```
9      if (foo % 2 == 0) {
10         printf("%d ist eine gerade Zahl.\n", foo);
11         printf("%d ist durch 2 teilbar.\n", foo);
12     }
```

Mit dem Schlüsselwort `else` kann der `if`-Block noch um Anweisungen ergänzt werden, die nur ausgeführt werden sollen, wenn die Bedingung gerade *nicht* erfüllt war:

Beispiel: `if-else`-Block

```
1  #include <stdio.h>
2
3  int main () {
4      int foo = 0;
5      printf("Bitte geben Sie eine Ganzzahl ein:\n");
6      scanf("%d", &foo);
7  }
```

```

8   if (foo % 2 == 0) {
9       printf("%d ist eine gerade Zahl.\n", foo);
10  } else {
11      printf("%d ist eine ungerade Zahl.\n", foo);
12  }
13 }

```

Stil: Einrückungen und Position der Klammern

Gute Programmierer sind sich einig, dass Einrückungen fester Bestandteil von gutem Code sind. Ohne diese verliert man in auch nur wenig komplexen Projekten schnell den Überblick und macht unnötige Fehler. Hier aber endet auch schon die Einigkeit; es gibt keine allgemein anerkannte „richtige Art, Code einzurücken“. Tabulatoren oder Leerzeichen, zwei oder vier Zeichen pro Einrückung, oder auch die Position der Klammern – verschiedenste Varianten werden gelebt und kontrovers und teilweise sehr emotional diskutiert, wie Abbildung 5.1 zeigt.

Sie sind also frei, Ihren Stil selbst zu definieren, sollten dabei aber zwei Regeln einhalten:

- Halten Sie Ihren Code einheitlich, d. h. behalten Sie den Einrückungs-Stil über ein gesamtes Projekt konsistent.
- Ändern Sie niemals den Einrückungs-Stil ihrer KollegInnen.

`if`-Blöcke dürfen nahezu beliebig tief verschachtelt werden¹. Das folgende Beispiel prüft also zuerst, ob eine eingegebene Zahl positiv war, und nur für positive Zahlen die *Parität* (die Eigenschaft, gerade oder ungerade zu sein).

Beispiel: Verschachtelter `if-else`-Block

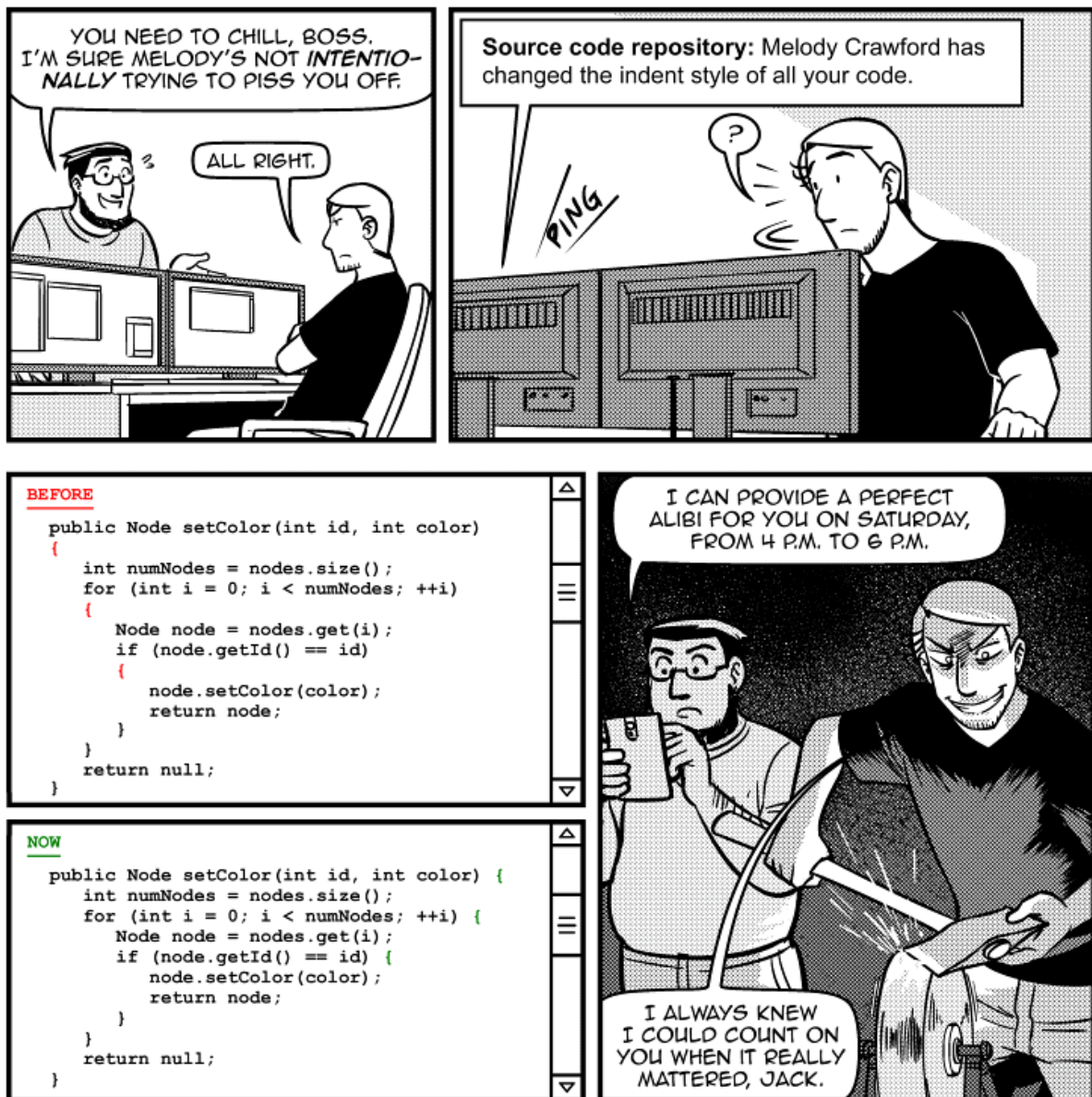
```

1  #include <stdio.h>
2
3  int main () {
4      int foo = 0;
5
6      printf("Bitte geben Sie eine Ganzzahl ein:\n");
7      scanf("%d", &foo);
8
9      if (foo > 0) {
10         if (foo % 2 == 0) {
11             printf("%d ist eine gerade Zahl.\n" , foo);
12         } else {
13             printf("%d ist eine ungerade Zahl.\n", foo);
14         }
15     } else {
16         printf("%d ist eine ungültige Zahl.\n" , foo);
17     }
18 }

```

Spätestens an diesem Beispiel erkennen Sie, weshalb auf Einrückung so viel Wert gelegt wird. Sehen Sie sich zum Vergleich auch das folgende – syntaktisch korrekte – Beispiel ohne Einrückungen an:

¹Der Standard aus dem Jahre 1999 verpflichtet die Programmierer von C-Compilern zwar „nur“ dazu, mindestens 127 Ebenen (256 für C++) zu unterstützen; im Falle des gcc bricht der Kompiliervorgang aber erst nach 6199 Ebenen mit Fehlermeldung ab.



Sandra and Woo by Oliver Knörzer (writer) and Powree (artist) – www.sandraandwoo.com

Abbildung 5.1.: Ein realistisches Szenario

Schlechter Stil: Code mit mehreren Hierarchie-Ebenen ohne Einrückungen

```
1  #include <stdio.h>
2  int main () {
3  int foo = 0;
4  printf("Bitte geben Sie eine Ganzzahl ein:\n");
5  scanf("%d", &foo);
6  if (foo > 0) {
7  if (foo % 2 == 0) {
8  printf("%d ist eine gerade Zahl.\n", foo);
9  } else {
10 printf("%d ist eine ungerade Zahl.\n", foo);
11 }} else {
12 printf("%d ist eine ungültige Zahl.\n", foo);
13 }}
```

Verschachtelte **if-else**-Blocks lassen sich formell auch auf einer Hierarchie-Ebene darstellen. Der folgende Code erzeugt dasselbe Verhalten wie die beiden obigen Beispiele, wirkt aber in manchen Augen klarer²:

Beispiel: if-Block mit mehreren Fällen

```
1  #include <stdio.h>
2
3  int main () {
4      int foo = 0;
5
6      printf("Bitte geben Sie eine Ganzzahl ein:\n");
7      scanf("%d", &foo);
8
9      if (foo <= 0) {
10         printf("%d ist eine ungültige Zahl.\n", foo);
11     } else if (foo % 2 == 0) {
12         printf("%d ist eine gerade Zahl.\n", foo);
13     } else {
14         printf("%d ist eine ungerade Zahl.\n", foo);
15     }
16 }
```

Achten Sie bei dieser Form aber darauf, dass nur der Code zur *ersten erfüllten Bedingung* ausgeführt wird, selbst wenn nachfolgende Blocks ebenfalls den Wahrheitswert *true* haben:

²Technisch handelt es sich hierbei um eine Anwendung der Regel, dass bei einzeiligen **if**-Blocks die Klammern entfallen können.

Beispiel: if-Block mit unerreichbarem Code

```
1  #include <stdio.h>
2
3  int main () {
4      int foo = 0;
5
6      printf("Bitte geben Sie eine Ganzzahl ein:\n");
7      scanf("%d", &foo);
8
9      if      (foo > 5) {
10         printf("%d ist größer als fünf.\n", foo);
11     } else if (foo > 10) {
12         printf("Diese Zeile wird nie ausgegeben.\n");
13     } else {
14         printf("%d ist kleiner als Fünf.\n" , foo);
15     }
16 }
```

Ausführungsbeispiel: if-Block mit unerreichbarem Code

```
Bitte geben Sie eine Ganzzahl ein:
50
50 ist größer als fünf.
```

Die Prüfung `foo > 10` findet wegen `else` nur dann statt, wenn `foo > 5` den Wahrheitswert *false* hatte. Für `foo` größer als 10 ist dies natürlich ausgeschlossen.

Ausdruck auf Verschiedenheit von 0 prüfen (1)

Sehr häufig kommt der Fall vor, in dem ausgeschlossen werden soll, dass mit einem Wert 0 gearbeitet wird. Betrachten Sie folgendes Beispiel:

Beispiel: Ausschluss des Wertes 0

```
1  #include <stdio.h>
2
3  int main () {
4      unsigned int playerCount = 0;
5
6      printf("Bitte geben Sie die Anzahl der Spieler ein:\n");
7      scanf("%u", &playerCount);
8
9      if (playerCount != 0) {
10         // Code für das Spiel
11     } else {
12         printf("Ungültige Eingabe:\n");
13     }
14 }
```

Wie Sie wissen, ist jede Ganzzahl ein Wahrheitswert. Da nur die 0 als Wahrheitswert *false* interpretiert wird, kann der explizite Vergleich (`!= 0`) hier entfallen:

Beispiel: Ausschluss des Wertes 0 ohne expliziten Vergleich

```
9     if (playerCount) {
10         // Code für das Spiel
11     } else {
```

5.3. Logische Operatoren

Nicht selten soll die Ausführung von Codeteilen von mehreren Teilbedingungen abhängen. Es ist grundsätzlich möglich, dies durch verschachtelte oder hintereinander gestellte *if*-Blocks zu realisieren. Beispiel: Es soll nur auf Eingaben zwischen 5 und 10 reagiert werden. Dies bedeutet, dass zwei Bedingungen erfüllt sein müssen: die Eingabe soll größer oder gleich 5 sein *und* kleiner oder gleich 10. Bisher können wir dies durch verschachtelte *if*-Blocks umsetzen:

Beispiel: Reaktion auf Eingaben innerhalb eines Wertebereichs

```
1  #include <stdio.h>
2
3  int main () {
4      int foo = 0;
5
6      printf("Bitte geben Sie eine Ganzzahl ein:\n");
7      scanf("%d", &foo);
8
9      if (foo >= 5) {
10         if (foo <= 10) {
11             printf("Triggered!\n"); // wird nur ausgeführt falls 5 <= foo <= 10
12         }
13     }
14 }
```

Übersichtlicher und knapper wird dies jedoch, wenn wir für die Verknüpfung der Bedingungen den AND-Operator benutzen. Wir können das zum einen mit dem bitweisen AND lösen, das wir bereits aus Abschnitt 3.2 kennen:

```
if ((foo >= 5) & (foo <= 10))
```

Hier werden zuerst die Ausdrücke `foo >= 5` und `foo <= 10` ausgewertet; die Ergebnisse sind jeweils entweder 0 oder 1, das heißt nur das Bit mit der niedrigsten Wertigkeit kann gesetzt sein.

Bei der Arbeit mit Wahrheitswerten sollte man allerdings statt der *bitweisen* Operatoren besser *logische* Operatoren verwendet werden. Diese setzen dieselben Prinzipien um, bearbeiten aber nicht jedes Bit des Werts einzeln, sondern unterscheiden nur zwischen 0 (*false*) und ungleich 0 (*true*). Das *logische* AND wird im Code als *doppeltes* `&&` gesetzt. Wir können das obige Beispiel also vereinfachen zu:

Beispiel: Reaktion auf Eingaben innerhalb eines Wertebereichs mit logischem AND

```
1  #include <stdio.h>
2
3  int main () {
4      int foo = 0;
5
6      printf("Bitte geben Sie eine Ganzzahl ein:\n");
7      scanf("%d", &foo);
8
9      if ((foo >= 5) && (foo <= 10))
10         printf("Triggered!\n");
11     }
12 }
```

Logische Operatoren arbeiten marginal schneller als bitweise Operatoren. Der Effekt ist nur unterschiedlich, wenn sie auf andere Werte als 0 und 1 angewandt werden. Betrachten wir dazu folgendes Beispiel:

Beispiel: Unterschied von logischem und bitweisem AND

```
1  #include <stdio.h>
2
3  int main () {
4      int foo = 5,    // binär 101
5          bar = 2;    // binär 010
6
7      if (foo & bar) {
8          printf("bitweises AND: true\n");
9      } else {
10         printf("bitweises AND: false\n");
11     }
12
13     if (foo && bar) {
14         printf("logisches AND: true\n");
15     } else {
16         printf("logisches AND: false\n");
17     }
18 }
```

Ausführungsbeispiel: Unterschied von logischem und bitweisem AND

```
bitweises AND: false
logisches AND: true
```

Die „Ausdrücke“ `foo` und `bar` haben jeweils keine Bits gleicher Wertigkeit, die beide gesetzt sind (in der Binärdarstellung in den Zeilen 4 und 5 stehen keine zwei Einsen untereinander). Daher wird das *bitweise* AND zu 0 ausgewertet – *false*.

Für das *logische* AND dagegen wird nur festgestellt, dass sowohl 5 als auch 2 von 0 verschieden sind. Beide „Ausdrücke“ sind *true*. Daher ist auch die Verknüpfung der Ausdrücke durch das logische AND *true*.

Neben dem logischen AND existiert auch das logische OR (`||`) und das logische NOT (`!`). Das logische

XOR ist im Verhalten gleich mit dem Ungleichheits-Operator `!=`. Tabelle 5.2 fasst nochmals diese Operatoren zusammen.

| Operation | Bitweiser Operator | Logischer Operator |
|-----------|--------------------|-------------------------|
| AND | <code>&</code> | <code>&&</code> |
| OR | <code> </code> | <code> </code> |
| XOR | <code>^</code> | <code>!=</code> |
| NOT | <code>~</code> | <code>!</code> |

Tabelle 5.2.: Bitweise und logische Operatoren in C

Besonders bei Oder-Verknüpfungen zählt es sich aus, mit dem logischen OR zu arbeiten. Vergleichen Sie die beiden folgenden Beispiele, in denen jeweils geprüft wird, ob die Spielerzahl außerhalb des erlaubten Bereichs liegt:

Gültigkeitsprüfung: Reihe von ifs

```

1  #include <stdio.h>
2
3  int main () {
4      unsigned int playerCount = 0;
5
6      printf("Spieleranzahl:\n");
7      scanf("%u", &playerCount);
8
9      if (playerCount < 2) {
10         printf(
11             "Ungeeignet für %d ",
12             playerCount
13         );
14         printf("Spieler.\n");
15     }
16
17     if (playerCount > 5) {
18         printf(
19             "Ungeeignet für %d ",
20             playerCount
21         );
22         printf("Spieler.\n");
23     }
24 }
```

Gültigkeitsprüfung: logisches OR

```

1  #include <stdio.h>
2
3  int main () {
4      unsigned int playerCount = 0;
5
6      printf("Spieleranzahl:\n");
7      scanf("%u", &playerCount);
8
9      if ((playerCount < 2) ||
10         (playerCount > 5) )
11     {
12         printf(
13             "Ungeeignet für %d ",
14             playerCount
15         );
16         printf("Spieler.\n");
17     }
18 }
```

Nicht nur ist der Code auf der linken Seite merklich länger; dieselben Anweisungen müssen für beide Teilbedingungen doppelt gesetzt werden. Solcher *redundanter* Code sollte immer vermieden werden. Wenn Sie im Verlauf eines Projektes Änderungen machen, müssen Sie diese auch in beiden `if`-Blocks durchführen. Der zweite Block wird schnell vergessen – es ergibt sich eine Fehlerquelle.

Ausdruck auf Verschiedenheit von 0 prüfen (2)

Wir hatten bereits gesehen, dass wir den Vergleich `!= 0` fallen lassen können. Wenn ein Codeteil nur genau dann ausgeführt werden soll, wenn ein Ausdruck gleich 0 ist, können wir dies mit dem *logischen* NOT (!) erreichen:

Beispiel: Fehlermeldung bei Eingabe 0

```
1  #include <stdio.h>
2
3  int main () {
4      unsigned int tableLength = 0;
5
6      printf("Anzahl der Zeilen:\n");
7      scanf("%u", &tableLength);
8
9      if (!tableLength) {
10         printf("Fehler: Kann keine leere Tabelle anlegen\n");
11     }
12 }
```

Hier wird zuerst geprüft, ob die Variable `tableLength` von 0 verschieden ist; diese Information wird dann negiert. Effektiv ersetzt der Ausdruck `!tableLength` damit den Vergleich mit 0. Ob diese Darstellung gegenüber `tableLength == 0` zu bevorzugen ist, liegt im Auge des Betrachters. Ähnliche Ausdrücke finden sich jedoch häufig in der Praxis; man sollte also als ProgrammierIn die Darstellung kennen und interpretieren können.

Fehlerquelle: Vergleichsoperator `==` vs. Zuweisungsoperator `=`

Ein häufiger Fehler ist es, den Zuweisungsoperator `=` anstelle des Vergleichsoperators `==` zu setzen. Im Bedingungs-Teil von `if`-Blocks ist dies besonders kritisch, da sich ausführbarer Code ergibt, der ein völlig unintuitives und fehlerhaftes Verhalten erzeugt. Betrachten Sie das folgende Beispiel:

Beispiel: `if` mit versehentlicher Wertzuweisung

```
1  #include <stdio.h>
2
3  int main () {
4      unsigned int tableLength = 0;
5
6      printf("Anzahl der Zeilen:\n");
7      scanf("%u", &tableLength);
8
9      if (tableLength = 0) {
10         printf("Fehler: Kann keine leere Tabelle anlegen\n");
11     }
12
13     printf("Länge der Tabelle: %d", tableLength);
14 }
```

Man könnte erwarten, dieser Code verhielte sich wie schon das Beispiel *Fehlermeldung bei Eingabe 0*. Tatsächlich aber ergibt sich folgendes Verhalten:

Ausführungsbeispiel: `if` mit versehentlicher Wertzuweisung

```
Anzahl der Zeilen:
5
Länge der Tabelle: 0
```

Statt in Zeile 9 `tableLength` mit 0 zu vergleichen wird der Wert 0 in der Variablen gespeichert. Zusätzlich gilt auch der zugewiesene Wert als „Ergebnis“ der Zuweisung: `tableLength = 0` wird also zu 0 ausgewertet und ist damit *false*. Die Meldung Fehler: Kann keine leere Tabelle anlegen wird also nie ausgegeben!

Der Compiler erkennt diesen Fehler und gibt eine entsprechende Warnung aus:

Compilerwarnung: `if` mit versehentlicher Wertzuweisung

```
myProgram.c: In function 'main':
myProgram.c:9:8: warning: suggest parentheses around assignment used as
truth value [-Wparentheses]
    if (tableLength = 0) {
        ~~~~~
```

5.4. Fallunterscheidungen: `switch`

Das Schlüsselwort `switch` wird benutzt, um Fallunterscheidungen mit vielen einzelnen Fällen umzusetzen. Die Form von `switch`-Blöcken lautet:

Syntax: `switch`

```
switch (Ausdruck) {
    case Wert1 :
        Anweisungen;
        break;
    case Wert2 :
        Anweisungen;
        break;
    ...
    default:
        Anweisungen;
        break;
}
```

`Ausdruck` ist dabei ein beliebiger Ausdruck, der zu einer *Ganzzahl* ausgewertet werden kann. Fließkommawerte oder andere Datentypen werden von `switch` leider nicht unterstützt. Dasselbe gilt für `Wert1`, `Wert2`, ...; jedoch müssen diese Ausdrücke bereits zur *Compile-Zeit* feststehen. Das bedeutet, dass `Wert1`, `Wert2`, ... keine Variablen oder Elemente enthalten dürfen, die erst bei Ausführung des Programms (zur *Laufzeit*) feststehen.

Wenn die Aussage `Ausdruck == Wert1` wahr ist, wird der Code unter der entsprechenden `case`-Zeile ausgeführt. Entsprechendes gilt für `Wert2`, ... Gilt für keinen der angegebenen Werte Gleichheit, so

werden die Anweisungen unter **default** ausgeführt.

In der Anwendung kann dies so aussehen:

Beispiel: Menü mit switch

```
1  #include <stdio.h>
2
3  int main () {
4      int selection = -1;
5
6      printf("Bitte wählen Sie einen Menüpunkt:\n");
7      printf("  1) Spiel starten\n");
8      printf("  2) Optionen\n");
9      printf("  3) Highscore zeigen\n");
10     printf("  0) Beenden\n");
11
12     scanf("%d", &selection);
13
14     switch (selection) {
15         case 1 :
16             // Code für: Spiel Starten
17             break;
18         case 2 :
19             // Code für: Optionen
20             break;
21         case 3 :
22             // Code für: Highscore
23             break;
24         case 0 :
25             // Code für: Beenden
26             break;
27         default:
28             printf("Ungültige Eingabe!\n");
29             break;
30     }
31 }
```

Der **default**-Teil ist optional. Lässt man diesen weg und trifft keine der **case**-Klauseln zu, so wird nichts ausgeführt – die Ausführung des Codes wird am Ende des **switch**-Blocks fortgesetzt.

Die Werte zu den **case**-Klauseln dürfen im selben **case**-Block nur jeweils ein einziges Mal vorkommen.

Man kann sich **switch**-Blocks als Kurzform für **if**-Blocks vorstellen³. Das vorige Beispiel lässt sich auch so programmieren:

³Die tatsächliche Umsetzung ist etwas komplexer und enthält einige Optimierungsschritte, die hier nicht besprochen werden können. Diese interne Umsetzung ist der Grund, weswegen **switch** nur mit Ganzzahlen funktioniert.

Beispiel: Menü mit if

```
1  #include <stdio.h>
2
3  int main () {
4      int selection = -1;
5
6      printf("Bitte wählen Sie einen Menüpunkt:\n");
7      printf("  1) Spiel starten\n");
8      printf("  2) Optionen\n");
9      printf("  3) Highscore zeigen\n");
10     printf("  0) Beenden\n");
11
12     scanf("%d", &selection);
13
14     if (selection == 1) {
15         // Code für: Spiel Starten
16     } else if (selection == 2) {
17         // Code für: Optionen
18     } else if (selection == 3) {
19         // Code für: Highscore
20     } else if (selection == 0) {
21         // Code für: Beenden
22     } else {
23         printf("Ungültige Eingabe!\n");
24     }
25 }
```

Wichtig ist das Schlüsselwort **break**. Mit diesem Befehl wird eine Kontrollstruktur wie ein **switch**-Block verlassen und die Codeausführung wird hinter dem aktuellen Block fortgesetzt. Betrachten Sie das folgende Beispiel:

Beispiel: switch ohne break

```
1  #include <stdio.h>
2
3  int main () {
4      int x = 1;
5
6      switch (x) {
7          case 0 :
8              printf("0\n");
9          case 1 :
10             printf("1\n");
11          case 2 :
12             printf("2\n");
13          case 3 :
14             printf("3\n");
15          default :
16             printf("d\n");
17      }
18 }
```

Das Ergebnis dieses Codes ist:

Ausführungsbeispiel: `switch` ohne `break`

```
1
2
3
d
```

Wie zu erwarten springt die Ausführung mit `switch` von Zeile 6 nach Zeile 9. Da hier aber keine `breaks` gesetzt wurden, „fällt die Ausführung durch die `case`-Klauseln“. Das bedeutet, dass am Ende der `case`-Klausel 1 die Code-Ausführung in Zeile 11 fortgesetzt wird, und somit alle `printf`-Anweisungen ausgeführt werden. Wird der Compiler mit der Option `-Wimplicit-fallthrough` gestartet, so finden Sie in der Compiler Ausgabe eine entsprechende Warnung:

Compiler-Warnung: `switch` ohne `break`

```
myProgram.c: In function 'main':
myProgram.c:8:7: warning: this statement may fall through
[-Wimplicit-fallthrough=]
    printf("0\n");
    ^~~~~~
myProgram.c:10:5: note: here
    case 1 :
    ^~~~
...
```

Kontrollstrukturen wie `switch` und `if` können (nahezu) beliebig tief ineinander verschachtelt werden.

5.5. Kombinierte Fallunterscheidung und Wertzuweisung – der Ternäre Operator ?

Nicht selten soll der Wert einer Variablen von einer Bedingung abhängen. Wir kennen bisher die Form

Beispiel: Bedingte Wertzuweisung mit `if`

```
1  int main () {
2      int Bedingung = 1, Variable;
3
4      if (Bedingung) {
5          Variable = 1;
6      } else {
7          Variable = 2;
8      }
9  }
```

Diese von einer Bedingung abhängige Wertzuweisung kann auch kompakt in einer einzelnen Zeile geschrieben werden. Mit dem *ternären Operator*⁴ ? lässt sich dieser Code auch schreiben als:

⁴von *ternär*: das Dritte. Dieser Operator braucht drei *Argumente*. Die Addition + ist beispielsweise ein *binärer* Operator, da sie zwei Argumente braucht – eben die beiden Zahlen, die addiert werden sollen.

Beispiel: Bedingte Wertzuweisung mit dem ternären Operator

```
1  int main () {  
2      int Bedingung = 1, Variable;  
3  
4      Variable = Bedingung ? 1 : 2;  
5  }
```

Wie bereits bei `if` ist `Bedingung` ein Ausdruck, dem ein Wahrheitswert zugeordnet werden kann. Ist `Bedingung` erfüllt, so wird der Ausdruck direkt hinter dem `?` ausgewertet und zugewiesen. Andernfalls wird der Ausdruck hinter dem `:` verwendet.

Variablen, die in `Bedingung` vorkommen, können auch in den Rückgabewerten vorkommen. So lässt sich beispielsweise die Betragsfunktion folgendermaßen implementieren:

Beispiel: Betrag einer Zahl mit dem ternären Operator

```
1  int main () {  
2      int x = -42;  
3  
4      x = x >= 0 ? x : -x;  
5  }
```

6. Die CPP-Referenz am Beispiel der math-library

Do not worry about your difficulties in Mathematics. I can assure you mine are still greater.

Albert Einstein

Zu vielen mathematischen Problemen wie Berechnung der Wurzel oder des Sinus einer Zahl stehen in der math-library `libm`¹ Lösungen bereit. Ich möchte Sie dazu einladen, diese Funktionen selbst zu erkunden. Dazu wird uns die Befehlsreferenz unter <https://en.cppreference.com/w/c> dienen. In diesem Kapitel lernen Sie, sich in der Befehlsreferenz zurecht zu finden und nützliche Features selbst zu entdecken.

6.1. Überblick über die Seite [cppreference.com](https://en.cppreference.com)

Wenn Sie die Seite <https://en.cppreference.com/w/c> besuchen, sollten Sie ein Bild wie in Abbildung 6.1 sehen.

Der angegebene Link führt Sie auf die *englische* Befehlsreferenz. Übersetzungen in verschiedene andere Sprachen, darunter auch Deutsch, stehen zur Verfügung. In der Regel sind diese aber weit weniger ausführlich und oft unvollständig². Wenn Sie dennoch mit der deutschen Version arbeiten möchten, finden Sie auf jeder Seite der Referenz am unteren Rand die Zeile *In other languages* und dahinter Links, die sie zu einer entsprechenden Übersetzung führen.

Am rechten oberen Rand befindet sich eine Textbox, in die Sie einen Suchbegriff eingeben können. Die Suche liefert ihnen Ergebnisse zu ihrem Begriff sowie zu „ähnlichen Ausdrücken“. Die Ergebnisse sind in zwei Spalten sortiert und betreffen die Sprache C++ (links) bzw. C (rechts). In Abbildung 6.2 finden Sie das Ergebnis der Suche nach dem Befehl `printf`. Jede Zeile der Suchergebnisse ist ein Link auf einen Artikel, der den Befehl oder das Konzept erklärt.

Ein solcher Artikel sieht i. d. R. aus wie in Abbildung 6.3. Befehle, die sich sehr ähnlich verhalten werden teils zu einem einzelnen Artikel zusammengefasst. Nach der Überschrift finden Sie die Zeile *Defined in header* `<stdio.h>`, die Ihnen mitteilt, welche *#includes* Sie setzen müssen, um den oder die erklärten Befehle in Ihren Programmen nutzen zu können.

Weiter schließt sich eine Syntax-Übersicht an, in der die erwarteten Parameter sowie ihre Datentypen aufgelistet werden. Manche dieser Formen standen nicht in der „Urform“ der Sprache C zur Verfügung. Für solche Features, die erst mit einem bestimmten Standard eingeführt wurden ist in der Referenz in grün das Einführungsdatum bzw. das „Verfallsdatum“ der Form aufgeführt. Ich erinnere Sie daran, dass wir hier den Standard C11 besprechen. In Abschnitt 6.3 werden wir anhand eines übersichtlicheren Beispiels auf die Struktur der Artikel eingehen.

¹Wie in Abschnitt 1.3 besprochen, muss dem Compiler mitgeteilt werden, wenn eine Bibliothek verwendet wird. Während allgemein von der *math-library* gesprochen wird, heißt die einzubindende Datei leider einfach nur `m` – ein unschöner Umstand, mit dem wir leben müssen.

²Meiner Erfahrung nach gilt dies generell in der Welt des Programmierens: Nützliche Ressourcen sind oft nur in Englisch verfügbar oder nur in dieser Sprache in vollem Umfang.

cppreference.com Create account

Page Discussion View View source History

c

C reference

| | | |
|--|---|--|
| Language Basic concepts C Keywords Preprocessor Expressions Declaration Initialization Functions Statements Headers | Type support Program utilities Variadic functions Error handling Dynamic memory management Date and time utilities Strings library Null-terminated strings: byte – multibyte – wide Algorithms | Numerics Common mathematical functions Floating-point environment (C99) Pseudo-random number generation Complex number arithmetic (C99) Type-generic math (C99) Input/output support Localization support Atomic operations library (C11) Thread support library (C11) |
|--|---|--|

Technical specifications
Dynamic memory extensions (dynamic memory TR)
Floating-point extensions, Part 1 (FP Ext 1 TS)
Floating-point extensions, Part 4 (FP Ext 4 TS)

External Links – Non-ANSI/ISO Libraries – Index – Symbol Index

Support us Recent changes FAQ Offline version
 What links here Related changes Upload file Special pages Printable version Permanent link Page information
 In other languages Český Deutsch Español Français Italiano 日本語 한국어 Português Русский 中文
 This page was last modified on 3 July 2017, at 20:56.
 Privacy policy About cppreference.com Disclaimers




 Powered By MediaWiki
  Powered by GeSHi
  Tiger Hosting

Abbildung 6.1.: Die Startseite der C-Referenz <https://en.cppreference.com/w/c>

cppreference.com Create account

Special page

Search results

Search

| | |
|---|----------------------------|
| C++ std::printf std::ctype::print std::ctype_base::print std::ctype_byname::print std::fprintf std::sprintf std::vprintf std::wprintf std::rint | C printf rint |
|---|----------------------------|

Abbildung 6.2.: Suchergebnisse in der CPP-Referenz

cppreference.com

Create account

Search

Page

Discussion

View

Edit

History

C

File input/output

printf, fprintf, sprintf, snprintf, printf_s, fprintf_s, sprintf_s, snprintf_s

Defined in header <stdio.h>

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

(1) (until C99)

int printf(const char *restrict format, ...);

(since C99)

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

(2) (until C99)

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

(since C99)

int sprintf(char *buffer, const char *format, ...);

(3) (until C99)

int sprintf(char *restrict buffer, const char *restrict format, ...);

(since C99)

int snprintf(char *restrict buffer, size_t bufsz, const char *restrict format, ...);

(4) (since C99)

int printf_s(const char *restrict format, ...);

(5) (since C11)

int fprintf_s(FILE *restrict stream, const char *restrict format, ...);

(6) (since C11)

int sprintf_s(char *restrict buffer, rsize_t bufsz, const char *restrict format, ...);

(7) (since C11)

int snprintf_s(char *restrict buffer, rsize_t bufsz, const char *restrict format, ...);

(8) (since C11)

Loads the data from the given locations, converts them to character string equivalents and writes the results to a variety of sinks.

- 1) Writes the results to the output stream `stdout`.
- 2) Writes the results to the output stream `stream`.

Abbildung 6.3.: Anfang des Artikels zu `printf` auf der CPP-Referenz

6.2. Funktionen zu einer Aufgabe finden

Die Funktionen, die die Sprache C zur Verfügung stellt, sind in thematisch verwandten *Bibliotheken* organisiert. Zu jeder Bibliothek existiert ein *Header*, dessen Name einen Hinweis auf die darin enthaltenen Funktionen gibt. Um Lösungen zu einem gegebenen Problem zu suchen, durchsucht man also am besten die Liste der Header. Diese kann von der Hauptseite aus über den Link *Headers* erreicht werden (siehe Abbildung 6.1, erste Spalte unten). Dieser Link führt Sie zu einer Ansicht wie in Abbildung 6.4.

In diesem Kapitel wollen wir uns mit mathematischen Funktionen befassen; wir klicken daher auf den Link *Common mathematical functions*.

Wie Sie in Abbildung 6.5 sehen, sind die Funktionen in einem ähnlichen Format aufgelistet wie schon die Header. Zu vielen Aufgaben stehen mehrere Funktionen zur Verfügung. Dies hat den Grund, dass die Sprache C *strong-typed* ist, d. h. dass Unterschiede zwischen beispielsweise **float**- und **double**-Werten bestehen. Funktionen, die mit einem **f** beginnen, berechnen i. d. R. einen **float**-Wert; der Anfang **l** weist auf einen **long**-Wert hin und **ll** erzeugt einen **long long**-Wert. Für manche Kontexte sind Ganzzahlen unsinnig; dort weist der Suffix **l** auf **long double** hin. Ist dem Funktionsnamen kein solcher *Präfix* vorangestellt, wird ein **double**-Wert berechnet³.

Aufgrund des *strong typing* muss auch für die Argumente der Funktionen unterschieden werden. Hier geben die *Suffixe* **f**, **l** und **ll** den Datentyp des Arguments an.

³Die Benennungs-Logik der C-Standardbibliotheken folgt oft noch veralteten Konventionen, die solche kryptischen Namen hervorbringen. Moderne ProgrammiererInnen bevorzugen „klare Namen“. Damit auch ältere Code-Teile weiterhin funktionieren musste diese Benennungsregeln fortgeführt werden.

cppreference.com

Create account

Search

Q

Page

Discussion

View

Edit

History

C

C Standard Library header files

| | |
|--------------------------|---|
| <assert.h> | Conditionally compiled macro that compares its argument to zero |
| <complex.h> (since C99) | Complex number arithmetic |
| <ctype.h> | Functions to determine the type contained in character data |
| <errno.h> | Macros reporting error conditions |
| <fenv.h> (since C99) | Floating-point environment |
| <float.h> | Limits of float types |
| <inttypes.h> (since C99) | Format conversion of integer types |
| <iso646.h> (since C95) | Alternative operator spellings |
| <limits.h> | Sizes of basic types |
| <locale.h> | Localization utilities |
| <math.h> | Common mathematics functions |

Abbildung 6.4.: Liste der Header auf der CPP-Referenz

cppreference.com

Create account

Search

Page

Discussion

View

Edit

History

C

Numerics

Common mathematical functions

Common mathematical functions

Functions

Defined in header <stdlib.h>

abs

labs

llabs (C99)

computes absolute value of an integral value ($|x|$)

(function)

div

ldiv

lldiv (C99)

computes quotient and remainder of integer division

(function)

Defined in header <inttypes.h>

computes absolute value of an integral value ($|x|$)

Abbildung 6.5.: Funktionen der math-library in der CPP-Referenz

Beispielsweise finden Sie die Funktion `abs`, die den Absolutbetrag einer **double**-Zahl berechnet und als **double**-Wert zurückgibt. Daneben ist aber auch u. a. die Funktion `fabsf` aufgelistet, die ebenso den Absolutbetrag einer Zahl berechnet. Die Zahl wird für diese Funktion jedoch als **long**-Wert angegeben und das Ergebnis als **float** berechnet.

6.3. Der Artikel zu `sqrt`

Abbildung 6.6 zeigt den Anfang des Artikels zum Befehl `sqrt`. Die Zeile *Defined in header <math.h>* weist darauf hin, dass wir `#include <math.h>` setzen müssen, um `sqrt` benutzen zu können. Von dem Befehl stehen drei Varianten zur Verfügung, von denen zwei erst mit dem Standard C99 eingeführt wurden (Punkte 1 bis 3). Weiterhin wurde mit dem Standard C99 ein *Makro* eingeführt (Punkt 4), das wir an dieser Stelle noch nicht bearbeiten wollen. In Kapitel 14 besprechen wir die hierzu relevanten

sqrt, sqrtf, sqrtl

| | | |
|------------------------------|---------------------------|-----------------|
| Defined in header <math.h> | | |
| float | sqrtf(float arg); | (1) (since C99) |
| double | sqrt(double arg); | (2) |
| long double | sqrtl(long double arg); | (3) (since C99) |
| Defined in header <tgmath.h> | | |
| #define | sqrt(arg) | (4) (since C99) |

- 1-3) Computes square root of arg.
- 4) Type-generic macro: If arg has type `long double`, `sqrtl` is called. Otherwise, if arg has integer type or the type `double`, `sqrt` is called. Otherwise, `sqrtf` is called. If arg is complex or imaginary, then the macro invokes the corresponding complex function (`csqrtf`, `csqrt`, `csqrtl`).

Parameters

arg - floating point value

Return value

If no errors occur, square root of arg (\sqrt{arg}), is returned.

If a domain error occurs, an implementation-defined value is returned (NaN where supported).

If a range error occurs due to underflow, the correct result (after rounding) is returned.

Error handling

Errors are reported as specified in `math_errhandling`.

Domain error occurs if arg is less than zero.

Abbildung 6.6.: Anfang des Artikels zu sqrt in der CPP-Referenz

Techniken. Für den Moment können Sie den Punkt ignorieren⁴.

Unter der tabellarischen Übersicht der im Artikel besprochenen Funktionen finden Sie eine Kurzbeschreibung des Effekts der einzelnen Formen. *1-3) Computes square root of arg.* – alle hier besprochenen Funktionen berechnen also die Quadratwurzel eines Arguments `arg`. Der Unterschied ist der Datentyp des Ergebnisses bzw. des Arguments.

Unter *Parameters* finden Sie genauere Erläuterungen zu den Argumenten, die den Funktionen übergeben werden können. Für `sqrt` ist dies die nüchterne Erklärung, dass `arg` eine beliebige Fließkommazahl beschreibt. Bei der Funktion `printf` wird hier etwa aufgelistet, wie ein Formatstring auszusehen hat.

Funktionen berechnen einen Wert, der dann einer anderen Variablen zugewiesen werden kann. Betrachten Sie das folgende Beispiel:

Beispiel: Betrag einer Zahl mit dem ternären Operator

```
1 #include <math.h>
2
3 int main () {
4     double root = sqrt(81.0);
5 }
```

⁴Wenn Sie später im Kursverlauf nochmals diesen Artikel in der Referenz lesen, beachten Sie bitte, dass für die Benutzung des Makros der Header `<tgmath.h>` eingebunden werden muss.

Wir benutzen die Funktion `sqrt` um aus dem `double`-Wert `81.0` die Wurzel zu berechnen und speichern diesen Wert der Wurzel dann in der Variablen `root`. Dieser berechnete Wert wird als Rückgabewert (*Return Value*) der Funktion bezeichnet.

Der Abschnitt *Return Values* gibt Details zu dem berechneten Wert unter der Annahme verschiedener Szenarios.

Wenn kein Fehler auftritt, wird die Quadratwurzel von `arg` berechnet. Wie Sie wissen, existiert aber nicht zu jeder Zahl eine Quadratwurzel. So sind negative Werte für `arg` nicht zulässig – dies ist gemeint mit *If a domain error occurs*. In diesem Fall hängt das Ergebnis von der Version des Compilers ab (*an implementation-defined value is returned*). In der Regel wird aber ein spezielles Bitmuster erzeugt, das sicher als „Fehler-Wert“ interpretiert werden kann (*NaN where supported* – *NaN* steht für *Not a Number*, also ein Fehlerwert. Siehe hierzu mehr in Abschnitt 15.3).

Die letzte Zeile *If a range error occurs due to underflow* beschreibt das Verhalten, wenn der Wert von `arg` nicht mehr korrekt mit seinem Datentyp abgebildet werden kann (vgl. Abschnitt B.6 im Anhang – manche Werte sind zu groß oder haben zu viele Nachkommastellen, um beispielsweise als `double` exakt abgebildet zu werden). In diesem Fall wird mit einem gerundeten Wert gerechnet.

Unter *Error Handling* werden weitere Details aufgelistet, die das Verhalten von `sqrt` in „abnormalen“ Situationen beschreiben. Im Allgemeinen ist es nicht nötig, all diese Details im Kopf zu behalten – dafür ist die Referenz da.

Am Ende der meisten Artikel finden Sie zu jedem Befehl einen Beispiel-Code, der die Anwendung der besprochenen Funktion verdeutlicht, sowie ein Ausgabebeispiel.

6.4. Weitere nützliche Funktionen der math-library

Die folgende Tabelle listet sehr gebräuchliche Funktionen sowie eine Kurzbeschreibung auf. Mit der `cpp`-Referenz sollten Sie nun dazu in der Lage sein, diese Funktionen in Ihren Programmen anzuwenden.

| Funktion | Effekt |
|--------------------|---|
| <code>sqrt</code> | Quadratwurzel einer Zahl |
| <code>cbrt</code> | Kubikwurzel einer Zahl |
| <code>pow</code> | Potenz einer Zahl x zu einem Exponenten y : x^y |
| <code>exp</code> | Exponentialfunktion |
| <code>log</code> | Natürlicher Logarithmus |
| <code>sin</code> | Sinus |
| <code>cos</code> | Cosinus |
| <code>tan</code> | Tangens |
| <code>asin</code> | Arcussinus |
| <code>acos</code> | Arcuscosinus |
| <code>atan</code> | Arcustangens |
| <code>atan2</code> | Arcustangens mit Unterscheidung der Quadranten |
| <code>ceil</code> | Zahl aufrunden |
| <code>floor</code> | Zahl abrunden |
| <code>trunc</code> | Nachkommaanteil abschneiden |
| <code>round</code> | Zahl runden |

Tabelle 6.1.: Gängige Funktionen der math-library

6.5. Die C++-Referenz

Ausflug: C++

Wie die URL der Seite schon vermuten lässt, ist die cpp-Referenz auch ein Verzeichnis der Features von C++. Gehen Sie hierzu von der Adresse <https://en.cppreference.com/w/cpp> aus, um Informationen spezifisch zu C++ zu finden.

Sie haben bereits gesehen, dass die Stichwortsuche direkt auf C++-spezifische Artikel verweist. Auf der Startseite der C++-Referenz finden Sie ebenfalls den Punkt *Headers*, von wo Sie auf die spezifischen libraries verwiesen werden. Die bedeutsamsten Bibliotheken sind hier aber schon direkt von der Startseite aus verlinkt.

7. Schleifen

You spin me right round, baby // Right round like a record, baby // Right round round round

Dead Or Alive

Computer können dazu benutzt werden, die immer gleichen Aufgaben wiederholt und in schneller Folge auszuführen. Es ist möglich, bei jeder Wiederholung einen einzelnen Eingabewert zu ändern und so z. B. Berechnungen für einen ganzen Wertebereich durchzuführen, oder Messwerte von einem Gerät zu überwachen.

Zeichnet man ein *Flussdiagramm* eines solchen Programms (wie in Abbildung 7.1), so findet sich in der Regel ein Programmteil, der zur Vorbereitung dient und in gewohnter Weise „von oben nach unten“ abgearbeitet wird. An diesen schließt sich ein Abschnitt an, der einige Male wiederholt werden soll, und daher im Flussdiagramm als Bogen dargestellt wird. Nach diesem Teil könnte die Ausgabe der Ergebnisse stattfinden, die wiederum in gewohnter *linearer* Weise (also von oben nach unten) bearbeitet wird.

Die Form dieses Flussdiagramms motiviert den Namen *Schleife* für eine solche Struktur.

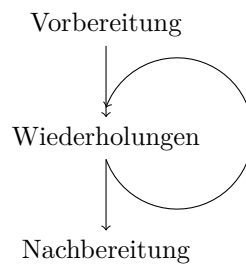


Abbildung 7.1.: Programmflussdiagramm mit Schleife

In der Regel ist die Zahl der Schleifendurchläufe an eine Bedingung geknüpft; genauso sind aber auch *Endlosschleifen* möglich. In diesem Kapitel werden wir verschiedene Schleifentypen und ihre Anwendungsfelder kennen lernen.

Laufende Programme zum Beenden zwingen: **STRG + C**

Macht man einen Fehler bei der Formulierung der Bedingung, so kann man unbeabsichtigt eine Endlosschleife erstellen. Ein solches Programm wird von sich selbst aus nie beendet. Wir können zu jeder Zeit aber das Beenden erzwingen, indem wir in der Konsole die Tastenkombination **STRG + C** drücken.

7.1. Programmsprünge: **goto**

Die einfachste Form, Schleifen zu implementieren, lässt sich mit der *Sprunganweisung* **goto** realisieren. Stellen wir uns vor, der Computer würde mit einem Cursor durch unser Programm laufen und Zeile für

Zeile bearbeiten, so ist **goto** der Befehl, den Cursor an eine bestimmte Stelle zu verschieben. Sprünge sind sowohl vorwärts als auch rückwärts möglich.

Für einen Sprung ist zunächst eine *Sprungmarke* nötig, also ein Punkt, ab der das Programm fortgeführt wird. Eine solche Sprungmarke hat einen Namen, der denselben Regeln folgt, wie Variablennamen (darf nur einmal im Programm vorkommen; Unterscheidung von Groß- und Kleinschreibung; nur alphanumerische Zeichen; darf nicht mit einer Zahl beginnen; maximal 40 Zeichen lang). Im Code wird sie durch einen Doppelpunkt abgeschlossen.

Der Sprung selbst folgt dann der Syntax:

Syntax: **goto**

```
goto Sprungmarke;
```

Hier sehen Sie ein einfaches Anwendungsbeispiel:

Beispiel: Sprünge mit **goto**

```
1  #include <stdio.h>
2
3  int main () {
4      printf("Erste Zeile der Ausgabe.\n");
5      goto ReEntryPoint;
6
7      printf("Dies wird nie ausgegeben.\n");
8
9      ReEntryPoint:
10     printf("Letzte Zeile der Ausgabe.\n");
11 }
```

Nach der Ausgabe in Zeile 4 springt der „Cursor“ weiter zu Zeile 9. Aller Code dazwischen wird nie ausgeführt. Die Ausgabe lautet entsprechend:

Ausführungsbeispiel: Sprünge mit **goto**

```
Erste Zeile der Ausgabe.
Letzte Zeile der Ausgabe.
```

Wir können Sprünge mit **if** an eine Bedingung knüpfen und so eine Schleife erzeugen, die auch wieder verlassen wird:

Beispiel: Quadratwurzeln der Zahlen 1 bis 10 mit **goto**

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main () {
5      double foo = 1;
```

```

6     iterationPoint:
7     printf("Die Wurzel aus %4.1lf ist %lf.\n", foo, sqrt(foo));
8     foo++;
9
10    if (foo <= 10) {goto iterationPoint;}
11
12    printf("Erfolgreicher Programmabschluss.\n");
13 }

```

Ausführungsbeispiel: Quadratwurzeln der Zahlen 1 bis 10 mit `goto`

```

Die Wurzel aus 1.0 ist 1.000000.
Die Wurzel aus 2.0 ist 1.414214.
Die Wurzel aus 3.0 ist 1.732051.
Die Wurzel aus 4.0 ist 2.000000.
Die Wurzel aus 5.0 ist 2.236068.
Die Wurzel aus 6.0 ist 2.449490.
Die Wurzel aus 7.0 ist 2.645751.
Die Wurzel aus 8.0 ist 2.828427.
Die Wurzel aus 9.0 ist 3.000000.
Die Wurzel aus 10.0 ist 3.162278.
Erfolgreicher Programmabschluss.

```

Spaghetti-Code

Was Sie gerade über `goto` gehört haben, dient vor allem dem besseren Verständnis der folgenden Schleifentypen. Programme, die viele `goto`-Sprunganweisungen enthalten werden schnell unübersichtlich. Die einzelnen „Fäden“ des Programmes kreuzen sich und laufen durcheinander wie Spaghetti auf einem Teller. Es wird schwierig, den Programmfluss zu verfolgen und Fehler sammeln sich an.

Die folgenden Schleifentypen sind für Menschen besser zu bedienen, werden aber vom Compiler ebenso in `goto`-Programmsprünge umgesetzt.

Es ist prinzipiell immer möglich, ohne den Befehl `goto` auszukommen und in den weit meisten Fällen auch zu bevorzugen. ProgrammiererInnen diskutieren kontrovers, ob der Befehl aus dem Sprachumfang der Sprache C (oder damit verwandten Sprachen) gestrichen werden sollte. In Abschnitt 7.5 werden Sie allerdings einen Fall sehen, wo `goto` tatsächlich eine *elegantere* Lösung darstellt, als die Alternativen.

7.2. Schleife mit Bedingung: `while`

Das Schlüsselwort `while` lehnt sich an den Sprachgebrauch an, und implementiert die Idee: *FALLS Bedingung erfüllt WIEDERHOLE Anweisungen*. Dabei ist *Bedingung* ein Ausdruck wie schon bei `if`, der zu einem Wahrheitswert ausgewertet werden kann. Die Syntax lautet:

Syntax: `while`

```

while (Bedingung) {
    Anweisungen;
}

```

Mit diesem Schlüsselwort übersetzt sich also das vorige Beispiel zu folgendem Code:

Beispiel: Quadratwurzeln der Zahlen 1 bis 10 mit `while`

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main () {
5      double foo = 1;
6
7      while (foo <= 10) {
8          printf("Die Wurzel aus %4.1lf ist %1f.\n", foo, sqrt(foo));
9          foo++;
10     }
11
12     printf("Erfolgreicher Programmabschluss.\n");
13 }
```

Wie schon bei `if` können bei einzeiligen Anweisungsblocks die {geschweiften Klammern} entfallen. Genauso wie bei `if`-Blocks empfehle ich bei Schleifen immer Klammern zu setzen.

Die Bedingung wird bereits vor Eintritt in die Schleife überprüft. Ist sie vor dem Eintritt nicht erfüllt, wird der Schleifen-Körper nie ausgeführt sondern komplett übersprungen.

Absichtliche Endlosschleife

Es gibt Situationen, in denen eine Schleife bewusst endlos lange laufen soll, z. B. um ein Gerät zu überwachen, das nie abgeschaltet wird, oder weil andere Mechanismen das Verlassen der Schleife bewirken können. Für diesen Fall setzt man für gewöhnlich einfach

```
while (1)
```

Die Konstante 1 kann sich nicht ändern und entspricht direkt dem Wahrheitswert *true*.

7.3. Schleife mit Bedingung und einem garantierten Durchlauf: `do-while`

Schleifen mit dem Konstrukt `do-while` funktionieren nach demselben Prinzip wie `while`-Schleifen. Allerdings findet die Prüfung der Bedingung erst *am Ende der Schleife* statt, und nicht schon vor Eintritt in die Schleife. Dies bewirkt, dass der Schleifenkörper mindestens einmal durchlaufen wird.

Die Syntax ist sehr ähnlich der von `while`:

Syntax: `do-while`

```
do {
    Anweisungen;
} while (Bedingung);
```

Die folgenden beiden Beispiele und Ausgaben verdeutlichen den Unterschied:

Schleife mit while

```
1  #include <stdio.h>
2
3  int main () {
4      unsigned int foo = 0;
5
6      while (foo > 0) {
7          printf("Schleifenkörper.\n");
8      }
9
10     printf("Programmende.\n");
11 }
```

Schleife mit do-while

```
1  #include <stdio.h>
2
3  int main () {
4      unsigned int foo = 0;
5
6      do {
7          printf("Schleifenkörper.\n");
8      } while (foo > 0);
9
10     printf("Programmende.\n");
11 }
```

In beiden Fällen ist die Bedingung `foo > 0` zu keiner Zeit erfüllt. Dennoch sind die beiden Ausgaben unterschiedlich:

Ausführungsbeispiel: Schleife mit while

Programmende.

Ausführungsbeispiel: Schleife mit do-while

Schleifenkörper.
Programmende.

Die **do-while**-Schleife kommt also zum Einsatz wenn der Schleifenkörper garantiert *mindestens einmal* ausgeführt werden soll.

7.4. Zählschleifen: for

Der Schleifen-Typ **while** ist generell für alle Szenarios geeignet. Das häufigste Szenario ist der Fall, in dem eine Variable für jede *Iteration* (d. h. für jeden Durchlauf der Schleife) hochgezählt wird. Zu diesem Zweck existiert die spezielle Form der **for**-Schleife, die eine kompakte Form solcher Konstrukte erlaubt.

Eine **for**-Schleife hat die folgende Form:

Syntax: for

```
for (Startanweisung; Bedingung; Iteration) {
    Anweisungen;
}
```

Wobei **Startanweisung** und **Iteration** jeweils reguläre C-Anweisungen sind und **Bedingung** ein Ausdruck, der zu einem Wahrheitswert ausgewertet werden kann. Diese Struktur wird genauso umgesetzt, als würde eine **while**-Schleife folgender Form programmiert:

Äquivalente Form mit while

```
Startanweisung;
while (Bedingung) {
    Anweisungen;
    Iteration;
}
```

Das heißt, **Startanweisung** wird automatisch vor Schleifenbeginn und **Iteration** am Ende jedes Schleifendurchlaufs ausgeführt. In der Regel ist **Startanweisung** die Zuweisung eines Startwerts zu einer **Zählvariablen**. Mit **Iteration** wird diese Zählvariable dann hochgezählt (oder in geeigneten Schritten verändert). Üblicherweise wird die **Bedingung** durch einen Vergleich der Zählvariable mit einem konstanten Wert oder einer Variable realisiert.

Das vorige Beispiel der Quadratwurzeln bis 10 schreibt sich damit folgendermaßen:

Beispiel: Quadratwurzeln der Zahlen 1 bis 10 mit for

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main () {
5      double foo;
6
7      for (foo = 1; foo <= 10; foo++) {
8          printf("Die Wurzel aus %.1lf ist %lf.\n", foo, sqrt(foo));
9      }
10
11     printf("Erfolgreicher Programmabschluss.\n");
12 }
```

Mit dem Standard C99 wurde auch die Möglichkeit eingeführt, in **Startanweisung** Variablen (plural) zu deklarieren und zu initialisieren. Diese „existieren“ dann allerdings nur innerhalb des Schleifenkörpers und können nach Ende der Schleife nicht mehr verwendet werden (es dürfen jedoch nach der Schleife *neue* Variablen mit demselben Namen angelegt werden). Wir werden in Abschnitt 9.1 mehr hierzu erfahren. Das folgende Beispiel zeigt diese kombinierte Deklaration und Initialisierung:

Beispiel: for-Schleife mit Deklaration

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main () {
5      for (double foo = 1; foo <= 10; foo++) {
6          printf("Die Wurzel aus %.1lf ist %lf.\n", foo, sqrt(foo));
7      }
8
9      printf("Erfolgreicher Programmabschluss.\n");
10 }
```

7.5. Eingriffe in den Kontrollfluss: **break** und **continue**

Es gibt Situationen, in denen eine Schleifen-Ausführung „vorzeitig“ abgebrochen oder ein Teil des Schleifenkörpers „übersprungen“ werden soll. Zu diesem Zweck dienen die Befehle **break** und **continue**.

Wir kennen **break** bereits von **switch**-Blöcken. Dort wurden sie benutzt, um den aktuellen **switch**-Block zu verlassen und mit der Programmausführung an das Ende des Blocks zu springen. Dieselbe Wirkung hat **break** auch bei Schleifen, gleich ob es sich dabei um **for**, **while** oder **do-while** handelt.

Das folgende Beispiel zeigt die Ausgabe von Zahlen und deren Quadrat. Es sollen maximal 100 Zahlen ausgegeben werden, die Ausgabe soll aber stoppen, wenn eine Quadratzahl größer als 1000 ist. Um den bedingten Abbruch der Schleife zu realisieren verwenden wir **break**¹.

Beispiel: Abbruch einer for-Schleife mit break

```
1  #include <stdio.h>
2
3  int main () {
4      for (int foo = 1; foo <= 100; foo++) {
5          printf("%d -> %d.\n", foo, foo * foo);
6          if (foo * foo > 1000) {break;}
7      }
8  }
```

Die Zahlen 1 bis 32 sowie ihre Quadrate (1 bis 1024) werden ausgegeben, bevor die Schleife durch **break** verlassen wird und die Ausgabe stoppt.

continue greift ähnlich in die Ausführung ein; jedoch wird die Schleife nicht verlassen sondern die Ausführung springt zum Schleifenbeginn zurück. Bei **for**-Schleifen wird zuvor noch **Iteration** ausgeführt, d. h. z. B. die Zahlvariable hochgezählt.

Das folgende Beispiel gibt die Zahlen von 1 bis 100 aus mit Ausnahme der durch 5 teilbaren Zahlen:

Beispiel: for mit continue

```
1  #include <stdio.h>
2
3  int main () {
4      for (int foo = 1; foo <= 100; foo++) {
5          if (foo % 5 == 0) {continue;}
6          printf("%d\n", foo);
7      }
8  }
```

¹Natürlich kann dazu auch ein logisches OR benutzt werden; hier soll aber die Anwendung von **break** illustriert werden.

8. Arrays und dynamische Programmierung

If debugging is the process of removing software bugs, then programming must be the process of putting them in.

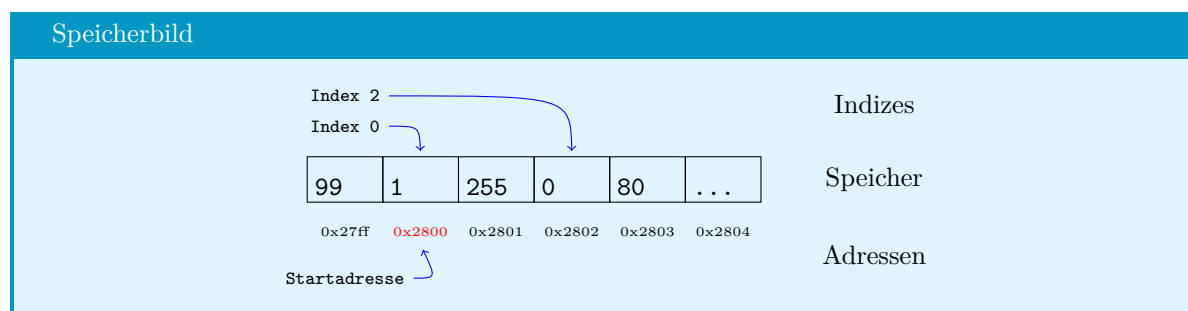
Edsger Dijkstra

Bis hierhin haben wir sehr kleine Datenmengen behandelt. Unser bisheriges „Arbeitsmaterial“ waren Variablen, die einige wenige Bytes halten. Eine Stärke von Computern ist es aber gerade, große Datenmengen schnell zu verarbeiten. Hier werden wir Möglichkeiten kennenlernen, nahezu beliebig große Datenmengen im Speicher zu halten und zu manipulieren.

8.1. Arrays

Arrays (Felder) sind Gruppen von Variablen, die man sich als nummerierte Liste vorstellen kann. Die einzelnen Elemente werden über ihren *Index* angesprochen, also die „Nummer des Eintrags in einer Liste“. Die Nummerierung beginnt hierbei *bei der 0*. Alle Elemente eines Arrays haben denselben Datentyp; man sagt daher auch, das Array selbst habe einen bestimmten Datentyp.

Das Handling von Arrays geschieht über Pointer. Man speichert die Adresse des ersten Elements. Im Speicher liegen alle Werte an direkt aufeinanderfolgenden Adressen. Um etwa das dritte Element eines Arrays anzusprechen (zu lesen oder zu überschreiben) geht man von der *Startadresse* um zwei Schritte weiter. Der Abstand zur Start-Adresse wird auch *Offset* genannt. Bei Arrays, deren Elemente nur ein einzelnes Byte groß sind also Offset und Index gleich.



8.1.1. Syntax-Elemente

Wie alle Variablen müssen auch Arrays zuerst deklariert werden. Dies geschieht in der Form:

Syntax: Arrays deklarieren

```
Datentyp Name[ElementAnzahl];
```

Damit wird ein Array vom Typ `Datentyp` angelegt, das *ElementAnzahl* Elemente hat und über die Variable `Name` angesprochen werden kann. Achtung: Die Variable `Name` selbst ist vom Typ `Datentyp *`, also ein *Pointer auf Daten vom Typ Datentyp*. Man nennt die einzelnen Elemente eines Arrays auch *Instanzen von Datentyp*.

Wir greifen (sowohl lesend als auch schreibend) auf Arrays zu, indem wir in [eckigen Klammern] den Index des Elements schreiben, das wir ansprechen möchten (d. h. seine „Nummer“ in einer Liste). Man spricht von *indiziertem Array-Zugriff*.

Indizierung ab 0

Achtung: Indizes beginnen bei 0. Das bedeutet, dass das letzte gültige Element des Arrays den Index `ElementAnzahl - 1` hat. Der Compiler führt bei Array-Zugriffen keine Gültigkeitskontrolle der Indizes durch. Es ist also möglich, ein Array mit 10 Elementen zu deklarieren, dann aber auf das 11. Element zuzugreifen. Wie bereits in Kapitel 4 erklärt wurde, erzeugt ein Lese- und Schreibzugriff auf *nicht-reservierte Speicherbereiche* undefiniertes Verhalten; mögliche Effekte sind Überschreiben von anderen Variablen, Programmabsturz, aber auch „nichts“, abhängig von der Struktur des Programmes. Auch, wenn ein fehlerhafter Arrayzugriff zunächst keinen Effekt hat, sollten Sie solche Speicherzugriffe unbedingt vermeiden, da sonst spätere Erweiterungen am Programm unter Umständen nicht mehr funktionieren, obwohl diese korrekt programmiert sind. Achten Sie daher genau auf die Codeteile, die Ihre Array-Zugriffe bestimmen.

Beispiel: Array-Zugriffe

```
1  #include <stdio.h>
2
3  int main () {
4      int list[100];
5
6      list[42] = 666;
7
8      printf("Das 43. Element der Liste 'list' ist %d.", list[42]);
9  }
```

Hier wird in Zeile 4 eine Liste angelegt, die 100 `int`-Werte umfasst. Wie auch sonst wird den Elementen des Arrays bei der Deklaration *kein* Startwert zugewiesen – alle 100 Elemente haben also zufällige Startwerte.

Zeile 6 greift schreibend auf das Element mit dem Index 42 zu. Da das erste Element eines Arrays den Index 0 hat, ist dies das 43. *Element* der Liste.

Analog dazu zeigt Zeile 8 einen lesenden Zugriff.

Obi-Wan-Fehler

Der Gegensatz zwischen der menschlichen Zählweise – beginnend bei der 1 – und der in C und C++ – beginnend bei der 0 – führt leider häufig zu *Offset-Fehlern*. Tatsächlich ist der Fehlertyp, sich bei der Indexberechnung um einen Wert 1 zu irren, so häufig, dass er einen eigenen Namen bekommen hat. Aus der englischen Ausdrucksweise *off by one* wurde zuerst die Kurzform *OB1* und hieraus bald die Sprechweise *Obi-Wan*, wie der Charakter aus Star Wars.

Da die Array-Variable nichts weiteres als ein Pointer auf das erste Element des Arrays ist, kann auch der *Dereferenzierungs-Operator* `*` benutzt werden, um das *erste Element* des Arrays zu bearbeiten:

Beispiel: Array-Zugriff mit Dereferenzierungs-Operator

```
1  #include <stdio.h>
2
3  int main () {
4      int list[100];
5
6      *list = 42;
7
8      printf("Das erste Element der Liste 'list' ist %d.", list[0]);
9      // Ausgabe: 42
10 }
```

Dieses Beispiel dient dazu, den inneren Aufbau von Arrays zu verdeutlichen. In der Praxis ist davon abzuraten, da die Verwendung des Dereferenzierungs-Operators `*` die Information „versteckt“, dass die dereferenzierte Variable mehr als ein Element hält. Benutzen Sie auch beim Zugriff auf das erste Element den Array-Index-Zugriff `[0]`.

8.1.2. Initialisierung

Wie alle Variablen werden den Elementen von Arrays bei der Deklaration keine Startwerte zugewiesen. Auch hier existiert jedoch eine Möglichkeit, bei der Deklaration eine *Initialisierung* anzufügen. Dies geschieht über die Syntax:

Syntax: Arrays deklarieren und initialisieren

```
Datentyp Name[ElementAnzahl] = {Wert1, ...};
```

Es wird eine Liste der Initialwerte in {geschweifte Klammern} angeben. Einzelne Elemente werden durch Kommata getrennt. „Fehlende“ Werte werden durch Nullen ergänzt. Betrachten Sie folgendes Beispiel:

Beispiel: Arrays deklarieren und initialisieren

```
1  #include <stdio.h>
2
3  int main () {
4      int list[5] = {10, 20, 30};
5
6      for (int i=0; i<5; i++) {
7          printf("%d: %2d\n", i, list[i]);
8      }
9  }
```

Ausführung: Arrays deklarieren und initialisieren

```
0: 10
1: 20
2: 30
3:  0
4:  0
```

for-Schleifen und Array-Indizes

Iteriert man über die Elemente eines Arrays (d. h. führt man eine Aktion mit jedem Array-Element durch), so bietet sich eine **for**-Schleife an. Die Zählvariable wird dabei mit der Zahl der Elemente verglichen. Gängig und praktisch ist der Vergleich `Zaehlvariable < ElementAnzahl`. Dies ist direkt kompatibel mit der Regel, dass der größte erlaubte Array-Index *um eins kleiner* als die Zahl der Elemente ist.

Eine solche *Initializer-List* muss mindestens einen Wert enthalten. Wollen Sie also ein Array erstellen, das vollständig mit Nullen befüllt ist, so setzen Sie:

Syntax: Arrays deklarieren und null-initialisieren

```
Datentyp Name[ElementAnzahl] = {0};
```

C++: leere *Initializer-List*

In der Sprache C++ ist es erlaubt, eine leere Liste zu übergeben um das Array mit null-Werten zu initialisieren:

C++-Syntax: Arrays deklarieren und null-initialisieren

```
Datatype Name[ElementCount] = {};
```

Dies hat dort jedoch untergeordnete Bedeutung, da die Sprache C++ Konzepte anbietet, die C-Arrays ersetzen und diesen vorzuziehen sind. Als Stichworte seien genannt:

- `std::array` (<https://en.cppreference.com/w/cpp/container/array>)
- `std::vector` (<https://en.cppreference.com/w/cpp/container/vector>)
- `std::list` (<https://en.cppreference.com/w/cpp/container/list>)
- `std::map` (<https://en.cppreference.com/w/cpp/container/map>)

Bei der Verwendung von *Initializer-Lists* ist es auch möglich, die Anzahl der Listenelemente bei der Deklaration auszulassen. Der Compiler erstellt dann ein Array welches alle Werte der *Initializer-List* umfasst (aber nicht mehr). Die Deklaration des Arrays geschieht hier mit leeren eckigen Klammern `[]`:

Syntax: Arrays-Deklaration mit automatischer Bestimmung der Array-Größe

```
Datentyp Name[] = {Wert1, ...};
```

Beispiel: Array mit 3 Elementen ohne explizite Bestimmung der Arraygröße

```
1 int list[] = {1, 2, 3};
```

In Abschnitt 8.4 werden wir eine Möglichkeit kennen lernen, zur *Laufzeit* die Größe eines Arrays zu ermitteln.

8.1.3. Pointer-Arithmetik

Arrays werden über Pointer angesprochen. Pointer sind Ganzzahl-Werte, mit denen gerechnet werden kann. Rechnungen mit Pointer-Variablen verhalten sich aber anders als mit normalen Variablen.

Bei der Addition und Subtraktion wird der Datentyp der Pointer-Variable mit einbezogen: Der addierte Wert wird vor der Addition mit der Größe des Pointer-Grundtyps multipliziert. Betrachten Sie das folgende Beispiel:

Beispiel: Pointer-Arithmetik

```
1  #include <stdio.h>
2
3  int main () {
4      int list[100];
5
6      list[0] = 1;
7      list[1] = 2;
8
9      // addresses
10     printf("base pointer:      %p\n", list    );
11     printf("base pointer plus one:  %p\n", list + 1);
12
13     // pointer dereferencing
14     printf("at start:          %d\n", * list    );
15     printf("at plus one:       %d\n", *(list + 1));
16     printf("at index one:      %d\n",  list  [1]);
17 }
```

Ausgabebeispiel: Pointer-Arithmetik

```
base pointer:      0x7fff8bc1f300
base pointer plus one:  0x7fff8bc1f304
at start:          1
at plus one:       2
at index one:      2
```

Wir lassen uns die Pointer `list` und `list + 1` ausgeben. Obwohl die Konstante 1 addiert wird, unterscheiden sich die Ausgaben um den Wert 4 – der Größe einer `int`-Variablen in Byte. Ändern wir das Beispiel so ab, dass `list` ein `double`-Array wird, so finden wir einen Unterschied von 8. Die Addition geht also um „ganze Speicherstellen“ oder „ganze Feld-Elemente“ weiter, statt um einzelne Bytes.

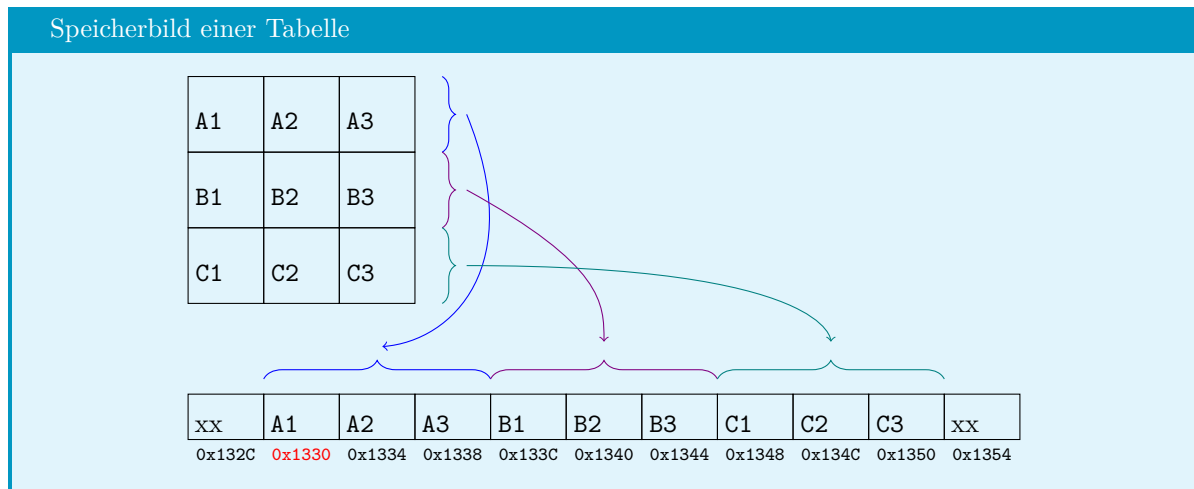
Dies beeinflusst selbstverständlich auch die *Dereferenzierung*. Damit verhält sich die Dereferenzierung eines Pointers nach Addition genauso wie der Index-Zugriffs-Operator `[]`. Entsprechend ist die Ausgabe von Zeile 14 und Zeile 15 identisch.

Die Subtraktion verhält sich analog – auch hier ändert sich die Adresse, auf die der Pointer zeigt nur um „ganze Speicherstellen“. Multiplikation, Division oder andere Operationen sind mit Pointer-Variablen nicht möglich.

8.1.4. Mehrdimensionale Arrays

Wir haben Arrays bislang als eindimensionale Listen kennen gelernt, die im Speicher dicht an dicht gepackt sind. Es ist aber auch möglich, zweidimensionale Tabellen oder höherdimensionale Objekte

abzubilden. Betrachten wir das Beispiel einer zweidimensionalen Tabelle mit Datentyp **int** (4 Byte pro Eintrag):



Tabellen werden im Speicher „geplättet“ an direkt aufeinanderfolgenden Adressen angelegt. Jede Zeile der Tabelle – für sich eine Liste – fügt sich im Speicher an die nächste. Eine zweidimensionale Tabelle kann also als „Liste von Listen“ gesehen werden.

Jede Zelle der Tabelle hat eine Zeilen- und Spaltennummer. Wir wissen, dass die Tabelle 3 Spalten breit ist. Damit finden wir für den Index in der „geplätteten Liste“:

$$\text{Index} = (\text{Breite} * \text{Zeile}) + \text{Spalte}$$

wobei die Zählung von Zeile und Spalte jeweils bei 0 beginnt.

Umgekehrt kann aus der Position einer Zelle in der geplätteten Liste wieder die Koordinate in der Tabelle berechnet werden: Die Zelle mit dem Wert B2 ist die fünfte Zelle der „geplätteten Liste“ im Speicher und hat damit den Index 4. Wir wissen, dass unsere Tabelle eine Breite von 3 Zellen hatte. Mit diesen Informationen können wir ansetzen:

$$\begin{aligned} \text{Zeile} &= \text{Index} / \text{Breite} \\ \text{Spalte} &= \text{Index} \% \text{Breite} \end{aligned}$$

Der Compiler kann solche Strukturen automatisch anlegen. Bei der Deklaration eines mehrdimensionalen Arrays hängen wir für jede Dimension eine eigene [Klammer] an. Ebenso wird beim (lesenden wie schreibenden) Zugriff jede Dimension durch eine eigene Klammer referenziert.

Syntax: Arrays-Deklaration mehrdimensionaler Arrays

```
Datentyp Name[dim1][dim2][...];
```

Syntax: Zugriff auf mehrdimensionale Arrays

```
Name[Index1][Index2][...] = ...;
```

Im Falle einer Tabelle könnte **dim1** dann für die Anzahl der Zeilen und **dim2** für die Anzahl der Spalten der Tabelle stehen.

Mehrdimensionale Arrays können auch direkt bei der Deklaration Initialisiert werden. Hierzu werden verschachtelte {geschweifte Klammern} benutzt. Betrachten Sie das folgende Beispiel:

Beispiel: Tabelle als mehrdimensionales C-Array

```

1  #include <stdio.h>
2
3  int main () {
4      int table[3][4] = {
5          {1, 2, 3, -6},
6          {4, 5, 6, -15},
7          {7, 8, 9, -24}
8      };
9
10     for (int row = 0; row < 3; row++) {
11         for (int col = 0; col < 4; col++) {
12             printf("%d\t", table[row][col]);
13         }
14         printf("\n");
15     }
16
17     printf("\n");
18     table[0][1] = 0;
19
20     for (int row = 0; row < 3; row++) {
21         for (int col = 0; col < 4; col++) {
22             printf("%d\t", table[row][col]);
23         }
24         printf("\n");
25     }
26 }

```

Ausführungsbeispiel: Tabelle als mehrdimensionales C-Array

| | | | |
|---|---|---|-----|
| 1 | 2 | 3 | -6 |
| 4 | 5 | 6 | -15 |
| 7 | 8 | 9 | -24 |
| 1 | 0 | 3 | -6 |
| 4 | 5 | 6 | -15 |
| 7 | 8 | 9 | -24 |

In den Zeilen 4..8 definieren wir eine Tabelle mit 3 Zeilen und 4 Spalten (auch als *3x4-Matrix* bezeichnet). Und befüllen diese direkt mit Werten.

Wir können `table[0]` als „die erste Zeile der Tabelle“ lesen. Dieses Objekt ist selbst eine Liste von vier Zahlen. Entsprechend wird dem Objekt `table[0]` die Liste `{1, 2, 3, -6}` zugewiesen. Analog funktioniert die Zuweisung der nächsten zwei Zeilen.

Mit `printf` geben wir in Zeile 12 das Tabellenelement in der Zeile `row` und der Spalte `col` aus. Diese Variablen werden im Kontext von `for`-Schleifen deklariert und mit Werten befüllt.

In Zeile 18 ändern wir (Zeile 1, Spalte 2) auf den Wert 0 und geben die veränderte Tabelle dann nochmals aus.

Reihenfolge der Dimensionen

Achten Sie bei mehrdimensionalen Arrays auch auf die Reihenfolge der Indizes. Eine Vertauschung kann zu Zugriffen außerhalb Ihrer Tabelle führen und bewirkt ebenfalls wieder undefiniertes Verhalten!

Wenn Sie im oberen Beispiel die Zeile

```
printf("%d\n", table[3][2]);
```

anfügen, so geschieht folgendes:

Der Compiler versucht auf Zeile 4, Spalte 3 zuzugreifen. Nach der zuvor gezeigten Formel wird berechnet:

$$\text{Index} = (\text{Breite} * \text{Zeile}) + \text{Spalte} = (4 * 3) + 2 = 14$$

Unsere Tabelle hat aber nur insgesamt 12 Werte. Der Zugriff geschieht also auf einen nicht-reservierten Speicherbereich und liefert eine „unsinnige“ Ausgabe.

8.2. Dynamische Speicherverwaltung

Die bisher gezeigten Techniken funktionieren, solange zur *Compile-Zeit* bereits bekannt ist, mit wie vielen Elementen gearbeitet werden soll¹. Häufig steht aber erst zur *Laufzeit* fest, wie viele Daten tatsächlich anfallen. Hier wollen wir Techniken *dynamischer Speicherverwaltung* kennenlernen, d. h. „zur Laufzeit“ Felder erstellen, vergrößern und verkleinern und diese Operationen direkt von Usereingaben abhängig machen.

8.2.1. Speicher allozieren und wieder freigeben: malloc, calloc und free

Wir haben im vorigen Abschnitt 8.1 kennen gelernt, wie wir mit einem Pointer auf Feldelemente zugreifen. Für die dynamische Speicherverwaltung benötigen wir also lediglich ein Mittel, einen Pointer auf einen geeignet großen Speicherbereich zu bekommen. Dazu dienen die Befehle `malloc` und `calloc`, die über den Header `stdlib.h` eingebunden werden.

Beide Befehle *allozieren* einen Speicherbereich (d. h. stellen eine Anfrage an das Betriebssystem, Speicher für die ausschließliche Verwendung durch das allozierende Programm zu reservieren) und geben einen Pointer auf den Beginn dieses Speicherbereichs zurück. Der Befehl `calloc` initialisiert diesen Speicherbereich zusätzlich mit Nullen. Der Befehl `malloc` arbeitet schneller, da das auf-null-Setzen entfallen kann. Die Abkürzungen stehen also für *memory allocate* und *clear allocate*.

Die Syntax der beiden Befehle ist ähnlich, leider aber nicht identisch:

Syntax: `malloc`

```
pointer = malloc(AnzahlBytes);
```

Syntax: `calloc`

```
pointer = calloc(AnzahlFelder,  
                BytesProFeld);
```

Hierbei ist `AnzahlBytes` eine vorzeichenlose (d. h. positive) Ganzzahl, die angibt, wie viele Bytes im Speicher benötigt werden. Ähnlich beschreibt `AnzahlFelder` eine vorzeichenlose Ganzzahl, die die Zahl der *Elemente* angibt, für die Speicher alloziert werden soll. `BytesProFeld` ist dann die Größe *eines* Elements in Bytes.

¹Oder, wenn eine Maximalzahl von Elementen bekannt ist, die höchstes auftreten können. Man benutzt dann ein Array dieser Maximalgröße und befüllt nur so viele Einträge, wie tatsächlich Daten anfallen.

Auf die Elemente des so reservierten Speicherbereichs kann über dieselben Syntaxelemente zugegriffen werden, die wir schon aus Abschnitt 8.1 kennen. Selbstverständlich müssen wir auch hier die selbe Sorgfalt bei der Berechnung der Indizes anlegen.

Reservierte Speicherbereiche sollten nach Nutzung wieder freigegeben werden ². Dazu dient der Befehl `free`, der ebenfalls mit dem Header `stdlib.h` eingebunden wird. Die Syntax ist denkbar einfach:

Syntax: `free`

```
free(pointer);
```

Wird reservierter Speicher nicht mehr freigegeben, so steht dieser anderen Prozessen nicht zur Verfügung. Programme laufen langsamer und können potentiell Aufgaben nicht mehr bewältigen, da die notwendige Speicherkapazität „nicht zur Verfügung steht“. Man spricht von *memory leakage*.

Damit können wir das folgende Beispiel verstehen:

Beispiel: Dynamische Speicherverwaltung

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main () {
5      double * array = malloc(88);      // 1 double = 8 byte ==> 11 Speicherstellen
6      double * nulls = calloc(11, 8);  // dito
7      int i;
8
9      for (i=0; i<11; i++) {
10         array[i] = (i * i) / 100.0;
11         printf("%2d    malloc: %4.2lf    calloc: %4.2lf\n", i, array[i], nulls[i]);
12     }
13
14     free(array);
15     free(nulls);
16 }
```

Ausführungsbeispiel: Dynamische Speicherverwaltung

```
0    malloc: 0.00    calloc: 0.00
1    malloc: 0.01    calloc: 0.00
2    malloc: 0.04    calloc: 0.00
3    malloc: 0.09    calloc: 0.00
4    malloc: 0.16    calloc: 0.00
5    malloc: 0.25    calloc: 0.00
6    malloc: 0.36    calloc: 0.00
7    malloc: 0.49    calloc: 0.00
8    malloc: 0.64    calloc: 0.00
9    malloc: 0.81    calloc: 0.00
10   malloc: 1.00    calloc: 0.00
```

²Moderne Betriebssysteme geben bei Beenden eines Programms den von diesem allozierten Speicher automatisch wieder frei, auch wenn dieser nicht explizit vom Programmierer freigegeben wurde. Allgemein gilt es als *guter Stil*, zu jeder Zeit nur soviel Speicher vorzuhalten, wie tatsächlich benötigt wird.

Datentypen und dynamische Speicherverwaltung

Wir hatten bereits angesprochen, dass bei der Arbeit in der Sprache C besonderer Wert auf die Verwendung der korrekten Datentypen zu legen ist. Daher auch einige Kommentare zu diesem Thema:

Die Parameter `AnzahlBytes`, `AnzahlFelder` und `BytesProFeld` sind formell vom Typ `size_t`. Dieser wird im Header `stdlib.h` definiert und ist i. d. R. ein **unsigned long int**, in jedem Fall aber ein vorzeichenloser Ganzzahl-Typ. Im Falle des `gcc` wird ein **unsigned long int** verwendet. Der Rückgabewert von `calloc` und `malloc` ist ein **void ***, also ein Pointer auf einen unspezifizierten Datentyp. Dies macht die Verwendung von `malloc` und `calloc` mit Pointern jeden beliebigen Typs möglich.

Zugriffe nach `free`

Auf Felder, die bereits mit `free` freigegeben werden, darf nicht mehr zugegriffen werden. Das bedeutet, dass weder Lesen (mit dem Dereferenzierungs-Operator `*` oder dem Array-Zugriffsoperator `[]`) noch Schreiben (mit denselben Operatoren) und insbesondere keine „zweite Freigabe“ mit `free` stattfinden darf. Beim Lesen und Schreiben ist das Verhalten undefiniert; bei `free` stürzt das Programm ab.

Praxistipp: `alloca`s und `free` gleichzeitig schreiben

Um in komplexeren Aufgaben nicht zu vergessen, ein Feld freizugeben, habe ich mir angewohnt, direkt nach `malloc` bzw. `calloc` ein zugehöriges `free` zu tippen. Erst danach schreibe ich den Code, der den so reservierten und wieder freigegebenen Speicherbereich auch tatsächlich nutzt.

Sinnvolle und einheitliche Programmlogik

Obwohl `malloc` und `calloc` dieselbe Kernaufgabe erfüllen – das Allokieren von Speicher – unterscheiden sie sich in ihrer Anwendung. Bei `malloc` wird die benötigte *Speichergröße* angegeben, während `calloc` als Parameter die *Feldzahl* und *Feldgröße* erwartet. Dieser Unterschied macht das Arbeiten unangenehm, da man als ProgrammierIn auswendig behalten muss, welcher der sehr ähnlichen Befehle nun welche Form erwartet.

Leider ist es nicht mehr möglich, einen der beiden Befehle in einer neuen Version der Sprache C so zu ändern, dass beide derselben Syntax folgen, da sonst viele ältere Programmcodes nicht mehr mit dem neuen Compiler *kompatibel* sind. Solche *legacy issues* ziehen sich durch die Welt der Informatik und bereiten an verschiedensten Stellen Schwierigkeiten. Einmal etabliert lassen sich auch ungünstige Konventionen aber nicht mehr loswerden, wie auch Abbildung 8.1 illustriert. Mit Blick auf dieses Negativ-Beispiel will ich Sie dazu aufrufen, sich eine einheitliche Struktur und Logik für Ihr Projekt zu überlegen, *bevor Sie die erste Codezeile schreiben*. Wir werden insbesondere in Kapitel 9 auf Möglichkeiten stoßen, die solche Überlegungen erfordern.

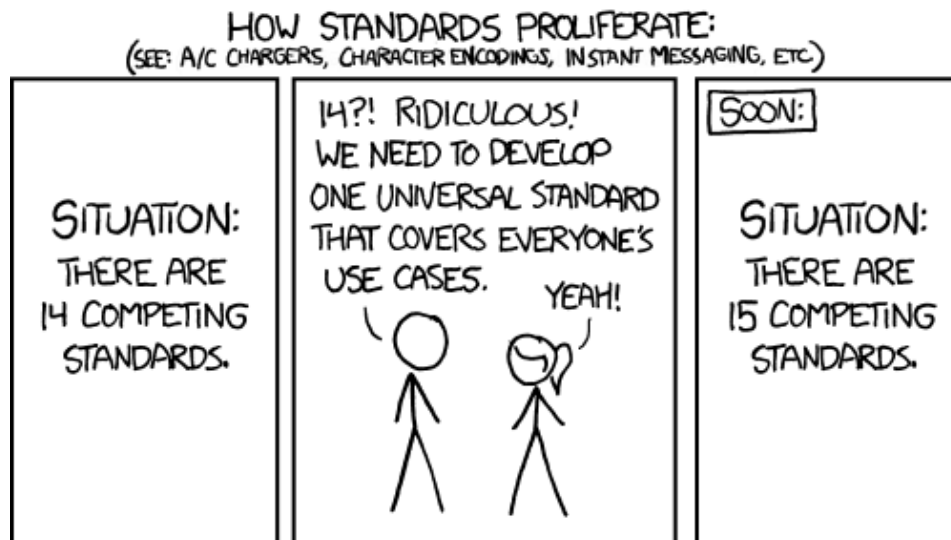


Abbildung 8.1.: Langlebigkeit von Standards in der IT. Quelle: <https://xkcd.com/927/>

Feldgrößen ermitteln mit `sizeof`

C stellt eine Vielzahl von Datentypen bereit, die jeweils andere Speicherbedarfe haben. Anstatt alle diese auswendig zu lernen sollte man mit der Funktion `sizeof` arbeiten, die die Größe jedes beliebigen Typs in Bytes zurück gibt. Im obigen Beispiel können wir damit auch schreiben:

Beispiel: Dynamische Speicherverwaltung und `sizeof`

```
5 double * array = malloc(11 * sizeof(double));
6 double * nulls = calloc(11, sizeof(double));
```

In Abschnitt 8.4 erfahren Sie mehr zu diesem Operator.

Erfolg der Allokierung prüfen

In seltenen Fällen kann die Allokierung von Speicher fehlschlagen. Dies ist insbesondere dann der Fall, wenn nicht mehr ausreichend *zusammenhängender* Speicher zur Verfügung steht, um ein Array der gewünschten Größe darin unterzubringen. In diesem Fall ist der Rückgabewert von `calloc` und `malloc` ein Null-Pointer (d. h. der Wert 0). Auf die Adresse 0 darf nicht zugegriffen werden; jeder Zugriff führt zum Programmabsturz. Daher sollte nach jedem `*alloc`-Befehl eine Fehlerprüfung stattfinden. Mit dem Befehl `exit` aus der `stdlib.h` können wir etwa das Programm beenden. In Kapitel 12 werden wir hierzu mehr hören.

Beispiel: Fehlerprüfung nach Allokierung

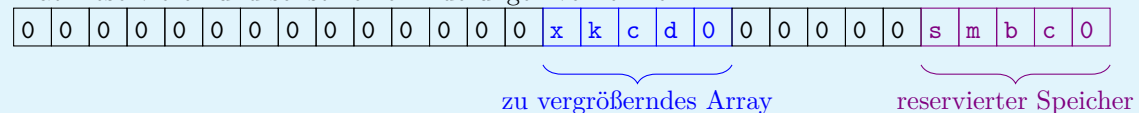
```
1 int * array = malloc(20);
2 if (!array) {
3     printf("Fehler! Speicher konnte nicht alloziert werden!\n");
4     exit(-1); // Programm beenden.
5 }
```

8.2.2. Feldgrößen ändern: realloc

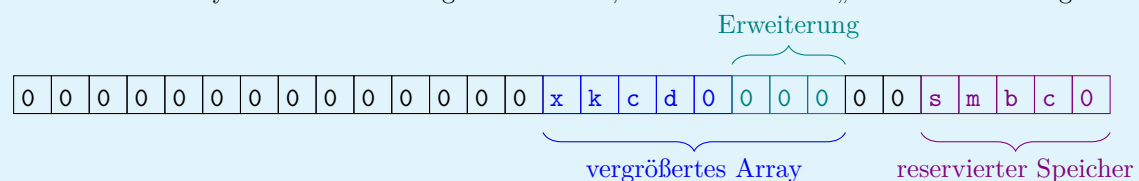
Der Befehl `realloc` (eingebunden über den Header `stdlib.h`) ändert die Größe eines bestehenden Arrays und erhält alle Werte bei (sofern sie in noch „in das neue Array passen“). In der Regel ändert sich der Wert des Pointers auf den Beginn des Arrays nicht. Bei Vergrößerung kann es jedoch sein, dass das Array sonst mit einem anderen reservierten Speicherbereich (z. B. ein anderes Array) überlappt. In diesem Fall wird `realloc` das gesamte Array an eine neue Stelle kopieren.

Visualisierung: Arraygrößen ändern

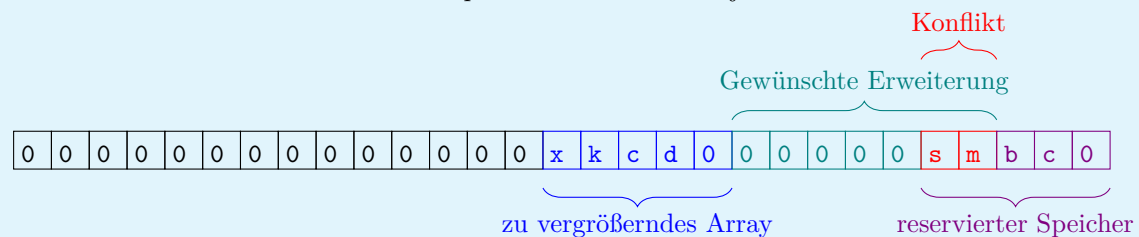
Betrachten wir zuerst den Fall, in dem sich zwischen dem Ende eines zu vergrößernden Arrays und dem nächsten reservierten Speicherbereich viel Platz befindet. `realloc` wird lediglich zusätzlichen Platz reservieren und sonst keine Änderungen vornehmen.



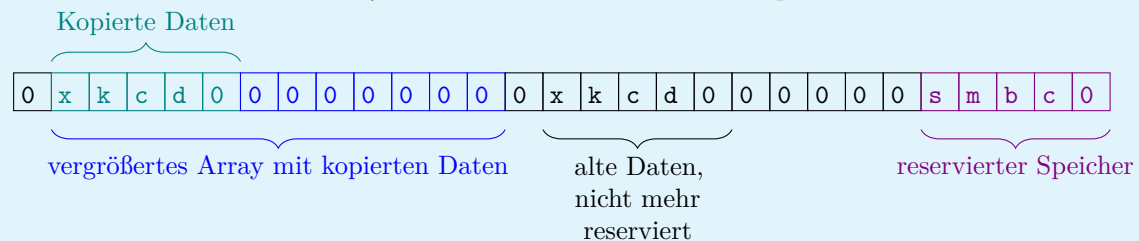
Soll nun das Array um drei Zellen vergrößert werden, so wird es einfach „nach hinten verlängert“:



Bei mehr als fünf Zellen zusätzlichen Speicherbedarf entsteht jedoch ein Konflikt:

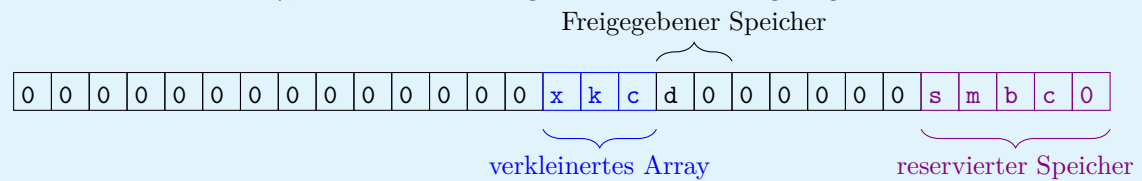


Stattdessen wird also das Array als Ganzes an eine neue Stelle kopiert:



Da die Daten des Arrays vor dem Vergrößern kopiert (im Gegensatz zu verschoben) werden, bleiben diese Werte in den Speicherzellen. Die Zellen sind aber nicht mehr reserviert. Eine folgende Anfrage, Speicher zu allozieren (z. B. mit `malloc`) kann also genau diese Zellen auswählen und für neue Zwecke benutzen.

Verkürzt man ein Array, so wird ebenso lediglich die Reservierung aufgehoben:



In der Syntax orientiert sich `realloc` am Befehl `malloc`:

Syntax: `realloc`

```
pointer = realloc(pointer, AnzahlBytesNeu);
```

Dabei ist `pointer` die Adresse des Arrays, dessen Größe geändert werden soll und `AnzahlBytesNeu` die gewünschte Größe des neuen Arrays.

Wie wir gesehen haben kann sich die Adresse des Arrays (also sein Startpunkt im Speicher) bei Größenänderung verschieben. Daher müssen wir auch hier das Ergebnis speichern; wir aktualisieren also den Wert der Variablen `pointer`. Natürlich kann es auch geschehen, dass die Größenänderung fehlschlägt (etwa, wenn ein zu großer Speicherbereich angefragt wird). In diesem Fall ist der Rückgabewert von `realloc` gleich 0. Es gehört zum guten Stil, auf solche Fehler mit `if` abzufangen.

Betrachten wir folgendes Beispiel aus der CPP-Referenz:

Beispiel: `realloc`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5      int *pa = malloc(10 * sizeof(*pa)); // allocate an array of 10 int
6      if (pa) {
7          printf("%ld bytes allocated. Storing ints: ", 10 * sizeof(int));
8          for(int n = 0; n < 10; ++n) {printf("%d ", pa[n] = n);}
9      }
10
11     // reallocate array to a larger size
12     int *pb = realloc(pa, 1000000 * sizeof(*pb));
13     if (pb) {
14         printf("\n%ld bytes allocated, first 10 ints are: ",
15             1000000 * sizeof(int));
16     };
17     for(int n = 0; n < 10; ++n) {printf("%d ", pb[n]);} // show the array
18     free(pb);
19 } else { // if realloc failed, the original pointer needs to be freed
20     free(pa);
21 }
22 }
```

(Mit leichten Abwandlungen übernommen von <https://en.cppreference.com/w/c/memory/realloc>)

Ausführungsbeispiel: `realloc`

```
40 bytes allocated. Storing ints: 0 1 2 3 4 5 6 7 8 9
4000000 bytes allocated, first 10 ints are: 0 1 2 3 4 5 6 7 8 9 0
```

In Zeile 5 wird versucht, mit `malloc` Speicher für ein Array zu reservieren, das 10 `int`-Werte fassen kann. Beachten Sie hier auch die Verwendung des `sizeof`-Operators: Abgefragt wird hier die Speichergröße von `*pa`. Die Variable `pa` ist ein Pointer auf `int`; damit ist die dereferenzierte Variable `*pa` selbst ein Ausdruck vom Typ `int`.

Nur wenn diese Allokierung erfolgreich war, wird das Array mit Werten befüllt und diese Werte zugleich auch auf dem Bildschirm ausgegeben. Beachten Sie hierbei die `printf`-Anweisung in Zeile 8: Ausgegeben wird der Ausdruck `pa[n] = n`. Dieser Ausdruck ist zunächst eine *Wertzuweisung* – in `pa[n]` wird also der Wert `n` gespeichert. Zugleich ist das „Ergebnis“ dieser Wertzuweisung der zugewiesene Wert selbst – `printf` gibt also den Wert von `n` aus.

In Zeile 12 versuchen wir nun, das Array so zu vergrößern, dass es 1 000 000 `int`-Werte halten kann. Wenn dies gelingt, wird die Variable `pb` einen von null verschiedenen Wert haben; der Pointer `pa` dagegen ist *möglicherweise nicht mehr gültig*!

Bei Erfolg dieser Vergrößerung können wir nun über `pb` auf die Werte des Arrays zugreifen, und finden die Elemente, die wir zuvor über `pa` gespeichert hatten. Obwohl im Programm zwei verschiedene Pointer-Variablen benutzt werden, existiert nur ein einziges Array.

Wie üblich müssen *dynamische Arrays* mit `free` freigegeben werden. War die Vergrößerung mit `realloc` erfolgreich, so muss dies über den Pointer `pb` geschehen, da `pa` möglicherweise „ins Leere“ zeigt. Könnte dagegen das Array nicht so weit vergrößert werden, so existiert weiterhin das Array mit der Adresse `pa`. Daher schreiben wir für diesen Fall in Zeile 20 `free(pa)`.

8.2.3. Heap und Stack

In Abschnitt 8.1 haben wir *automatische Arrays* kennen gelernt. Später in Abschnitt 8.2 haben wir *dynamische Arrays* eingeführt. Während der Zugriff auf die Array-Elemente für beide Arten gleich ist, bestehen doch Unterschiede. *Automatische Arrays* sind in ihrer Größe fest und können nach dem Anlegen nicht mehr verändert werden. Dafür müssen wir uns nicht um die Freigabe des Speichers nach Benutzung kümmern. Für *dynamische Arrays* gilt in dieser Hinsicht das Gegenteil.

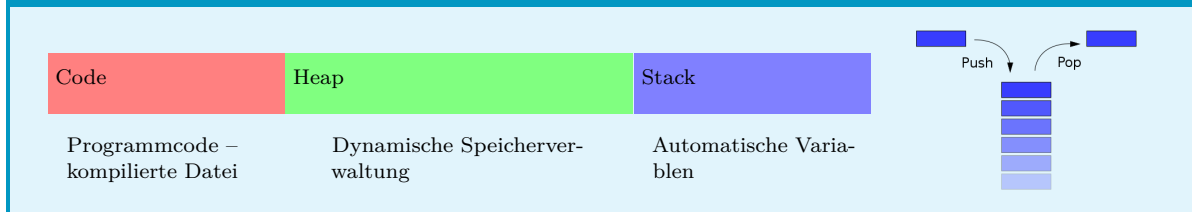
Um grob zu verstehen, warum das so ist, müssen Sie wissen, dass der Arbeitsspeicher in mehrere Segmente geteilt wird.

Ein erstes Segment ist der Code-Bereich. Hier liegt der Maschinencode. Dieser wird vom Prozessor ausgeführt.

Hieran schließt sich der *Heap* (englisch: Haufen) an. Es ist ein Bereich, auf den mehr oder minder „wahllos“ zugegriffen werden kann: beliebige Schnitte des Heaps können reserviert und wieder freigegeben werden, ohne eine bestimmte Struktur oder Reihenfolge einhalten zu müssen. Auf dem Heap „leben“ die dynamischen Arrays.

Der *Stack* dagegen ist eine strikt geordnete Speicherstruktur: Hier liegen alle Werte dicht an dicht, wie Schichten eines geordneten Stapels. Dies ist nur möglich, wenn die einzelnen Schichten nicht ihre Größe ändern (und damit Lücken hinterlassen oder andere Schichten zum nachrücken zwingen). Neue Elemente können nur „oben auf den Stack gelegt“ werden. Zum Freigeben des Speichers muss in umgekehrter Reihenfolge vorgegangen werden, der Stack wird „von oben nach unten“ freigegeben. Auf dem Stack „leben“ die automatischen Variablen, d. h. Arrays wie in Abschnitt 8.1 gezeigt, aber auch die „normalen Variablen“, die nur einen einzelnen Wert halten können.

Visualisierung: Heap und Stack



Der Stack hat zwar eine regelmäßige Gestalt, auf seine komplexe Verwaltung wird hier aber nicht näher eingegangen. Wir müssen lediglich die Variablen deklarieren, jedoch nicht explizit Speicherplatz allozieren oder freigeben. Den Rest übernehmen der Compiler und ggfs. das Betriebssystem.

Keine dynamischen Operationen an automatischen Variablen

Operationen der dynamischen Speicherverwaltung (insbesondere die Befehle `realloc` und `free`) sind auf dem Stack nicht erlaubt. Daher darf *niemals* die Größe eines automatischen Arrays mit `realloc` geändert werden und auch *niemals* sein Speicher mit `free` freigegeben werden. Der Versuch, dies zu tun führt sofort zum Programmabsturz.

8.2.4. C++

Ausflug: C++

Wie üblich stehen auch in C++ alle soeben gezeigten Mittel zur Verfügung. Automatische Arrays werden ebenso unterstützt wie in C. Für *dynamische Arrays* dagegen sollten nicht mehr `malloc`, `calloc` und `free` verwendet werden, sondern die Operatoren `new` und `delete`. Die Syntax lautet:

Syntax: `new` und `delete`

```
pointer = new datatype [AnzahlFelder];  
...  
delete pointer;
```

Auf die Unterschiede kann an dieser Stelle leider nicht im Detail eingegangen werden. Stattdessen sei hier vermerkt, dass die C++ Standard Library eine Vielzahl von Mitteln zur Verfügung stellt, die den *direkten Umgang* mit Pointern unnötig machen. Diese Mittel werden in einem eigenen Kurs vorgestellt und sind den C-Arrays i. d. R. klar vorzuziehen.

8.2.5. Mehrdimensionale dynamische Arrays

In Abschnitt 8.1.4 haben wir bereits gesehen, wie mehrdimensionale Objekte wie Tabellen im Speicher organisiert werden können. An dieser Stelle möchte ich zwei Techniken besprechen, wie solche Objekte auch mit dynamischen Arrays verwaltet werden können.

Geplättete Listen

Der sicher einfachste Weg ist es, eine Tabelle (oder sein höherdimensionales Objekt) zu „plättchen“, also die einzelnen Listen, aus denen es besteht, hintereinander im Speicher anzuordnen. In diesem Fall muss

zwar für jeden Zugriff aus den „Koordinaten“ (also z.B. Spalten- und Zeilennummern der Tabelle) ein Index berechnet werden. Der sonstige *Overhead* (Verwaltungsaufwand) hält sich jedoch in Grenzen.

Folgendes Beispiel erzeugt eine Tabelle, deren Maße vom User bestimmt werden, und befüllt die Zellen mit ihren Indizes. Anschließend wird die Tabelle auf dem Bildschirm ausgegeben:

Beispiel: Tabelle mit dynamischem Array

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5      int width = -1, height = -1, i, r, c;
6
7      // Usereingaben
8      printf("Bitte geben Sie die Breite der Tabelle ein:\n");
9      scanf("%d", &width);
10
11     printf("Bitte geben Sie die Höhe der Tabelle ein:\n");
12     scanf("%d", &height);
13
14     if ((height <= 0) || (width <= 0)) {
15         printf("Ungültige Eingabe -- Programm wird beendet.\n");
16         exit(-1);
17     }
18
19     // Speicher für Tabelle allozieren
20     int * table = malloc(width * height * sizeof(*table));
21     if (!table) {
22         printf("Allozierung fehlgeschlagen -- Programm wird beendet.\n");
23         exit(-1);
24     }
25
26     // Tabelleninhalt schreiben
27     for (i = 0; i < width * height; i++) {
28         table[i] = i;
29     }
30
31     // Tabelle auf dem Bildschirm ausgeben
32     for (r = 0; r < height; r++) {
33         for (c = 0; c < width; c++) {
34             printf("%3d\t", table[r * width + c]);
35         }
36         printf("\n");
37     }
38
39     // Speicher freigeben
40     free(table);
41 }
```

Ausgabebeispiel: Tabelle mit dynamischem Array

Bitte geben Sie die Breite der Tabelle ein:

5

Bitte geben Sie die Höhe der Tabelle ein:

3

| | | | | |
|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 |

Das Array `table`, in dem wir unsere Tabelle geplättet ablegen, kann mit `realloc` vergrößert und verkleinert werden. Soll die Struktur der Tabelle erhalten bleiben, können wir auf dieses Mittel allerdings nicht zurückgreifen.

Ausgehend vom obigen Code (*Tabelle mit dynamischem Array*) ändern wir das Ende des Codes ab und versuchen, ab Zeile 39 die Tabelle um eine Spalte zu verbreitern:

Beispiel: Tabelle vergrößern mit dynamischem Array – (fehlerhafter Code)

```
39  // Tabelle verbreitern
40  width++;
41  int * newTable = realloc(table, width * height * sizeof(*table));
42  if (!newTable) {
43      printf("Allozierung fehlgeschlagen -- Programm wird beendet.\n");
44      free(table);
45      exit(-1);
46  } else {
47      table = newTable;
48  }
49
50  // Tabelle auf dem Bildschirm ausgeben
51  printf("\nVerbreiterte Tabelle:\n");
52  for (r = 0; r < height; r++) {
53      for (c = 0; c < width; c++) {
54          printf("%3d\t", table[r * width + c]);
55      }
56      printf("\n");
57  }
58
59  // Speicher freigeben
60  free(table);
61 }
```

Die Ausgabe lautet:

Ausgabebeispiel: Tabelle vergrößern mit dynamischem Array – (fehlerhafter Code)

Bitte geben Sie die Breite der Tabelle ein:

5

Bitte geben Sie die Höhe der Tabelle ein:

3

| | | | | |
|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 |

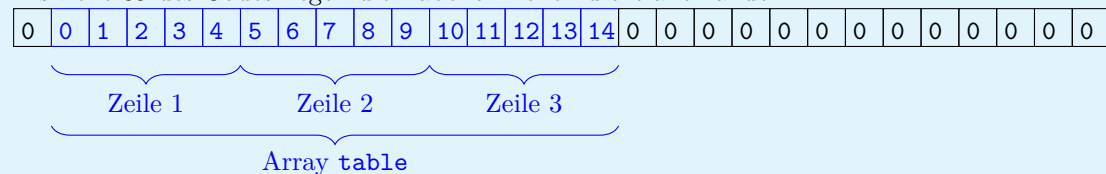
Verbreiterte Tabelle:

| | | | | | |
|----|----|----|---|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 0 | 0 | 0 |

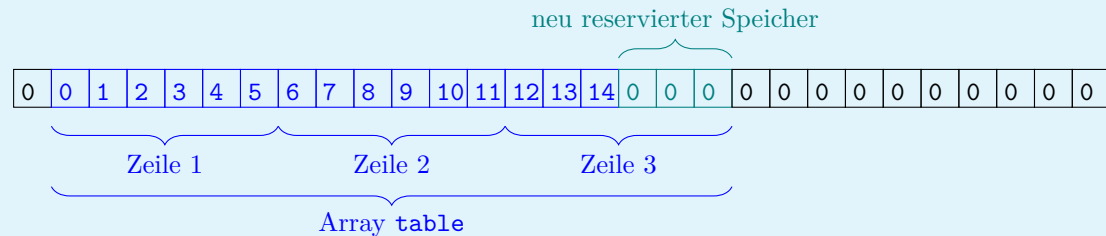
Um dieses Verhalten zu verstehen, müssen wir uns das Speicherbild ansehen:

Visualisierung: Abbild der Tabelle im Speicher

Bis Zeile 39 des Codes liegen die Tabellen-Zeilen dicht aneinander:



Mit `realloc` erweitern wir unseren Speicher um weitere 3 Elemente, um so Platz für eine zusätzliche Spalte zu haben. Wie in Abschnitt 8.2.2 gezeigt, wird dieser Platz *am Ende des Arrays* angehängt. Wegen `width++` in Code-Zeile 48 lesen wir ab jetzt aber Ketten von 6 Speicherstellen als eine Zeile, und erhalten eine zersetzte Tabelle:



Wenn dagegen nur eine Zeile an die Tabelle angehängt werden soll, funktioniert dieses Beispiel mit `realloc` genau wie erwartet, da neue Zeilen immer *am Ende* der „geplätteten Liste“ angehängt werden.

Um nun auch Spalten anhängen zu können, müssen wir manuell ein neues Array erstellen und alle Tabellen-Einträge an die richtigen Stellen kopieren. Wir gehen wieder vom Beispiel *Tabelle mit dynamischem Array* aus und ergänzen ab Zeile 39:

Beispiel: Tabelle vergrößern mit dynamischem Array – (korrekter Code)

```
39 // neue Tabelle anlegen
40 int * newTable = calloc((width + 1) * height, sizeof(*table));
41 if (!newTable) {
42     printf("Allozierung fehlgeschlagen -- Programm wird beendet.\n");
43     exit(-1);
44 }
45
46 // Tabelleninhalt kopieren
47 for (r = 0; r < height; r++) {
48     for (c = 0; c < width; c++) {
49         newTable[r * (width + 1) + c] = table[r * width + c];
50     }
51 }
52
53 // Speicher der alten Tabelle freigeben, Variablen aktualisieren
54 free(table);
55 table = newTable;
56 width++;
57
58 // Tabelle auf dem Bildschirm ausgeben
59 printf("\nVerbreiterte Tabelle:\n");
60 for (r = 0; r < height; r++) {
61     for (c = 0; c < width; c++) {
62         printf("%3d\t", table[r * width + c]);
63     }
64     printf("\n");
65 }
66
67 // Speicher freigeben
68 free(table);
69 }
```

Ausgabebeispiel: Tabelle vergrößern mit dynamischem Array – (korrekter Code)

Bitte geben Sie die Breite der Tabelle ein:

5

Bitte geben Sie die Höhe der Tabelle ein:

3

| | | | | |
|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 |

Verbreiterte Tabelle:

| | | | | | |
|----|----|----|----|----|---|
| 0 | 1 | 2 | 3 | 4 | 0 |
| 5 | 6 | 7 | 8 | 9 | 0 |
| 10 | 11 | 12 | 13 | 14 | 0 |

Pointer zweiter und tieferer Ebene

Wir haben bereits festgestellt, dass sich eine Tabelle auch als *Liste von Listen* auffassen lässt. Listen werden mit Pointern verwaltet. Die Sprache C erlaubt auch, *Pointer auf Pointer* anzulegen. Diesem Gedanken folgend wollen wir eine alternative Darstellungsmöglichkeit von Tabellen diskutieren:

Beispiel: Tabelle Pointer zweiter Ebene

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5      int width = -1, height = -1, r, c;
6
7      // Usereingaben
8      printf("Bitte geben Sie die Breite der Tabelle ein:\n"); scanf("%d", &width);
9      printf("Bitte geben Sie die Höhe der Tabelle ein:\n");  scanf("%d", &height);
10
11     if ((height <= 0) || (width <= 0)) {
12         printf("Ungültige Eingabe -- Programm wird beendet.\n");
13         exit(-1);
14     }
15
16     // Speicher für Tabelle allozieren
17     int ** table = malloc(height * sizeof(*table));           // Liste der Zeilen
18     if (!table) {
19         printf("Allozierung fehlgeschlagen -- Programm wird beendet.\n");
20         exit(-1);
21     }
22     for (r = 0; r < height; r++) {
23         table[r] = malloc( width * sizeof(*(table[r])) );    // Zeilen selbst
24     }
25
26     // Tabelleninhalt schreiben
27     for (r = 0; r < height; r++) {
28         for (c = 0; c < width; c++) {
29             table[r][c] = r * width + c;
30         }
31     }
32
33     // Tabelle auf dem Bildschirm ausgeben
34     for (r = 0; r < height; r++) {
35         for (c = 0; c < width; c++) {
36             printf("%3d\t", table[r][c]);
37         }
38         printf("\n");
39     }
40
41     // Speicher freigeben
42     for (r = 0; r < height; r++) {free(table[r]);}
43     free(table);
44 }
```

Die Ausgabe ist dieselbe wie im vorigen Beispiel (*Tabelle mit dynamischem Array*); „unter der Haube“ sieht es jedoch bedeutend anders aus.

Zeile 17 deklariert einen `int **`, also einen *Pointer auf einen `int *`*. Diese Variable `table` verweist also nicht auf den Speicherbereich, in dem die Tabellendaten selbst liegen, sondern auf Punkte im Speicher, die angeben, wo die einzelnen Zeilen der Tabelle zu finden sind.

Für jeden Eintrag in dieser Liste aus Pointern wird nun nochmal ein eigener Speicherbereich alloziert, der die tatsächlichen Tabellenwerte aufnehmen kann (Zeilen 22-24).

Mit diesem aufwändigeren Aufbau der Speicherstruktur können wir nun etwas komfortabler auf die Tabellenelemente zugreifen. In Zeile 29 wird `table[r][c]` beschrieben, wir geben also direkt den Zeilenindex `r` und den Spaltenindex `c` an, ohne einen geplätteten Index berechnen zu müssen. Halten Sie sich vor Augen: `table` ist ein `int **`. Damit liefert der Zugriff `table[r]` den Wert mit dem Index `r` aus der *Liste der Listen*, also einen Pointer auf die Zeile `r`. Auf diese Liste kann nun mit dem Spaltenindex `c` zugegriffen werden.

Insgesamt haben wir also `r + 1` Listen, die die Tabelle aufbauen (`r` Listen für die Inhalte der Zeilen selbst und eine Liste, die angibt, wo im Speicher diese Zeilen zu finden sind). Alle diese Listen müssen natürlich auch wieder freigegeben werden. Hierzu dienen die Zeilen 42 und 43. Wichtig ist, dies in *umgekehrter Reihenfolge des Anlegens* zu tun. Nach einem `free` „existiert“ ein Array nicht mehr und ein Zugriff darauf darf eigentlich nicht mehr stattfinden. Wenn wir also die Zeilen 42 und 43 vertauschen, würden in der `for`-Schleife verbotene Zugriffe auf `table` stattfinden und möglicherweise den Programmabsturz hervorrufen.

Wollen wir diese Tabelle vergrößern, so gilt es, zwei Fälle zu unterscheiden: Eine Änderung der Zeilen-Zahl oder der Spalten-Zahl. Wenn wir Zeilen hinzufügen, reicht es, lediglich `table` mit `realloc` zu vergrößern und mit `malloc` Platz für diese neuen Zeilen zu allozieren. Für zusätzliche Spalten müssen wir jede Liste `table[r]` mit `realloc` vergrößern. Zum Verkleinern solcher Strukturen müssen Sie auch darauf achten, dass wegfallende Zeilen mit `free(table[r])` freigegeben werden müssen.

Geplättete Listen bevorzugen

Wie Sie sehen ist die Verwaltung von Listen mit Pointern zweiter oder höherer Ebene komplexer und bietet mehr Fehlerquellen. Im Allgemeinen sind daher „geplättete Listen“ zu bevorzugen. Relevant kann diese Technik aber dort werden, wo sie keine vollständige, „rechteckige“ Tabelle beschreiben, sondern wo jede Zeile eine andere Länge hat. Für diesen Fall müssen Sie natürlich noch ein zusätzliches Array anlegen, in dem Sie die Längen der Zeilen speichern.

Reihenfolge der Dimensionen frei wählbar

In den obigen Beispielen folgte ich der Konvention „Zeilen zuerst, Spalten später“. Es gibt aber keine zwingenden Gründe hierfür; Sie können die obigen Beispiele leicht so anpassen, dass die Reihenfolge von Zeilen und Spalten vertauscht werden. Die Berechnung der geplätteten Indizes kann gleich bleiben, oder nach der Formel

$$\text{Index} = c * \text{height} + r$$

stattfinden. Dies entspricht einer Spaltenweisen Anordnung im Speicher.

Für welche Anordnung Sie sich auch entscheiden: Behalten Sie eine einmal gewählte Konvention bei, da Sie sonst durch die Verwechslung von Spalten und Zeilen viele Fehler machen werden. Die hier gezeigten Konventionen orientieren sich an den Gewohnheiten der Mehrheit der ProgrammiererInnen. Für die Arbeit in Teams ist diese Reihenfolge also zu bevorzugen.

8.3. C-Strings

Bisher haben wir fast ausschließlich mit Zahlenwerten gearbeitet. Wie wir in Abbildung 1.1 gesehen haben, sind Buchstaben aber nur „Interpretationsweisen“ von Ganzzahlen. Texte – Zeichenketten, oder englisch: *Strings* – lassen sich also mit Ganzzahl-Arrays behandeln. Um diese (häufig auftretende) Aufgabe zu erleichtern bringt die Sprache C einige Mittel für den Umgang mit Strings mit.

8.3.1. Spezielle Syntax-Elemente

Für einfache Aufgaben steht uns ein Vorrat aus 256 verschiedenen Zeichen zur Verfügung. Dies ist genau der Wertebereich des Datentyps `char` (von englisch: *character*, Buchstabe oder Zeichen). Wir verwenden also `char`-Arrays zur Arbeit mit Strings.

Es ist zwar möglich, jeden Buchstaben in der ASCII-Tabelle nachzuschlagen und manuell den zugehörigen Zahlenwert zuzuweisen; jedoch wäre dies reichlich aufwändig und fehleranfällig. Stattdessen können wir auch 'einfache Anführungszeichen' benutzen, um „den Zeichencode für das eingeschlossene Zeichen“ auszudrücken.

Beispiel: ASCII-Codes

```
1  #include <stdio.h>
2
3  int main () {
4      printf("Der ASCII-Code von 'a' ist %d.\n", 'a');
5  }
```

Der Format-String erwartet eine Dezimalzahl (%d). Um diesen Platzhalter auszufüllen wird der Ausdruck 'a' angeboten. Statt also eine (nicht existente) Variable `a` auszugeben wird der ASCII-Code des Zeichens `a` gelesen. Entsprechend lautet die Ausgabe:

Ausführungsbeispiel: ASCII-Codes

Der ASCII-Code von 'a' ist 97.

Regelmäßigkeit von ASCII-Codes

Die Anordnung der Zeichen im ASCII-Vorrat ist nicht zufällig. Ein nützliches Ergebnis dieser Anordnung ist, dass sich die Bitmuster von Groß- und Kleinbuchstaben in genau einem Bit unterscheiden. Es ist also möglich, „auf den ersten Blick“ festzustellen, ob eine bestimmte Folge von 1en und 0en einen großen oder einen kleinen Buchstaben beschreibt.

Da Buchstaben nur Interpretationsweisen von Zahlen sind, kann mit diesen auch gerechnet werden wie mit normalen Ganzzahlen. Insbesondere beschreibt der Ausdruck

`'a' - 'A'`

damit auch den Abstand zwischen Groß- und Kleinbuchstaben. Der Wert dieses Ausdrucks kann zu jedem Großbuchstaben addiert werden um den entsprechenden Kleinbuchstaben zu finden; umgekehrt wandelt die Subtraktion einen Kleinbuchstaben in einen Großbuchstaben um.

Mit diesem Mittel können wir bereits ein `char`-Array anlegen und einen Text darin ablegen:

Beispiel: Wertzuweisung Strings (1)

```
1  #include <stdio.h>
2
3  int main () {
4      char string[] = {
5          'D', 'u', 'n', 'e', ' ', 'b', 'y', ' ',
6          'F', 'r', 'a', 'n', 'k', ' ', 'H', 'e', 'r', 'b', 'e', 'r', 't', 0};
7  }
```

Damit legen wir ein automatisches Array namens **string** an, und speichern darin den Text **Dune by Frank Herbert**. Die Größe wird durch die Initializer-Syntax automatisch ermittelt und die einfachen Anführungszeichen sorgen dafür, dass die ASCII-Codes richtig zugewiesen werden.

Beachten Sie auch das letzte Zeichen dieses Beispiels: Das Zeichen 0 (*Null-Char*) wird im Kontext von Zeichenketten benutzt um das Ende eines Strings zu markieren. Der Vorteil hiervon ist, dass man nicht im Vorhinein die Länge eines Strings kennen muss, um sinnvoll mit diesem arbeiten zu können. Bei der Ausgabe auf dem Bildschirm werden beispielsweise so lange Zeichen gedruckt, bis man auf dieses Null-Char stößt. Statt die Zahl 0 zu schreiben kann auch die *Escape-Sequenz* `\0` benutzt werden.

Wenn wir nicht nur mit einzelnen Schriftzeichen arbeiten, sondern ganze *Zeichenketten* bearbeiten, dann ist auch die oben gezeigte Syntax noch unhandlich. Zu diesem Zweck existiert die Möglichkeit, ganze Zeichenketten in "doppelte Anführungszeichen" zu setzen. Gleichwertig zu obigem Code (*Wertzuweisung Strings (1)*) ist also auch:

Beispiel: Wertzuweisung Strings (2)

```
1  #include <stdio.h>
2
3  int main () {
4      char string[] = "Dune by Frank Herbert";
5  }
```

Sie sehen: Nicht nur ist dieser Code bedeutend kürzer; Sie können auch auf die Nennung des Null-Chars verzichten. Solche Zeichenketten in "doppelten Anführungszeichen" nennen wir *String-Literals*.

Zur Ausgabe von Strings steht uns ein eigenes Formatierungszeichen zur Verfügung. Im Formatstring können wir `%s` setzen, um eine Zeichenkette auszugeben. In der Parameterliste wird hier der *Pointer auf das char-Array* übergeben:

Beispiel: Formattierte Ausgabe von Strings (1)

```
1  #include <stdio.h>
2
3  int main () {
4      char string[] = "Dune by Frank Herbert";
5      printf("My favourite book is %s.\n", string);
6  }
```

Ausführungsbeispiel: Formattierte Ausgabe von Strings (1)

```
My favourite book is Dune by Frank Herbert.
```


Ähnlich wie auch bei den Zahlentypen kann in einem `printf`-Formatstring angegeben werden, wie viel Platz für die Ausgabe zur Verfügung gestellt werden soll. Wie schon zuvor erzeugt eine positive Zahl dabei eine rechtsbündige Ausgabe; negative Zahlen halten den Text linksbündig.

Beispiel: Formattierte Ausgabe von Strings (2)

```
1  #include <stdio.h>
2
3  int main () {
4      printf("|%s|\n" , "Dune");
5      printf("|%10s|\n" , "Dune");
6      printf("|%-10s|\n" , "Dune");
7  }
```

Ausführungsbeispiel: Formattierte Ausgabe von Strings (2)

```
|Dune|
|      Dune|
|Dune      |
```

8.3.2. User-Eingaben

Wir können `scanf` mit dem Formatstring-Platzhalter `%s` benutzen, um den User direkt zur Texteingabe aufzufordern. Dafür muss bereits vor der Eingabe genug Speicherplatz reserviert sein, um die gesamte Usereingabe aufzunehmen. Wie üblich werden Zeilenumbrüche, Tabulatoren und Leerzeichen als Trennzeichen interpretiert:

Beispiel: Strings mit scanf einlesen (1)

```
1  #include <stdio.h>
2
3  int main () {
4      char userinput[1024];
5
6      printf("Bitte nennen Sie eine gute Band:\n");
7      scanf("%s", userinput);
8
9      printf("Ihre Wahl war: %s\n", userinput);
10 }
```

Ausführungsbeispiel: Strings mit scanf einlesen (1)

```
Bitte nennen Sie eine gute Band:
Wir Sind Helden
Ihre Wahl war: Wir
```

Wenn (wie im Beispiel oben) diese Vorgabe von Trennzeichen Probleme macht, so kann auch die „Set-Schreibweise“ verwenden: zwischen `%[` und `]` werden die Zeichen eingeschlossen, die zum String gehören dürfen; alle anderen Zeichen werden als Trennzeichen aufgefasst. Zeichen-Bereiche können mit Bindestrich (`-`) umschrieben werden. So erlaubt etwa

```
scanf("%[A-Za-z ]", userinput);
```

die Eingabe von Klein- und Großbuchstaben sowie Leerzeichen.

Meist noch kompakter ist die „Ausschlussset-Schreibweise“: Ist das erste Zeichen in den [eckigen Klammern] ein Zirkumflex (^), so werden alle aufgelisteten Zeichen als Trennzeichen interpretiert; alle sonstigen Eingaben werden zum String gerechnet:

Beispiel: Strings mit scanf einlesen (2)

```
1  #include <stdio.h>
2
3  int main () {
4      char userinput[1024];
5
6      printf("Bitte nennen Sie eine gute Band:\n");
7      scanf("%[^\n]", userinput);
8
9      printf("Ihre Wahl war: %s\n", userinput);
10 }
```

Ausführungsbeispiel: Strings mit scanf einlesen (2)

```
Bitte nennen Sie eine gute Band:
Wir Sind Helden
Ihre Wahl war: Wir Sind Helden
```

Zu lange Usereingaben

Im obigen Beispiel haben wir ein Kilobyte an Arbeitsspeicher (1024 Bytes) zur Verfügung gestellt, um einen wenige Bytes umfassenden String aufzunehmen. Dies mag verschwenderisch erscheinen, ist aber tatsächlich gute Praxis. Im Allgemeinen ist nicht bekannt, was der User eingeben wird, und wir hatten zuvor schon gesehen, dass Eingaben, die den reservierten Speicherbereich übersteigen, andere Speicherzellen überschreiben können (*bleeding*). Um dies zu vermeiden werden in Situationen, die eine String-Eingabe erfordern große Puffer bereitgestellt.

Usereingaben begrenzen

Da auch ein sehr großer Puffer für Usereingaben „vollaufen“ kann, bietet `scanf` die Möglichkeit, die Usereingabe auf eine bestimmte Zeichenzahl zu begrenzen. Dazu wird hinter das %-Zeichen eine Zahl gesetzt, die die maximale Eingabelänge beschreibt. Dies funktioniert sowohl mit %s, mit der Set-Schreibweise und mit der Ausschlussset-Schreibweise. In den obigen Beispielen sollte man also

```
scanf("%1023s", userinput);    bzw.    scanf("%1023[^\n]", userinput);
```

setzen.

Es ist hierbei kein Versehen, dass die Eingabe auf 1023 Zeichen begrenzt wird, obwohl `userinput` ein Array mit bis zu 1024 `char`-Elementen ist. Das letzte Element eines Strings ist immer der Null-char, der bei der Begrenzung von `scanf` nicht mitgerechnet wird.

8.3.3. Formatierte Strings erzeugen: `sprintf` und `snprintf`

Ebenso wie mit `printf` formatierter Text auf dem Bildschirm ausgegeben werden kann, lassen sich Strings aus Variablen-Werten generieren. Dazu dienen die Befehle `sprintf` und `snprintf`, die im Header `<stdio.h>` deklariert werden.

`sprintf` nimmt dieselben Parameter an wie `printf`, mit dem Unterschied, dass dem Formatstring noch ein *Puffer* vorangestellt wird, d. h. ein Pointer auf einen Speicherbereich, in den der String geschrieben werden soll:

Syntax: `sprintf`

```
sprintf(buffer, formatstring, ...);
```

Beispiel: Einen String mit `sprintf` generieren

```
1  #include <stdio.h>
2
3  int main () {
4      char buffer[1024];
5
6      sprintf(buffer, "normaler Text mit Formatcodes: %d", 17);
7
8      printf("Generierter String: \"%s\"\n", buffer);
9  }
```

Ausführungsbeispiel: Einen String mit `sprintf` generieren

```
Generierter String: "normaler Text mit Formatcodes: 17"
```

`buffer` kann ein beliebiger `char *` sein. Es kann also sowohl ein automatisches Array (wie im obigen Beispiel) beschrieben werden als auch ein dynamisches Array (von `malloc`). In jeden Fall aber muss sichergestellt sein, dass der Speicherbereich tatsächlich reserviert ist, und nicht über die Grenzen der Reservierung hinaus geschrieben wird. Beachten Sie hierbei auch, dass ein abschließender Null-char mit geschrieben wird.

Der Befehl `snprintf` funktioniert ebenso wie `sprintf`, verlangt aber noch einen zusätzlichen Parameter vom Typ `unsigned long`, der angibt, wie viele Zeichen maximal geschrieben werden dürfen (inclusive abschließendem Null-char). Dies stellt einen zusätzlichen Sicherheitsmechanismus gegen versehentliches Schreiben über den reservierten Bereich hinaus dar und ist grundsätzlich zu bevorzugen. Ist der zu schreibende String länger als dieser `unsigned long`-Wert, so wird der Text abgeschnitten. Auch `snprintf` schließt den Puffertext mit einem Null-char ab, selbst wenn der eigentliche Text abgeschnitten wurde.

Syntax: `snprintf`

```
snprintf(buffer, length, formatstring, ...);
```

Beispiel: Einen String mit `snprintf` generieren

```
1  #include <stdio.h>
2
3  int main () {
4      char buffer[10];
5
6      snprintf(buffer, 9, "normaler Text mit Formatcodes: %d", 17);
7
8      printf("Generierter String: \"%s\"\n", buffer);
9  }
```

Ausführungsbeispiel: Einen String mit `sprintf` generieren

Generierter String: "normaler"

Beide Befehle geben die Zahl der geschriebenen Zeichen (ohne Null-char) zurück. Zusätzlich kann `snprintf` genutzt werden, um den Speicherbedarf einer Ausgabe zu ermitteln, ohne Werte in den Speicher zu schreiben: Wird als `buffer` und als `length` jeweils der Wert 0 übergeben, so ermittelt `snprintf` lediglich die Länge eines Textes:

Beispiel: String-Länge mit `snprintf` ermitteln

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main () {
5      int number, length;
6      char * buffer = NULL;
7
8      printf("Bitte eine Ganzzahl eingeben:\n");
9      scanf("%d", &number);
10
11     // Bedarf ermitteln
12     length = snprintf(0, 0, "%d", number);
13
14     buffer = malloc(length + 1);
15     if (!buffer) {
16         printf("Fehler: Speicher konnte nicht reserviert werden");
17         exit(-1);
18     }
19
20     // tatsächlich schreiben
21     snprintf(buffer, length + 1, "%d", number);
22
23     printf("Eingabe als String: %s\n", buffer);
24
25     free(buffer);
26 }
```

8.3.4. Nützliche Funktionen aus der String-Library

Die folgenden Funktionen sind nützlich beim Umgang mit Arrays und Zeichenketten. Zu Details konsultieren Sie bitte die Befehlsreferenz unter <https://en.cppreference.com/w/c/>. Insbesondere sei auch auf die Übersicht unter <https://en.cppreference.com/w/c/string/byte> hingewiesen.

| Funktion | Header | Effekt |
|---|-------------------------------|--|
| <code>strlen</code> | <code><string.h></code> | Länge eines Strings (ohne Null-char) ermitteln |
| <code>strcmp</code> | <code><string.h></code> | Den <i>Inhalt</i> zweier Strings vergleichen |
| <code>strcat</code> | <code><string.h></code> | Zwei Strings verketteten („String-Addition“) |
| <code>strcpy</code> | <code><string.h></code> | Den <i>Inhalt</i> eines Strings kopieren |
| <code>strchr</code> | <code><string.h></code> | Nach einem Zeichen in einem String suchen (Suchrichtung links nach rechts) |
| <code>strrchr</code> | <code><string.h></code> | Nach einem Zeichen in einem String suchen (Suchrichtung rechts nach links) |
| <code>memXXX</code> | <code><string.h></code> | (Varianten von <code>strcmp</code> , <code>strcpy</code> , <code>strchr</code> und <code>strrchr</code> die auch mit Null-chars im Array funktionieren) |
| <code>tolower</code> , <code>toupper</code> | <code><string.h></code> | Einen String in Klein- oder Großbuchstaben umwandeln |
| <code>isXXX</code> | <code><ctype.h></code> | (Verschiedene Funktionen, die Wahrheitswert zurückgeben, z. B. für <i>ist Zeichen ein alphanumerisches Zeichen?</i>) |
| <code>atoXXX</code> | <code><stdlib.h></code> | (Verschiedene Funktionen, die einen String als Zahl interpretieren und einen <code>int</code> , <code>double</code> , oder ähnlichen Datentyp zurück geben.) |

Tabelle 8.1.: Gängige Funktionen der string-library

8.4. Speicherbedarf von Datenstrukturen ermitteln: `sizeof`

Wir hatten den Operator `sizeof` im Abschnitt 8.2 kennen gelernt. Er dient dazu, den Speicherbedarf eines Objekts in Byte zu bestimmen. Als *Argument* können sowohl Datentypen als auch Variablen und Ausdrücke übergeben werden. An dieser Stelle sollen einige Sonderfälle besprochen werden, die sich im Umgang mit Arrays ergeben.

Wie bekannt werden Arrays über Pointer verwaltet. Auf gängigen Systemen sind Pointer 64-bit Werte; es wäre also zu erwarten, dass `sizeof(pointer)` immer den Wert 8 ergibt. Während dies bei Pointern auf *dynamische Arrays* tatsächlich so ist, gibt `sizeof` bei *automatischen Arrays* die *Gesamtgröße* des Feldes aus. Betrachten Sie folgendes Beispiel:

Beispiel: sizeof mit automatischen und dynamischen Arrays

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main () {
5      char  autoString[] = "ABCDEFGHJIJ";
6      char * dynString   = malloc(200);
7
8      for (int i=0; i<10; i++) {
9          dynString[i] = 'A' + i;
10     }
11     dynString[10] = 0;
12
13     int  autoInts[] = {1, 2, 3, 4, 5};
14     int * dynInts   = malloc(200);
15
16     printf("automatischer String\n");
17     printf("%s\n" ,      autoString );
18     printf("%ld\n", sizeof(autoString));
19
20     printf("dynamischer String\n");
21     printf("%s\n" ,      dynString );
22     printf("%ld\n", sizeof(dynString));
23
24     printf("\n");
25
26     printf("einzelne int-Variable : %ld\n", sizeof(*dynInts))
27     printf("automatisches int-Array: %ld\n", sizeof(autoInts));
28     printf("dynamisches  int-Array: %ld\n", sizeof( dynInts));
29
30
31     free(dynString);
32     free(dynInts);
33 }
```

Ausführungsbeispiel: sizeof mit automatischen und dynamischen Arrays

```
automatischer String
ABCDEFGHJIJ
11
dynamischer String
ABCDEFGHJIJ
8

einzelne int-Variable : 4
automatisches int-Array: 20
dynamisches  int-Array: 8
```

In den Zeilen 5 und 6 werden jeweils `char *`-Variablen deklariert (die Zeilen 8-11 dienen lediglich dazu, dem dynamischen Array `dynString` denselben Inhalt zu geben wie `autoString`). Analog dazu deklarieren wir in Zeile 13 und 14 zwei `int *`-Variablen.

Da die automatischen Variablen auf dem Stack abgelegt werden und der Verwaltung durch Compiler bzw. des Betriebssystems unterliegen, ist es möglich, ihren Speicherbedarf zur Runtime anzugeben. Entsprechend geben Zeile 18 und 27 die Werte 11 (Zeichenkette inklusive Null-char) bzw. 20 (fünf Werte mal 4 Bytes pro Wert) aus. Die Zeilen 22 und 28 dagegen fragen die Größe eines dynamischen Arrays ab. Diese liegen auf dem Heap und können zur Runtime nicht mehr analysiert werden. Entsprechend lautet beide Male die Ausgabe 8 (die Speicherbreite eines Pointers: ein 64-bit Wert). In Zeile 26 fragen wir die Größe eines *einzelnen Werts* ab. Dieser Wert ist vom Typ **int** und ist somit 4 Bytes breit.

Die Funktion **strlen** (deklariert im Header **string.h**) gibt die Zahl der Zeichen bis zum ersten Null-char aus (also die Länge des Strings) und funktioniert sowohl mit dynamischen als auch mit automatischen Arrays. Dieses Ergebnis ist aber nicht zwingend identisch mit dem Speicherbedarf, selbst, wenn wir mit automatischen Arrays arbeiten:

Beispiel: Unterschied zwischen **sizeof** und **strlen**

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main () {
5      char string[] = "AB\OCD";
6
7      printf("%s\n", string);
8      printf("%s\n", string + 3);
9
10     printf("sizeof: %ld\n", sizeof(string));
11     printf("strlen: %ld\n", strlen(string));
12 }
```

Ausführungsbeispiel: Unterschied zwischen **sizeof** und **strlen**

```
AB
CD
sizeof: 6
strlen: 2
```

Da hier **string** einen Null-char enthält (gegeben durch die Escape-Sequenz `\0`), bricht die Ausgabe nach dem B ab. Es wird immer noch automatisch ein Null-char an das Ende des Strings angehängt, so dass auch Zeile 8 eine „sinnvolle“ Ausgabe liefert.

So, wie die String-Ausgabe beim Null-char endet, erkennt auch **strlen** dieses Zeichen als Ende des Strings an. Daher ist das Ergebnis von Zeile 11 der Wert 2 – die Länge von **AB**. **sizeof** dagegen liefert die Größe der Speicherstruktur auf dem Stack zurück, unabhängig von seinem Inhalt. Bei Programmstart wurden hier 6 Bytes reserviert (vier für die Buchstaben A-D, eines für das explizit angegebene Null-char durch `\0` und eines für das automatische Null-char zum Abschluss des Strings.)

8.5. Debugging-Tool **valgrind**

Der Umgang mit Arrays (insbesondere mit dynamischen Arrays) bringt einige Schwierigkeiten mit sich. Niemals darf auf nicht reservierte Speicherbereiche zugegriffen werden. Außerdem muss jeder dynamisch reservierte Speicherblock wieder mit *free* freigegeben werden; danach darf kein Zugriff mehr erfolgen.

In Projekten mittlerer Größe können diese Regeln leicht verletzt werden. Die Folgen sind schwer abzuschätzen; sie können versteckt bleiben oder zum Programmabsturz führen.

Um leichter zu prüfen, ob ein eigenes Programm alle Regeln befolgt, existiert auf unixoiden Systemen³ das Kommandozeilen-Tool **valgrind**. Dieses Tool startet ein Programm und führt es regulär aus; dabei wird jede Maschinencode-Anweisung des Programms überprüft. Array-Zugriffe außerhalb des gültigen Bereichs oder vergessene **free**-Anweisungen werden in einem Report ausgegeben.

Beispiel: Im folgenden Code fehlt ein **free**. Außerdem soll mit **printf** ein Wert aus einem nicht mehr reservierten Speicherbereich ausgegeben werden:

Beispiel: Fehlerhafter Code mit dynamischer Speicherverwaltung

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main () {
5      int * x = malloc(20);
6      x[0] = 500;
7
8      printf("Ausgabe mit printf: %d\n", x[5]);
9  }
```

Dieser Code wird wie folgt kompiliert und dann *über das tool valgrind* gestartet:

Kompilieren und Starten über valgrind

```
gcc -std=c11 -Wall -Wpedantic -Wimplicit-fallthrough myProgram.c -lm -o myProgram
valgrind ./myProgram
```

Die Ausgabe sieht folgendermaßen aus:

Ausführungsbeispiel: Fehlerhafter Code mit dynamischer Speicherverwaltung über valgrind

```
==28330== Memcheck, a memory error detector
==28330== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==28330== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==28330== Command: ./myProgram
==28330==
==28330== Invalid read of size 4
==28330==    at 0x1086B2: main (in /home/blue-chameleon/Codes/myProgram)
==28330==   Address 0x522d054 is 0 bytes after a block of size 20 alloc'd
==28330==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-1.
==28330==   by 0x10869B: main (in /home/blue-chameleon/Codes/myProgram)
==28330==
Ausgabe mit printf: 0
==28330==
==28330== HEAP SUMMARY:
==28330==    in use at exit: 20 bytes in 1 blocks
==28330==   total heap usage: 2 allocs, 1 frees, 1,044 bytes allocated
==28330==
```

³d. h. auf Linux und Mac


```

==28330== LEAK SUMMARY:
==28330==     definitely lost: 20 bytes in 1 blocks
==28330==     indirectly lost: 0 bytes in 0 blocks
==28330==     possibly lost: 0 bytes in 0 blocks
==28330==     still reachable: 0 bytes in 0 blocks
==28330==     suppressed: 0 bytes in 0 blocks
==28330== Rerun with --leak-check=full to see details of leaked memory
==28330==
==28330== For counts of detected and suppressed errors, rerun with: -v
==28330== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)

```

Die tatsächliche Ausgabe unseres Codes wird also „zwischen den Report eingeschoben“. Wir finden die Information `Invalid read of size 4`, die uns darauf hinweist, dass Zeile 8 unseres Codes einen unerlaubten Lesezugriff anweist. Zwar geht aus dem Report von `valgrind` nicht hervor, welche Zeile unseres Codes fehlerhaft war, allerdings kann die Ursache bereits stark eingeschränkt werden. (Die Zeile `at 0x1086B2: main (in /home/blue-chameleon/Codes/myProgram)` sagt aus, dass der Zugriff in der Funktion `main` stattfindet. Bisher haben wir ausschließlich mit einer einzigen Funktion gearbeitet; in Kapitel 9 werden wir aber sehen, wie wir unser Programm strukturieren und eigene Funktionen schreiben. Diese Strukturierung wird auch im `valgrind`-Report erhalten bleiben.)

Die Zeile `in use at exit: 20 bytes in 1 blocks` weist darauf hin, dass einmal vergessen wurde, einen Speicherblock mit `free` freizugeben. Der fragliche Speicherblock ist 20 Bytes groß. Mit dieser Information kann man nun auf die Suche gehen und das fehlende `free` zum Code hinzufügen.

Korrekt geschriebener Code wie der folgende:

Beispiel: Korrekter Code mit dynamischer Speicherverwaltung

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main () {
5      int * x = malloc(20);
6
7      x[0] = 500;
8      printf("Ausgabe mit printf: %d\n", x[0]);
9
10     free(x);
11 }

```

erzeugt folgenden Report:

Ausführungsbeispiel: Korrekter Code mit dynamischer Speicherverwaltung über `valgrind`

```

==28459== Memcheck, a memory error detector
==28459== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==28459== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==28459== Command: ./myProgram
==28459==

```

```

Ausgabe mit printf: 500
==28459==
==28459==  HEAP SUMMARY:
==28459==      in use at exit: 0 bytes in 0 blocks
==28459==    total heap usage: 2 allocs, 2 frees, 1,044 bytes allocated
==28459==
==28459== All heap blocks were freed -- no leaks are possible
==28459==
==28459== For counts of detected and suppressed errors, rerun with: -v
==28459== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

Die letzte Zeile `ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)` sollte für jedes Projekt angestrebt werden.

Da jede Anweisung von `valgrind` überprüft werden muss, geschieht die Ausführung natürlich bedeutend langsamer.

8.6. Variable Length Arrays

Um dem Aufwand zu entkommen, Arrays manuell mit `malloc` anzulegen und wieder mit `free` freizugeben, verwenden EinsteigerInnen gerne Ausdrücke wie

```
int array[n];
```

wobei `n` eine beliebige `int`-Variable ist. Solche *Variable Length Arrays (VLAs)* werden seit dem Standard C99 unterstützt, sollten allerdings vermieden werden.

Wie alle Arrays, die auf diese Art deklariert werden, handelt es sich um *automatische* Arrays, die auf dem *Stack* angelegt werden. Das bedeutet, der Compiler versucht, die Speicheranordnung zu optimieren und verwaltet das Array selbst. Da zur *Kompilierzeit* in diesem Fall aber der Speicherbedarf nicht bekannt ist (`n` ist eine Variable, die erst zur *Laufzeit* einen Wert erhält), muss „auf Verdacht“ Speicherplatz reserviert werden. Außerdem fehlen so Informationen für die Warnmechanismen; d. h. bei der Arbeit mit VLAs erhalten wir weniger Hinweise auf Fehler im Code.

Ausflug: C++

Die Sprache C++ bietet viele Konzepte, die die direkte Arbeit mit Arrays unnötig machen; jedoch existiert dieses Konzept auch in C++, wie wir bereits gehört haben. VLAs wurden dort mit dem Standard C++14 (aus dem Jahr 2014) schließlich auch eingeführt; dort gilt aber umso mehr die Empfehlung, das Konstrukt zu vermeiden, da die Überwiegende Mehrheit der ProgrammierInnen noch immer den Standard C++11 verwenden und daher eigener Code als inkompatibel angesehen wird.

9. Strukturierte Programmierung

Before software can be reusable it first has to be usable.

Ralph Johnson

Viele Aufgaben wiederholen sich oder lassen sich verallgemeinern. Folgender Code berechnet beispielsweise den Wert von Eulers Zahl, also $\exp(1)$, kann aber auch dazu verwendet werden, andere Potenzen von e zu finden:

Beispiel: Berechnung von Eulers Zahl

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main () {
5      double x0 = 1.0;
6
7      int    accuracy = 20;
8      double x        = 1.0;
9      double d        = 1.0;
10     double e        = 1.0;
11
12     for (int i=1; i<accuracy; i++) {
13         x *= x0;
14         d *= i;
15         e += x / d;
16     }
17
18     printf("Manuell: %10.8lf\n", e);
19     printf("Mathlib: %10.8lf\n", exp(x0));
20 }
```

Ausführungsbeispiel: Berechnung von Eulers Zahl

```
Manuell: 2.71828183
Mathlib: 2.71828183
```

Zeile 18 zeigt sowohl, dass dieser Code das gewünschte Ergebnis liefert, aber auch, dass es möglich sein sollte, solchen Code kompakt zu einem eigenen Befehl (hier `exp`) zusammenzufassen. In diesem Kapitel werden wir sehen, wie wir solche *Funktionen* schreiben.

9.1. Scopes

Die Funktionen, die wir selbst definieren werden, sollen vom restlichen Programmcode unabhängig sein. Das bedeutet, dass Variablen, die wir für in unserer Funktion benutzen keine anderen Werte überschreiben sollen, die wir im restlichen Programm benutzen. Dies soll für alle Programme gelten, die wir je schreiben, gleich, welche Variablennamen darin vorkommen. Um dies zu ermöglichen, existiert in C das Konzept von *Scopes*:

Scopes sind Code-Abschnitte, innerhalb derer Variablen existieren, d. h. innerhalb derer die Variablen über ihren Namen benutzt werden können. Variablen, die innerhalb eines Scopes deklariert wurden, „existieren“ außerhalb dieses Scopes nicht. Scopes werden durch {geschweifte Klammern} eingegrenzt.

Beispiel: Scopes (1)

```
1  #include <stdio.h>
2
3  int main () {
4      int x = 1;
5
6      {
7          int y = 2;
8          printf("%d\n", y);
9      }
10
11     // printf("%d\n", y); -- Fehler: y nicht deklariert
12 }
```

Die Variable `y` existiert nur zwischen Zeile 7 (wo sie deklariert wird) und Zeile 9 (wo der Scope endet, innerhalb derer `y` „lebt“). Daher würde Zeile 11 einen Compiler-Fehler erzeugen, da auf eine nicht deklarierte Variable verwiesen wird.

Variablen sind „nach innen sichtbar“. Das bedeutet, dass die Variable `x` auch innerhalb des Scopes gelesen oder verändert werden kann, da sie außerhalb des Scopes deklariert wurde und nach innen „fortlebt“:

Beispiel: Scopes (2)

```
1  #include <stdio.h>
2
3  int main () {
4      int x = 1;
5
6      {
7          int y = 2;
8          printf("%d\n", x); // kein Problem -- Sichtbarkeit nach innen
9          printf("%d\n", y);
10     }
11
12     // printf("%d\n", y); -- Fehler: y nicht deklariert
13 }
```

Innerhalb von Scopes dürfen Namen auch neu vergeben werden. Das heißt, dass innerhalb des Scopes eine eigene Variable `x` deklariert werden darf. Die alte Variable `x` existiert weiterhin, ist aber unter diesem Namen nicht mehr ansprechbar. Stattdessen ist das neue `x` innerhalb des Scopes komplett unabhängig von dem `x` außerhalb, und existiert ebenso wie `y` nur innerhalb des Scopes:

Beispiel: Scopes (3)

```
1  #include <stdio.h>
2
3  int main () {
4      int x = 1;
5
6      {
7          int y = 2;
8          printf("%d\n", x);    // Sichtbarkeit nach innen -- 1 wird angezeigt
9          printf("%d\n", y);
10
11         double x = 5.0;
12         printf("%lf\n", x);   // neue Variable wird unter dem Namen x angesprochen
13     }
14
15     printf("%d\n", x);        // alte Variable wird angesprochen -- Ausgabe 1
16 }
```

Ausführungsbeispiel: Scopes (3)

```
1
2
5.000000
1
```

Scopes können (nahezu) beliebig tief ineinander verschachtelt werden. Scopes, die auf gleicher Ebene stehen „teilen ihren Inhalt nicht“, d. h. die Variablen, die in einem Scope deklariert werden sind also nicht in einem anderen Scope nicht verfügbar:

Beispiel: Scopes (4)

```
1  #include <stdio.h>
2
3  int main () {
4      int x = 1;
5
6      {
7          int y = 2;
8          printf("%d\n", x);    // Sichtbarkeit nach innen -- 1 wird angezeigt
9          printf("%d\n", y);
10
11         double x = 5.0;
12         printf("%lf\n", x);   // neue Variable
13     }
```

```

14  {
15      // printf("%d\n", y); // Fehler -- keine Sichtbarkeit in andere Scopes
16                               // der gleichen Ebene
17      printf("%d\n", x);      // Sichtbarkeit nach innen -- 1 wird angezeigt.
18                               // Die double-Variable x ist "vergessen"
19
20      double x = 10.0;
21      printf("%lf\n", x);    // neue Variable, wie zuvor.
22  }
23  printf("%d\n", x);
24  }

```

9.2. Funktionen

Eine *Funktion* ist ein Programmabschnitt, der gewissermaßen unabhängig vom Rest des Codes existiert, und von jeder Stelle im Programm *aufgerufen* werden kann. Wir haben mit `printf` und `scanf` schon zwei Funktionen im Detail kennengelernt; die Befehle der math-library sind ebenfalls solche Funktionen. Tatsächlich haben wir auch schon eine Funktion selbst geschrieben: `int main ()` ist der Form nach ebenso ein solches Objekt.

Konzeptuell ist eine Funktion eine Arbeitsanweisung, die einen Wert zurückgibt. Daher braucht eine Funktion zunächst einen *Rückgabotyp*, also den Datentyp des Werts, der berechnet wird. Im Falle der `main` Funktion ist das eben `int`. Es folgt ein Name, über den die Funktion aufgerufen werden kann (wie eben `printf`) so wie eine *Parameterliste*, d. h. eine Liste von Variablen anhand derer z. B. das Ergebnis der Funktion berechnet werden soll. Im Falle der `int main ()` ist dies eine leere Liste; wir werden bald Beispiele sehen, in denen auch Parameter werden.

Als erste Zeile einer Funktion folgen wir also der Syntax:

Syntax: Deklaration von Funktionen

```
Rückgabotyp Funktionsname(Parameterliste) {...}
```

Wir nennen diesen Satz von Informationen (Rückgabotyp, Funktionsname und Parameterliste) auch den *Prototyp* der Funktion. Verwandt damit ist der Begriff der *Signatur*, die nur Rückgabotyp und Parameterliste beschreibt.

Die Parameterliste ist eine durch Kommata getrennte Liste von Datentypen und Bezeichnern (also Variablennamen). Diese Variablennamen sind unabhängig vom Rest des Codes: Eine Funktion öffnet einen eigenen Scope. In der Parameterliste werden die Variablen also erst deklariert. Wir können etwa folgenden Prototypen für Funktion vorschlagen, die die Exponentialfunktion berechnet:

Signatur der Funktion `myExp`

```
double myExp(double x0, int accuracy) {...
```

Nach der Funktionssignatur folgt – eingeschlossen in {geschweifte Klammern} – der Funktionsrumpf bzw. -körper (*Function-Body*). Dabei handelt es sich um beliebigen C-Code, wie wir ihn bisher schon kennen gelernt haben. Teil dieses Codes muss an irgend einer Stelle die Anweisung `return expression` sein. Diese Anweisung sorgt dafür, dass das *aufrufende Programm* den berechneten Wert `expression`

empfängt und die Funktion verlassen wird¹. Folgendes Beispiel zeigt eine mögliche Umsetzung der Funktion `myExp`:

Beispiel: Funktion `myExp`

```
1  double myExp(double x0, int accuracy) {
2      double x          = 1.0;
3      double denominator = 1.0;
4      double result      = 1.0;
5
6      for (int i=1; i<accuracy; i++) {
7          x          *= x0;
8          denominator *= i;
9          result      += x / denominator;
10     }
11
12     return result;
13 }
```

Die Variablen `x0` und `accuracy` werden also schon im Funktions-Prototypen deklariert; `x`, `denominator` und `result` folgen im Funktionsrumpf. Alle diese fünf Variablen gehören demselben Scope an.

Um eine Funktion aufzurufen, setzen wir einfach ihren Namen, gefolgt von der Parameterliste in runden Klammern (). Beim Aufruf schreiben wir in die Parameterliste die Ausdrücke die übergeben werden sollen.

Beispiel: Funktion `myExp` in Anwendung

```
1  #include <stdio.h>
2  #include <math.h>
3
4  double myExp(double x0, int accuracy) {
5      double x          = 1.0;
6      double denominator = 1.0;
7      double result      = 1.0;
8
9      for (int i=1; i<accuracy; i++) {
10         x          *= x0;
11         denominator *= i;
12         result      += x / denominator;
13     }
14
15     return result;
16 }
```

¹Eine Funktion, die keine `return`-Anweisung enthält, wird ebenso verlassen, sobald ihre letzte Zeile ausgeführt wurde. Allerdings kann dann keiner der berechneten Werte vom Hauptprogramm empfangen werden. Stattdessen erhält die aufrufende Stelle einen zufälligen Wert, wie bei einer uninitialisierten Variable.

```

17  int main () {
18      double d, e;
19
20      printf("Genauigkeit\tNäherung\tAbweichung\tRelativ\n");
21      for (int i=0; i<10; i++) {
22          e = myExp(1, i);
23          d = e - exp(1);
24          printf("%d\t%+8.6lf\t%+8.6lf\t%8.3lf\n", i, e, d, d/e);
25      }
26  }

```

Ausführungsbeispiel: Funktion myExp in Anwendung

| Genauigkeit | Näherung | Abweichung | Relativ |
|-------------|-----------|------------|---------|
| 0 | +1.000000 | -1.718282 | -1.718 |
| 1 | +1.000000 | -1.718282 | -1.718 |
| 2 | +2.000000 | -0.718282 | -0.359 |
| 3 | +2.500000 | -0.218282 | -0.087 |
| 4 | +2.666667 | -0.051615 | -0.019 |
| 5 | +2.708333 | -0.009948 | -0.004 |
| 6 | +2.716667 | -0.001615 | -0.001 |
| 7 | +2.718056 | -0.000226 | -0.000 |
| 8 | +2.718254 | -0.000028 | -0.000 |
| 9 | +2.718279 | -0.000003 | -0.000 |

In Zeile 22 wird die Funktion `myExp` aufgerufen. Als Parameter wird jeweils die Zahl 1 und der Wert von `i` übergeben. Diese beiden Werte werden an die Speicherstellen von `x0` und `accuracy` im Scope von `myExp` kopiert, bevor die Ausführung der Funktion beginnt. Mit Zeile 15 wird diese wieder verlassen und der soeben berechnete Wert `result` zurückgegeben, d. h. ist jetzt in der Funktion `main` lesbar. Achtung: die Symbole `denominator` und `result` existieren nur im Scope von `myExp` und sind in `main` nicht ansprechbar.

Der Rückgabewert einer Funktion muss nicht „entgegen genommen“ werden. Wenn der Aufruf nicht im Kontext einer Wertzuweisung oder einer Bedingung (z. B. eines `if`-Blocks) geschieht, wird das Ergebnis der Berechnung einfach verworfen. Wir haben dies tatsächlich bisher bei jedem Aufruf von `printf` benutzt: `printf` „berechnet“ eigentlich die Zahl der Zeichen, die auf dem Bildschirm ausgegeben werden. Da wir aber nie eine Wertzuweisung angeboten haben wurde der Wert verworfen.

Beispiel: Rückgabewert von printf

```

1  #include <stdio.h>
2  int main () {
3      int zeichen = printf("Die Peanuts, Charles M. Schulz\n");
4      printf("Die letzte Ausgabe hatte %d Zeichen.\n", zeichen);
5  }

```

Ausführungsbeispiel: Rückgabewert von printf

```

Die Peanuts, Charles M. Schulz
Die letzte Ausgabe hatte 31 Zeichen.

```


9.2.1. Scopes in bisher bekannten Strukturen

Neben Funktionen legen auch andere Strukturen eigene Scopes an: Alle Blöcke, die durch {geschweifte Klammern} eingefasst werden, stellen einen eigenen Scope dar. Das betrifft insbesondere **if**, **for** und **while**. Aus diesem Grund existieren Variablen, die in einer **for**-Schleife deklariert werden auch nur innerhalb der Schleife.

9.2.2. Forward Declaration

Wie schon bei den Variablen gilt: Eine Funktion muss deklariert sein, *bevor* sie aufgerufen werden kann. Im obigen Beispiel ist dies der Fall. `myExp` steht *vor* `main`. Möchte man allerdings eine Funktion aufrufen, die erst *nach* `main` steht, so muss der Funktions-Prototyp explizit *vor* `main` deklariert werden. Wir nennen diese Technik *forward declaration*:

Beispiel: Funktion `myExp` mit forward declaration

```
1  #include <stdio.h>
2  #include <math.h>
3
4  double myExp(double x0, int accuracy); // Deklaration ohne Body
5
6  int main () {
7      double d, e;
8
9      printf("Genauigkeit\tNäherung\tAbweichung\tRelativ\n");
10     for (int i=0; i<20; i++) {
11         e = myExp(1, i);
12         d = e - exp(1);
13         printf("%d\t%+8.6lf\t%+8.6lf\t%8.3lf\n", i, e, d, d/e);
14     }
15 }
16
17 double myExp(double x0, int accuracy) {
18     double x          = 1.0;
19     double denominator = 1.0;
20     double result      = 1.0;
21
22     for (int i=1; i<accuracy; i++) {
23         x          *= x0;
24         denominator *= i;
25         result      += x / denominator;
26     }
27
28     return result;
29 }
```

Dies ist insbesondere dann wichtig, wenn Funktionen in wechselseitiger Abhängigkeit stehen: Eine Funktion `FuncA` soll eine andere Funktion `FuncB` aufrufen. (Ohne Forward Declaration) muss `FuncB` vor `FuncA` deklariert werden, damit an der entsprechenden Stelle in `FuncA` das aufzurufende Objekt `FuncB` „bekannt“ ist. Wenn nun aber ebenfalls auch von `FuncB` aus wiederum `FuncA` gestartet werden soll, lässt sich die Aufgabe ohne Forward Declaration nicht lösen.

Sehen Sie sich hierzu auch folgendes Beispiel an:

Beispiel: Forward declaration mit wechselseitigem Aufruf

```
1  #include <stdio.h>
2
3  int funcA(int depth);
4  int funcB(int depth);
5
6  int main () {
7      printf("Final result: %d\n", funcA(3));
8  }
9
10 int funcA(int depth) {
11     printf("Call to funcA with depth %d\n", depth);
12     if (depth) {
13         return 2 * funcB(depth - 1);
14     } else {
15         return 1;
16     }
17 }
18
19 int funcB(int depth) {
20     printf("Call to funcB with depth %d\n", depth);
21     if (depth) {
22         return 3 * funcA(depth - 1);
23     } else {
24         return 1;
25     }
26 }
```

Ausführungsbeispiel: Forward declaration mit wechselseitigem Aufruf

```
Call to funcA with depth 3
Call to funcB with depth 2
Call to funcA with depth 1
Call to funcB with depth 0
Final result: 12
```

Wir starten in `main`: hier wird in Zeile 7 `funcA` aufgerufen und der Wert 3 als Parameter übergeben. Entsprechend wird als nächstes Zeile 11 ausgeführt und `Call to funcA with depth 3` ausgegeben. Da die lokale Variable `depth` den Wert 3 hat, entscheidet das `if` in Zeile 12 auf `true`, und Zeile 13 wird ausgeführt. Diese soll `funcA` verlassen und einen Wert zurück geben. Bevor dieser Wert zurückgegeben werden kann, muss er aber erst berechnet werden, was die „Berechnung“ von `funcB` erfordert. Also erhält `funcB` den Parameter Wert des Ausdrucks `depth - 1`, was also zu 2 ausgewertet wird.

Folgerichtig wird als nächstes Zeile 20 ausgeführt, und die Ausgabe `Call to funcB with depth 2` erscheint: Wir haben wieder eine Variable mit dem Namen `depth`, die aber unabhängig von `funcA` ist – jede Funktion hat seinen *eigenen* Scope. 2 ist ungleich 0, daher entscheidet auch das `if` in Zeile 21 auf `true`, und wir finden uns in einer ähnlichen Situation wie zuvor: Zur Berechnung des Funktionswerts von `funcB` muss zunächst noch einmal `funcA` ausgewertet werden.

Naiv könnte man annehmen, wir würden nun in `funcA` „zurückkehren“. Tatsächlich aber steht dieser Aufruf unabhängig vom ersten „Betreten“ von `funcA`: eine neue *Instanz* (*Stack Frame*), d. h. ein eigener Speicherbereich mit seinen *eigenen Variablen* wird angelegt und Code darauf ausgeführt. Entsprechend haben wir hier eine *dritte* Speicherstelle, die lokal über den Namen `depth` angesprochen werden kann

und die hier den Wert 1 beinhaltet. Noch einmal wird in Zeile 13 die Funktion `funcB` aufgerufen, für die ebenso eine zweite Instanz geöffnet wird.

Diese zweite Instanz von `funcB` schließlich erhält für ihre Variable `depth` den Wert 0, so dass das `if` in Zeile 21 schließlich auf `false` entscheidet, und hier den Wert 1 zurück gibt.

Dieser Wert 1 geht nun an die aufrufende Stelle – d. h. die zweite Instanz von `funcA` in Zeile 13. Damit wird also der Ausdruck `2 * 1` berechnet zu 2, und wieder zurück gegeben. Dieser Wert 2 landet bei wiederum bei der Stelle, die die zweite Instanz von `funcA` aufgerufen hatte – die erste Instanz von `funcB` in Zeile 22. Dort berechnet das Programm also den Wert des Ausdrucks `3 * 2` zu 6 und gibt dies an die erste Instanz von `funcA` zurück. Hier schließlich wird noch der Wert von `2 * 6` in Zeile 13 ausgewertet und endlich an die Funktion `main` zurück gegeben.

9.2.3. Übergabe „By Reference“ und „By Value“

Funktionen arbeiten immer nur mit *Kopien* der übergebenen Parameter. Sollen die Parameter selbst sich ändern, so kann anstelle des Wertes (Übergabe „by value“) ihre Adresse an die Funktion übergeben werden (Übergabe „by reference“). In der Funktion kann dann mit den bekannten Pointer-Techniken gearbeitet werden, um auf die Speicherstelle zuzugreifen.

Beispiel: Funktion mit Wertübergabe by value und by reference

```
1  #include <stdio.h>
2
3  int setToZero(int x, int * y) {
4      x = 0;          // Ändert die lokale Variable x (existiert nur in diesem Scope)
5      *y = 0;         // Ändert die Speicherstelle an der Adresse von y
6      return 0;       // Dummy-Rückgabewert
7  }
8
9  int main () {
10     int x = 7, y = 7;
11     setToZero(x, &y);
12
13     printf("x=%d, y=%d\n", x, y);
14 }
```

Ausführungsbeispiel: Funktion mit Wertübergabe by value und by reference

x=7, y=0

9.2.4. Datentyp `void`

Funktionen die keinen Wert zurückgeben können mit dem „Datentyp“ `void` ausgezeichnet werden. Es handelt sich hierbei tatsächlich um einen „Nicht-Datentyp“, der explizit ausdrückt, dass diese Funktion keinen Wert zurückgibt.

Beispiel: void Funktion

```

1  #include <stdio.h>
2
3  void showChameleon() {
4      printf("          _.-\n");
5      printf("      _,-, -' ' ' ' _,- }' _.' ' _.-.\n");
6      printf("      _,-, -' ' ' ' _,- ( @)' _,-,\n");
7      printf("      _,-, -' ' ' ' _,- _.-.-----' ' }\n");
8      printf("      : _.' ' _.-== ' ' _,-, : : ' ' }\n");
9      printf("      } _.' ' _.- / _,-, : : : _,-,\n");
10     printf("      : : _.-, _,-, _,-, _.-, _.-, _.-, _.-\n");
11     printf("      : ; _.-' ' _.- : _,-, ' ; _.-, _.-, _.-\n");
12     printf("      { _.-, _.-, _.-, _.-, _.-, _.-, _.- }\n");
13     printf("      } _.-, _.-, _.-, _.-, _.-, _.-\n");
14     printf("      : _.-, _.-, _.-, _.-, _.-, _.-\n");
15     printf(" _.-' } _.-, _.-, _.-, _.-, _.-\n");
16     printf(" _.-' { { ; ; , ' _.-\n");
17     printf("      } } : ; _,- } \n");
18     printf("      { ' , _.-, ' _.-\n");
19     printf("      pils ' , _.-, _.-, _.-\n");
20     printf("      _.-, _.-, _.-\n");
21
22     // http://ascii.co.uk/art/chameleon
23 }
24
25 int main () {
26     showChameleon();
27 }

```

Für ältere Versionen des Compilers mussten auch leere Parameterlisten mit dem Schlüsselwort **void** markiert werden. Statt **int main ()** lesen sie daher in der Literatur auch manchmal die Zeile **int main (void)**

An dieser Stelle sei außerdem erwähnt, dass `void *` für einen Pointer auf beliebige Daten steht. Bei Wertzuweisungen an Variablen vom Typ `void *` wird nur überprüft, ob der zugewiesene Datentyp ein Pointer ist; ob es sich etwa um einen `int *` oder einen `double *` handelt, wird dagegen nicht geprüft.

Beispiel: void *

```
1  int main () {
2      int      i;
3      double   d;
4
5      int      * ip = &i;
6      double   * dp = &d;
7      void     * vp = &i;
```

```

8 // ip = d; // Fehler: d ist kein int-typ
9 ip = i; // Warnung: i ist kein Pointer
10 ip = dp; // Warnung: unverträgliche Pointer-Typen
11 vp = ip; // okay -- keine Prüfung des Pointer-Typs
12 vp = i; // Warnung: i ist kein Pointer
13 ip = vp; // okay -- void pointer wird nicht geprüft.
14 }

```

Die entsprechenden Warnungen des Compilers lauten:

Compilerausgabe: void *

```

myProgram.c: In function 'main':
myProgram.c:9:6: warning: assignment makes pointer from integer without a cast
[-Wint-conversion]
    ip = i; // Warnung: i ist kein Pointer
    ^
myProgram.c:10:6: warning: assignment from incompatible pointer type
[-Wincompatible-pointer-types]
    ip = dp; // Warnung: unverträgliche Pointer-Typen
    ^
myProgram.c:12:6: warning: assignment makes pointer from integer without a cast
[-Wint-conversion]
    vp = i; // Warnung: i ist kein Pointer
    ^
myProgram.c:9:6: warning: 'i' is used uninitialized in this function
[-Wuninitialized]
    ip = i; // Warnung: i ist kein Pointer
    ~~~~^

```

Natürlich dürfen **void *** Variablen nicht dereferenziert werden. Da **void** keinen Datentyp beschreibt, kann der Compiler auch nicht entscheiden, wie viele Bytes ab der Start-Adresse gelesen werden sollen, oder wie dieser Speicherblock zu interpretieren ist. Stattdessen müssen zur tatsächlichen Arbeit mit **void *** Typecasts ausgeführt werden (siehe Abschnitt 3.3).

Ein Anwendungsbeispiel für **void *** ist etwa die Funktion `memcpy` aus der `string.h`. Diese Funktion kopiert eine bestimmte Zahl Bytes von einer Quell-Adresse an eine Ziel-Adresse. Statt aber beispielsweise nur für **int**-Arrays deklariert zu sein, wird der Funktion ein beliebiger Pointer übergeben; der Prototyp lautet:

```
void * memcpy (void * dest, const void * src, size_t count);
```

(Siehe <https://en.cppreference.com/w/c/string/byte/memcpy>).

Dabei bedeutet **const**, dass die Werte an der Adresse **src** sich nicht ändern dürfen. (Dies erlaubt dem Compiler verschiedene Optimierungs-Schritte.)

Sowohl für **dest** als auch für **src** können beliebige Pointer übergeben werden. Damit können mit demselben Befehl Arrays beliebigen Typs kopiert werden:

Beispiel: memcpy (1)

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main () {
5      int    src_i[10], dst_i[10];
6      double src_d[10], dst_d[10];
7
8      for (int i=0; i<10; i++) {
9          src_i[i] = 2 * i;
10         src_d[i] = 2 * i;
11     }
12
13     memcpy(dst_i, src_i, sizeof(src_i));    // kopiert 40 Bytes
14     memcpy(dst_d, src_d, sizeof(src_d));    // kopiert 80 Bytes
15
16     for (int i=0; i<10; i++) {
17         printf("%2d, %2d, %4.1lf\n", i, dst_i[i], dst_d[i]);
18     }
19 }
```

Derselbe Befehl `memcpy` nimmt in Zeile 13 einen `int *` an, und wird in Zeile 14 auch mit einem `double *` als Parameter aufgerufen.

Es ist sogar möglich, „typenfremd“ zu kopieren:

Beispiel: memcpy (2)

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main () {
5      char src[10] = {1, 1, 1, 1, 1,    // 1 Byte per char
6                      1, 1, 1, 1, 1};
7      short dst[ 5];                    // 2 Byte per short
8
9      memcpy(dst, src, 10);
10
11     for (int i=0; i<5; i++) {
12         printf("%i\n", dst[i]);
13     }
14 }
```

Ausführungsbeispiel: memcpy (2)

```
257
257
257
257
257
```

Das Array `src` besteht aus 10 Elementen, die jeweils ein Byte groß sind, und das Bitpattern 00000001

besitzen. Diese werden über `memcpy` an die Adresse kopiert, auf die `dst` verweist. Das Array `dst` ist vom Typ `short`, d. h. pro Feld-Element werden 2 Byte gelesen. Ein solches `short`-Element wird also aus zwei `char`-Elementen zusammengesetzt und hat das Bitpattern 0000000100000001 – dies entspricht der Dezimalzahl 257.

9.2.5. Globale Variablen

Bisher haben wir nur Variablen gesehen, die *innerhalb* von Funktionen deklariert wurden (sei dies in der `main` oder in anderen Funktionen). Es ist aber auch möglich, Variablen ganz außerhalb jedes Scopes zu deklarieren und sie damit *global* verfügbar zu machen:

Beispiel: Globale Variablen

```
1  #include <stdio.h>
2
3  int global = 5;  // Deklaration AUSSERHALB von Funktionen
4
5  void setToSeven () {
6      printf("\tvorher : \t%d\n", global);
7      global = 7;
8      printf("\tnachher: \t%d\n", global);
9  }
10
11 int main () {
12     printf("bei Start: \t\t%d\n", global);
13
14     setToSeven();
15     printf("nach setToSeven: \t%d\n", global);
16
17     global = 9;
18     printf("nach Änderung in main: \t%d\n", global);
19
20     setToSeven();
21 }
```

Ausführungsbeispiel: Globale Variablen

```
bei Start:           5
    vorher :         5
    nachher:         7
nach setToSeven:     7
nach Änderung in main: 9
    vorher :         9
    nachher:         7
```

Die Variable `global` steht *außerhalb* aller Scopes und ist wie alle Variablen „nach innen“ sichtbar. Daher können sowohl die Funktion `main` als auch die Funktion `setToSeven` auf die Speicherstelle zugreifen, die mit `global` bezeichnet wird.

Globale Variablen sparsam verwenden

Es erscheint verlockend, alle Variablen global zu deklarieren und damit an jeder Stelle alle Informationen abgreifbar zu machen. Damit wird aber der komplette Mechanismus außer Kraft gesetzt, der strukturierte Programmierung ermöglicht. Die Kapselung von Variablen in Funktionen erlaubt, Arbeitsschritte zu beschreiben, ohne dabei die komplette Struktur des restlichen Programms im Kopf behalten zu müssen. Sind alle Variablen global, so fällt diese Kapselung weg. Nur Variablen, die tatsächlich an *jeder* Stelle des Programms verfügbar sein sollen, machen als globale Variablen Sinn.

Ein Beispiel: Sie wollen ein Brettspiel implementieren. Der Zustand des Spielbretts ist für nahezu alle Programmteile (z. B. Berechnung der Punkte, Auswertung ob das Spiel gewonnen wurde, Darstellung auf dem Bildschirm, ..) relevant. Der Zustand des Bretts könnte daher sinnvoll in einer globalen Variable umgesetzt werden. Dagegen ist das letzte Würfelergebnis nur für bestimmte Aufgaben interessant, und sollte daher *nicht* global gespeichert werden.

Ein weiteres Beispiel: Naturkonstanten wie die Lichtgeschwindigkeit, die Kreiszahl π sind sinnvoll als globale Variablen angelegt.

9.2.6. Was als Funktion auslagern

Immer wieder wird die Frage gestellt, welche Programmteile sinnvoll in Funktionen ausgelagert werden sollten und welche nicht. Während es dafür keine allgemeingültige Formel gibt, kann man zumindest einige Situationen aufführen, in denen es sich anbietet, Code in einer eigenen Funktion anzulegen.

Generell ist es sinnvoll, alle Arbeitsschritte, die eine gedankliche Einheit bilden, in einer Funktion zu verkapseln. Eine gedankliche Einheit ist etwa „Berechne den Wert einer Funktion“ oder „Gib Text auf dem Bildschirm aus“.

Insbesondere sollte aller Code, den man mehrfach verwenden will in Funktionen stehen. Kopierter Code macht das Programm länger und damit schwieriger zu warten. Vor allem aber ergibt sich daraus eine Fehlerquelle, da Änderungen an einem kopierten Codeteil verlangen, auch alle anderen Stellen zu ändern, an denen diese Kopien auftauchen. Es ist leicht, diese Stellen zu vergessen und so inkonsistenten und fehlerhaften Code zu erhalten.

Schließlich kann es von Vorteil sein, selbst Codeblöcke in Funktionen zu setzen, die nur einmal aufgerufen werden. Ein Funktionsname sollte eindeutig beschreiben, welche Aufgabe die Funktion erfüllt. Ist dies

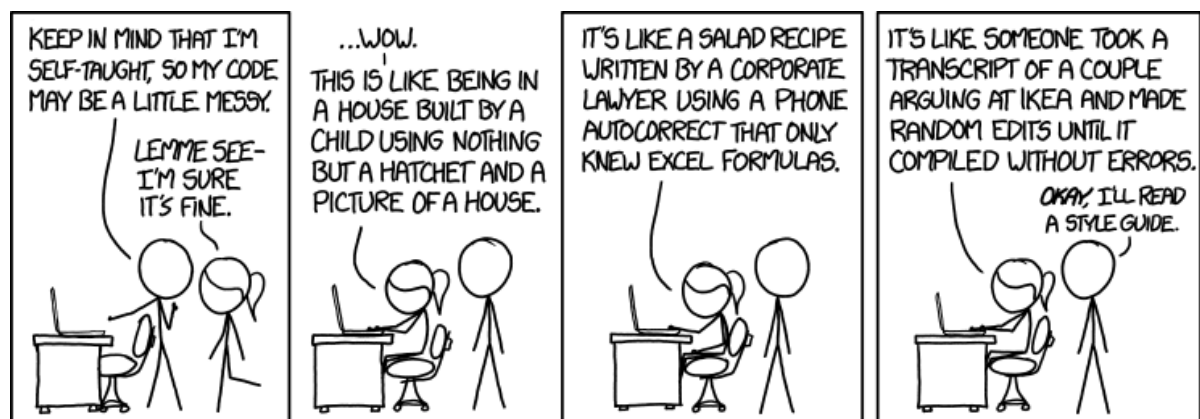


Abbildung 9.1.: Code Quality. Quelle: <https://xkcd.com/1513/>

gegeben, so kann der Ablauf eines Programms klarer im Code gelesen werden, ohne jedes Detail Zeile für Zeile nachvollziehen zu müssen. Sehen Sie sich dazu folgenden Code an:

Beispiel: main eines Spiels

```
1  /* ... #include-Zeilen stehen hier ... */
2
3  /* ... Funktionen stehen hier ... */
4
5  int main () {
6      showTitleAnimation();
7
8      int selectedMenuPoint = 0;
9      do {
10         showMenuScreen();
11         selectedMenuPoint = askForMenuOption();
12         performAction(selectedMenuPoint);
13     } while(selectedMenuPoint);
14 }
```

Hier wird vor Start des Spiels eine Animation gezeigt. Im Anschluss wird jeweils in Folge ein Menübildschirm dargestellt, eine Usereingabe angenommen und dann dafür gesorgt, dass die Auswahl des Users ausgeführt wird. Dies wird solange durchgeführt, bis die Funktion `askForMenuOption` den Wert 0 zurückgibt. Dies könnte (und sollte) z. B. dann der Fall sein, wenn der User auswählt, das Spiel zu beenden.

Obwohl in diesem Beispiel nicht klar ist, wie die Animation im Detail aussieht, oder welche Menüpunkte angeboten werden, ist klar, wie das Programm organisiert ist. Neue Features können leicht eingefügt werden. Soll etwa eine Animation ausgeführt werden, sobald ein Menüpunkt ausgewählt wurde, so kann nach Zeile 11 noch eine neue Funktion `showMenuAnimation(selectedMenuPoint)` eingefügt werden.

Funktionen können sich auch untereinander aufrufen, wie wir in Abschnitt 9.2.2 gesehen haben. Dies „kostet“ einigen Verwaltungsaufwand (um den wir als ProgrammiererInnen uns aber nicht explizit kümmern müssen). Der Compiler muss für jeden Aufruf einer Funktion mitspeichern, an welche Stelle bei Verlassen der aufgerufenen Funktion zurück gesprungen werden soll; außerdem müssen die Speicherbereiche für die einzelnen Funktionen angelegt und wieder frei gegeben werden. Bei geringer *Aufruftiefe* (bis etwa 20 Aufruf-Ebenen) fällt dies nicht weiter ins Gewicht. Stärkere Verschachtelung macht Programme aber merklich langsamer. Natürlich ist es vor allem schwierig, eine Struktur mit so vielen Ebenen zu überblicken. Daher möchte ich Ihnen diesen Gedanken als einzige Beschränkung für die Sinnhaftigkeit von Funktionen mitgeben: halten Sie die Verschachtelung klein und übersichtlich. In Kapitel 16 werden wir diesen Gedanken nochmals aufgreifen.

9.2.7. Rückgabe mehrerer Werte

Eine Funktion kann nur entweder nichts (`void`) oder genau einen einzigen Wert zurückgeben. Es gibt aber Situationen, in denen dieselbe Funktion *mehrere* Werte zugleich berechnen soll. Um dies zu ermöglichen stehen uns zwei Techniken zur Verfügung: Wir können ein *Array* oder ein `struct` zurückgeben.

Arrays werden über Pointer verwaltet, daher muss der Rückgabetyt auch ein Pointer sein. (Tatsächlich gibt unsere Funktion also nur einen einzigen Wert zurück; dieser gibt dann aber eine Speicherstelle an, an der alle berechneten Werte zu finden sind.)

Sehen Sie sich folgendes Beispiel an, das ein Rezept für Obstsalat auf verschiedene Menschenmengen anpasst:

Beispiel: Umrechnung eines Rezepts

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  // https://www.chefkoch.de/rezepte/654221166964999/Bunter-Obstsalat.html
4
5  double * fruitsaladForPersons(double persons) {
6      double * reVal = malloc(8 * sizeof(*reVal));
7
8      reVal[0] = persons/4.0 * 3; // apples
9      reVal[1] = persons/4.0 * 2; // bananas
10     reVal[2] = persons/4.0 * 2; // peaches
11     reVal[3] = persons/4.0 * 2; // kiwis
12     reVal[4] = persons/4.0 * 1; // mangos
13     reVal[5] = persons/4.0 * 1; // oranges
14     reVal[6] = persons/4.0 * 200; // grams of grapes
15     reVal[7] = persons/4.0 * 50; // grams of walnuts
16
17     return reVal;
18 }
19
20 int main () {
21     double persons = 2;
22     double * requirements = fruitsaladForPersons(persons);
23
24     printf("Obstsalat für %1.0lf Personen:\n", persons);
25     printf("\tÄpfel      : %5.1lf\n", requirements[0]);
26     printf("\tBananen    : %5.1lf\n", requirements[1]);
27     printf("\tPfirsiche: %5.1lf\n", requirements[2]);
28     printf("\tKiwis      : %5.1lf\n", requirements[3]);
29     printf("\tMangos     : %5.1lf\n", requirements[4]);
30     printf("\tOrangen    : %5.1lf\n", requirements[5]);
31     printf("\tTrauben    : %5.1lf g\n", requirements[6]);
32     printf("\tWalnüsse   : %5.1lf g\n", requirements[7]);
33
34     free(requirements);
35 }
```

Ausführungsbeispiel: Umrechnung eines Rezepts

```
Obstsalat für 2 Personen:
    Äpfel      : 1.5
    Bananen    : 1.0
    Pfirsiche: 1.0
    Kiwis      : 1.0
    Mangos     : 0.5
    Orangen    : 0.5
    Trauben    : 100.0 g
    Walnüsse   : 25.0 g
```

Beachten Sie hier auch, dass der in Zeile 5 reservierte Speicherbereich wieder freigegeben werden muss. Dies geschieht in Zeile 35.

structs werden in Kapitel 10 besprochen.

9.2.8. Speicherklasse **static**

Variablen sind immer nur in ihrem aktuellen Scope sichtbar, und werden i. d. R. zerstört, sobald der aktuelle Scope verlassen wird. Manchmal soll aber der Wert einer Speicherstelle erhalten bleiben, selbst wenn die Ausführung den Scope verlässt, in dem die Variable definiert wurde. Bisher können wir dies nur lösen, indem wir die entsprechende Variable in einem Scope anlegen, das für die gesamte Lebensdauer des Programms existiert (also in der **main** oder als globale Variable), und einen Pointer auf diesen Wert „durchreichen“.

Beispiel: Funktionsaufrufe mitzählen (1)

```
1  #include <stdio.h>
2
3  void foobar(int * callCounter) {
4      (*callCounter)++;
5      printf("Aufruf #%d\n", *callCounter);
6  }
7
8  int main () {
9      int calls = 0;
10
11     foobar(&calls);
12     foobar(&calls);
13 }
```

Ausführungsbeispiel: Funktionsaufrufe mitzählen (1)

```
Aufruf #1
Aufruf #2
```

Dieser Code funktioniert zwar, ist aber fehleranfällig. Zunächst muss ein Anwender den Zweck von **calls** kennen, um die Funktion **foobar** richtig zu nutzen. Insbesondere darf **calls** nicht versehentlich geändert werden. Nicht zuletzt „belegt“ diese Technik einen Variablennamen in der **main**.

Als Alternative dazu können Variablen in der Funktion **foobar** mit dem Schlüsselwort **static** deklariert werden. Auf diese Weise bleibt die Speicherstelle reserviert, d. h. gespeicherte Werte erhalten, auch wenn die Funktion später wieder aufgerufen wird. Der Scope-Mechanismus hingegen wird nicht außer Kraft gesetzt. In Code sieht das dann so aus:

Beispiel: Funktionsaufrufe mitzählen (2)

```
1  #include <stdio.h>
2
3  void foobar() {
4      static int callCounter = 0;
5      callCounter++;
6      printf("Aufruf #%d\n", callCounter);
7  }
8
9  int main () {
10     foobar();
11     foobar();
12     // printf("%d\n", callCounter); -- unzulässig, falscher Scope
13 }
```

Der Compiler legt *ein einziges Mal* die Speicherstelle für `callCounter` an, selbst wenn (wie hier) die Funktion `foobar` mehrfach aufgerufen wird. Entsprechend findet auch die Wertzuweisung in Zeile 4 nur ein einziges Mal statt. Danach „erinnert“ sich der Compiler an die bereits angelegte Speicherstelle sobald `foobar` „betreten“ wird, „vergisst“ aber diesen Variablennamen (nicht aber die Speicherstelle selbst) wieder, wenn die Funktion verlassen wird.

9.3. Optimierung: `inline`

Bei einem Funktionsaufruf werden die Werte von Parametern kopiert. Ebenso wird die Stelle gespeichert, von der weg gesprungen wurde, so dass beim Verlassen der Funktion die Ausführung nach dem Sprung-Befehl dort fortgesetzt werden kann. Dieses Kopieren und Speichern kostet Zeit. Bei häufig wiederkehrenden Aufrufen führt dies zu merklichen Performance-Einbrüchen.

Um dem entgegen zu wirken, können *kurze* Funktionen als `inline` deklariert werden². Dies teilt dem Compiler mit, dass der Funktionskörper an der aufrufenden Stelle eingefügt werden sollte, statt einen normalen Sprung und Rücksprung durchzuführen. Ergebnis ist eine größere ausführbare Datei (da derselbe Code an mehreren Stellen eingefügt wird), der aber schneller läuft. Für die ProgrammiererInnen ergibt sich sonst kein Unterschied – Funktionsaufruf sowie Scoping funktionieren wie bisher besprochen.

Um eine Funktion als `inline` zu deklarieren, schreibt man eine forward declaration und stellt dieser das Schlüsselwort `inline` voran:

Syntax: `inline`-Funktion

```
inline Rückgabety Funktionsname (Parameterliste);
```

Beispiel: Maximum von zwei Ganzzahlen finden

```
1  inline int max (int a, int b);
2      int max (int a, int b) {return a > b ? a : b;}
```

²Es gibt allerdings keine Garantie dafür, dass der Compiler die Funktion wirklich an entsprechende Stellen im einfügt. Es handelt sich also lediglich um einen Hinweis an den Compiler.

9.4. Funktionszeiger

Alle Objekte mit denen wir arbeiten müssen im Speicher abgelegt werden. Sie haben daher notwendigerweise auch eine Adresse. Dies umfasst einzelne Zahlen, Gruppen von Zahlen (Arrays) und eben auch Programmcode. Bei der Ausführung eines Programms wird dieses zunächst in den Arbeitsspeicher kopiert, und dann die Maschinencode-Anweisungen ausgeführt, die in der Datei vermerkt waren. Wir können die Adressen von bestimmten Programnteilen ausfindig machen und mit der Ausführung an diese Adressen springen. Zu diesem Zweck existiert das Konzept von **Funktionszeigern**, also Pointer auf Funktionen.

Funktionszeiger sind wie alle Pointer Variablen, die zunächst deklariert werden müssen. Diese Deklaration enthält alle Informationen der Funktionssignatur, also Rückgabewert der Funktion und Parameterliste. Dies drückt sich in der folgenden (leider etwas unhandlichen Syntax) aus:

Syntax: Deklaration Funktionszeiger

```
Rückgabebetyp (*Variablenname) (Parameterliste);
```

Solchen Variablen kann nun die Adresse einer beliebigen Funktion (mit passender Signatur) zugewiesen werden. Dies geschieht *ohne* jegliche Operatoren, einfach über den Funktionsnamen:

Beispiel: Funktionszeiger (1)

```
1  #include <stdio.h>
2
3  void foo() {
4      printf("blerb.\n");
5  }
6
7  int main () {
8      void (*funcPtr)() = foo;
9  }
```

Der Codeabschnitt, auf den dieser Funktionspointer zeigt, kann nun mit dem neuen Symbol `funcPtr` aufgerufen werden, ganz als würde man direkt mit `foo` arbeiten:

Beispiel: Funktionszeiger (2)

```
1  #include <stdio.h>
2
3  void foo() {
4      printf("blerb.\n");
5  }
6
7  int main () {
8      void (*funcPtr)() = foo;
```

```

1   printf("direkt : ");
2   foo();
3
4   printf("funcPtr: ");
5   funcPtr();
6   }

```

Ausführungsbeispiel: Funktionszeiger (2)

```

direkt : blerb.
funcPtr: blerb.

```

Tipp

Der Name einer Funktion beschreibt selbst einen Funktionszeiger! Der Aufruf einer Funktion ist eine *Operation*, vergleichbar mit dem Dereferenzieren (Operator `*`). Daher müssen auch Funktionen, die keine Parameter erwarten, mit leeren Klammern `()` aufgerufen werden.

Der Zweck dieser Technik ist, das Verhalten anderer Funktionen vom Ergebnis anderer Berechnungen abhängig zu machen, ohne sich dabei auf die Art der Berechnung festzulegen.

Als Beispiel sei die Funktion `qsort`³ genannt (definiert in `stdlib.h`). Diese Funktion kann Arrays beliebigen Typs nach beliebigen Kriterien sortieren. Die Sortier-Kriterien werden in Form einer Funktion `comp` beschrieben, die zwei Werte `a` und `b` miteinander vergleicht. (Hier verwende ich den Namen `comp`, um den folgenden Abschnitt leichter lesbar zu halten; die Vergleichsfunktion darf in der Anwendung aber einen beliebigen Namen haben.)

Da Arrays *beliebigen* Typs sortiert werden, können an `comp` nicht die *Werte* `a` und `b` übergeben werden, sondern nur ihre *Adressen* `&a` und `&b`.

Wenn `a` *vor* `b` in der sortierten Liste auftauchen soll, soll der Rückgabewert von `comp(&a, &b)` gleich `-1` sein. Falls dagegen `b` vor `a` sortiert werden soll, so muss `+1` zurückgegeben werden. Schließlich wird `comp` den Wert `0` zurückgeben, wenn `a` und `b` gleichwertig sind. (*Gleichwertig* bedeutet in diesem Kontext nicht zwingend *gleich*. Wir könnten zum Beispiel Zahlen nach ihrer Quersumme sortieren. Für diese Sortierung sind die Zahlen `19` und `91` dann gleichwertig, nicht aber gleich.)

Die cpp-Referenz teilt uns die Signatur der Funktion `qsort` mit:

Prototyp der Funktion `qsort`

```

void qsort( void * ptr, size_t count, size_t size,
            int (*comp)(const void *, const void *) );

```

Betrachten wir die einzelnen Parameter:

- Der erste Parameter `ptr` ist ein Pointer auf ein beliebiges Array und daher vom Typ `void *`.
- Mit `count` wird die Zahl der Elemente des Arrays beschrieben. (Der Typ `size_t` ist ein Alias (ein alternativer Name) für `unsigned int`.)
- Der Wert `size` beschreibt den Speicherbedarf *eines* Elements des Arrays. Wir brauchen diese Information, da `qsort` u. a. für `int`-, `double`- oder `char`-Arrays funktionieren soll, die allesamt unterschiedlichen Speicherbedarf haben.

³Der Sortieralgorithmus *quicksort*, der dieser Funktion zugrunde liegt, wird in der Vorlesung *Algorithmen und Datenstrukturen* an der Universität Regensburg genauer besprochen.

- Die letzte Zeile schließlich benennt den Parameter `comp`, der eine Vergleichsfunktion benennt. Diese Funktion wird einen `int` zurück geben, und nimmt zwei `const void *` als Argumente an. Dabei bedeutet `const` wieder, dass `comp` die Werte von `a` und `b` nur lesen, nicht aber verändern wird. Der Modifier `const` gehört zum Datentyp und muss daher auch in unserer Funktions- Signatur auftauchen. Wieder gilt: Da der Datentyp von `a` und `b` beliebig ist, können nur `void *` übergeben werden.

Mit diesen Informationen können wir das (leicht abgewandelte) Beispiel von der cpp-Referenz verstehen:

Beispiel: qsort mit zwei Sortiermethoden

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int comp_ascending(const void* a, const void* b) {
5      int arg1 = *(const int *) a;
6      int arg2 = *(const int *) b;
7
8      if (arg1 < arg2) return -1;
9      if (arg1 > arg2) return 1;
10     return 0;
11 }
12
13 int comp_descending(const void* a, const void* b) {
14     return -comp_ascending(a, b);
15 }
16
17 int main(void) {
18     int ints[] = { -2, 99, 0, -743, 2, 4 };
19     int size = sizeof(ints) / sizeof(*ints);
20
21     printf("Aufsteigende Sortierung:\n");
22     qsort(ints, size, sizeof(int), comp_ascending);
23
24     for (int i = 0; i < size; i++) {
25         printf("%d ", ints[i]);
26     }
27     printf("\n");
28
29     printf("Absteigende Sortierung:\n");
30     qsort(ints, size, sizeof(int), comp_descending);
31
32     for (int i = 0; i < size; i++) {
33         printf("%d ", ints[i]);
34     }
35     printf("\n");
36 }

```

Ausführungsbeispiel: `qsort` mit zwei Sortiermethoden

Aufsteigende Sortierung:

-743 -2 0 2 4 99

Absteigende Sortierung:

99 4 2 0 -2 -743

Obwohl die Funktion `qsort` nur einmal programmiert wurde, kann sie mit diesen Techniken also *generisch* benutzt werden, d. h. sie ist für viele verschiedene Zwecke einsetzbar, die bei der Erstellung von `qsort` noch nicht einmal vorgesehen waren.

10. Speicher-Strukturen

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

C.A.R. Hoare

Werte im Speicher bilden oft funktionelle Einheiten. In einem Jump'n'Run-Game gehören beispielsweise die x- und y-Koordinate, Blickrichtung sowie die Lebenspunkte der Spielfigur „zusammen“. Es bietet sich an, Aufgaben wie die Animation beim Laufen, die Überprüfung auf Kollision mit anderen Spielelementen und ähnliches jeweils in Funktionen auszulagern. Diesen Funktionen müssen dann aber jeweils alle genannten Daten als Parameter übergeben werden. Dies kann schnell lästig werden und bietet zusätzlich eine Fehlerquelle, da die Parameter in der richtigen Reihenfolge übergeben werden müssen. An dieser Stelle wollen wir Methoden betrachten, die solche thematischen Einheiten zu kompakten Blöcken zusammen zu fassen.

10.1. Sammlungen von Werten: **structs**

Ein **struct** ist ein *selbstdefinierter Datentyp*, in dem mehrere Datensätze zu einem Block zusammengefasst werden. Man kann sich eine **struct** also wie eine Sammlung von Tabellenköpfen vorstellen. **structs** haben einen Typen-Namen, sowie für jede Zeile der Tabelle einen Datentyp und einen Feldnamen.

Eine **struct** wird folgendermaßen deklariert:

Syntax: Deklaration einer **struct**

```
struct Name {  
    Datentyp Feldname1;  
    Datentyp Feldname2;  
    ...  
};
```

Wie üblich ist **Name** dabei eine alphanumerische Zeichenkette aus bis zu 40 Zeichen, die einen eindeutigen Namen beschreibt, d. h. die nicht schon eine Variable im aktuellen Scope oder eine Funktion benennt.

Datentyp ist ein beliebiger Datentyp wie z. B. **int**. Ebenso erlaubt sind Zeiger auf Datentypen oder auch andere **structs** bzw. Zeiger darauf.

Die **Feldnamen** sind alphanumerische Zeichenketten von bis zu 40 Zeichen Länge, die jedoch *nicht* global eindeutig sein müssen: eine **struct** öffnet ihren eigenen Scope (vgl. die {geschweiften Klammern}, in die die Scope-Felder eingeschlossen sind).

Grundsätzlich kann eine **struct** an jeder Stelle im Programm angelegt werden; sie ist dann wie gewohnt nur für den aktuellen Scope gültig. Meist wird die Definition einer **struct** im gesamten Programm gebraucht; man setzt sie daher üblicherweise global, also vor/außerhalb der **main**.

Mit dieser Definition einer **struct** kann man nun Variablen dieses Typs anlegen. Eine Variable eines solchen Typs kann man sich als Spalte einer solchen Tabelle vorstellen. Eine solche Variable wird deklariert mit der Syntax:

Syntax: Deklaration einer **struct**

```
struct Name Variablenname;
```

Beispiel: Deklaration einer **struct**

```
1 struct Karteieintrag {
2     char vorname[50];
3     char nachname[50];
4     int geburtsjahr;
5     int geburtsmonat;
6     int geburtstag;
7 };
8
9 int main () {
10     struct Karteieintrag Cessi, Chami;
11 }
```

Visualisierung: Sammlung Tabellenköpfe

| Tabellenkopf | Cessi | Chami |
|--------------|--------|-------------|
| vorname | Ce | Lisl |
| nachname | Saurus | Kohlenstoff |
| geburtsjahr | 1897 | 1996 |
| geburtsmonat | 9 | 9 |
| geburtstag | 30 | 26 |

10.1.1. Direkter Zugriff auf **struct**-Elemente

Im obigen Beispiel haben wir die **struct** definiert und Variablen des Typs **struct Karteieintrag** deklariert. Um nun die einzelnen Felder anzusprechen (d.h. die Tabellen-Elemente zu lesen und zu schreiben) bedienen wir uns der Syntax **Variable.Feldname**. Achten hier auf die Unterscheidung zwischen *Datentyp* (z.B. **struct Karteieintrag**), *Variable* (z.B. Cessi) und *Feldname* (z.B. vorname).

Beispiel: Deklaration einer **struct** und Schreiben in eine **struct**-Variable

```
1 #include <string.h>
2
3 struct Karteieintrag {                // Deklaration Datentyp
4     char vorname[50];                // Deklaration Feldelemente
5     char nachname[50];
6     int geburtsjahr;
7     int geburtsmonat;
8     int geburtstag;
9 };
10
11 int main () {
12     struct Karteieintrag Cessi, Chami; // Deklaration Variablen
13
14     strcpy(Cessi.vorname, "Ce");      // Zugriffe auf Feldelemente in 'Cessi'
15     strcpy(Cessi.nachname, "Saurus");
16     Cessi.geburtsjahr = 1897;
17     Cessi.geburtsmonat = 9;
18     Cessi.geburtstag = 30;
```

```

19 strcpy(Chami.vorname, "Lisl");           // Zugriffe auf Feldelemente in 'Chami'
20 strcpy(Chami.nachname, "Kohlenstoff");
21 Chami.geburtsjahr = 1996;
22 Chami.geburtsmonat = 9;
23 Chami.geburtstag = 26;
24 }

```

Anstatt nun also mit zwei mal fünf getrennten Variablen hantieren zu müssen, können wir nun fünf Werte „als Paket“ behandeln; ein solches Paket darf beispielsweise als Parameter einer Funktion übergeben. Vergleichen Sie die beiden folgenden Funktionen sowie ihre Aufrufe:

Beispiel: Vergleich Funktion mit struct-Parameter und getrennte Variablen

```

1  #include <stdio.h>
2  #include <string.h>
3
4  struct Karteieintrag {
5      char vorname[50];
6      char nachname[50];
7      int geburtsjahr;
8      int geburtsmonat;
9      int geburtstag;
10 };
11
12 void KarteiZeigenGetrennt (
13     char vorname[50],
14     char nachname[50],
15     int geburtsjahr,
16     int geburtsmonat,
17     int geburtstag
18 ) {
19     printf("Vorname:\t%s\n", vorname);
20     printf("Nachname:\t%s\n", nachname);
21     printf("Geburtstag:\t%04d-%02d-%02d\n",
22         geburtsjahr, geburtsmonat, geburtstag
23     );
24 }
25
26 void KarteiZeigenStruct (struct Karteieintrag person) {
27     printf("Vorname:\t%s\n", person.vorname);
28     printf("Nachname:\t%s\n", person.nachname);
29     printf("Geburtstag:\t%04d-%02d-%02d\n",
30         person.geburtsjahr, person.geburtsmonat, person.geburtstag
31     );
32 }
33
34 int main () {
35     struct Karteieintrag Cessi, Chami;
36
37     /* Eintragen von Werten in Variablen 'Cessi', 'Chami' */

```

```

38     KarteiZeigenGetrennt(
39         Cessi.vorname,
40         Cessi.nachname,
41         Cessi.geburtsjahr,
42         Cessi.geburtsmonat,
43         Cessi.geburtstag
44     );
45
46     printf("\n");
47
48     KarteiZeigenStruct(Chami);
49 }

```

Ausgabebeispiel: Vergleich Funktion mit struct-Parameter und getrennte Variablen

```

Vorname:      Ce
Nachname:     Saurus
Geburtstag:   1897-09-30

Vorname:      Lisl
Nachname:     Kohlenstoff
Geburtstag:   1996-09-26

```

Beide Varianten erzeugen eine Ausgabe derselben Form. Bei `KarteiZeigenGetrennt` müssen jedoch alle fünf Werte im Aufruf getrennt aufgeführt werden (Zeilen 38-44), und erhalten in der Funktions-signatur einen eigenen Eintrag (Zeilen 12-18). Bei der `struct`-Version `KarteiZeigenStruct` dagegen kann der komplette Datensatz „als Einheit“ übergeben werden (Zeile 48). Entsprechend kürzer fällt auch die Funktionssignatur aus (Zeile 26): Hier wird nur eine einzelne Variable `person` vom Typ `struct Karteieintrag` entgegengenommen.

Auch `structs` können ineinander verschachtelt werden:

Beispiel: Verschachtelte Structs

```

1  struct Datum {
2      int jahr;
3      int monat;
4      int tag;
5  };
6
7  struct Karteieintrag {
8      char vorname[50];
9      char nachname[50];
10     struct Datum geburtstag;
11 };
12
13 int main () {
14     struct Karteieintrag Tiny;
15     Tiny.geburtstag.jahr = 1994;
16     Tiny.geburtstag.monat = 2;
17     Tiny.geburtstag.tag = 3;
18 }

```

10.1.2. Aliase für Datentypen: `typedef`

`structs` erlauben zwar bedeutend übersichtlichere Strukturen; es wird aber bald lästig, wiederholt das Schlüsselwort `struct` selbst zu tippen. Aus diesem Grund wird die Deklaration von `structs` häufig mit `typedef` verbunden.

`typedef` wird benutzt, um zu bestehenden Namen von Datentypen alternative Namen zu definieren (sogenannte *Aliase*). Dies folgt der Syntax:

Syntax: Deklaration einer `struct`

```
typedef BezeichnerOriginaltyp Alias;
```

Sobald auf diese Weise ein `Alias` definiert ist, kann dieser an jeder Stelle im Code benutzt werden und ersetzt so den gesamten Ausdruck `BezeichnerOriginaltyp`.

Beispiel: `typedef` und `struct`

```
1 struct Karteieintrag_struct {
2     char vorname[50];
3     char nachname[50];
4     int geburtsjahr;
5     int geburtsmonat;
6     int geburtstag;
7 };
8
9 typedef struct Karteieintrag_struct Karteieintrag;
10
11 int main () {
12     Karteieintrag Cessi, Chami;
13 }
```

`Typedef` kann auch für vordefinierte Datentypen benutzt werden, die häufig getippt werden müssen:

Beispiel: `typedef` und lange Standard-Typen

```
1 typedef unsigned long long int bignum;
2 bignum globalvar = 9999999999999999;
```

Die Definition des Alias `Karteieintrag` für `struct Karteieintrag_struct` erspart in Zeile 12 die Wiederholung des Schlüsselworts `struct`; insbesondere gilt das auch für alle weiteren Referenzen auf den Datentyp wie etwa in der Signatur von Funktionen. Neben der Typenbezeichnung `Karteieintrag` kann aber auch weiterhin `struct Karteieintrag_struct` als Synonym verwendet werden.

Noch kompakter und bequemer wird dies bei der Verwendung von *anonymen structs*, also `structs` ohne eigenen Namen. Hier kann die Auflistung der Felder direkt in den `typedef`-Befehl mit aufgenommen werden:

Beispiel: typedef und struct

```
1  typedef struct {
2      char vorname[50];
3      char nachname[50];
4      int geburtsjahr;
5      int geburtsmonat;
6      int geburtstag;
7  } Karteieintrag;
8
9  int main () {
10     Karteieintrag Cessi, Chami;
11 }
```

Schließlich ist es auch möglich, im selben Zuge *abgeleitete* Typen zu definieren, z. B. einen Pointer auf die soeben definierte **struct**:

Beispiel: typedef und struct mit abgeleitetem Typ

```
1  typedef struct {
2      char vorname[50];
3      char nachname[50];
4      int geburtsjahr;
5      int geburtsmonat;
6      int geburtstag;
7  } Karteieintrag, * pKarteieintrag;
8
9  int main () {
10     Karteieintrag Cessi;
11     pKarteieintrag CessiRef1 = &Cessi;
12     Karteieintrag * CessiRef2 = &Cessi;
13 }
```

Durch das * in Zeile 7 wird ausgedrückt, dass das folgende Symbol **pKarteieintrag** einen Pointer auf die definierte Struktur beschreibt. Folglich wird in Zeile 11 mit **CessiRef1** eine Variable definiert, die ein Pointer auf **Karteieintrag** ist. Daneben existiert auch die bereits bekannte Form aus Zeile 12. Die Variablen **CessiRef1** und **CessiRef2** haben also denselben Datentyp.

Sinn und Unsinn von typedefs

Die Möglichkeit, mit **typedef** kürzere Aliase für unhandliche Ausdrücke einzuführen birgt die Gefahr, sich in einem undurchsichtigen Gewirr von Definitionen zu verstricken. Gerade wenn von **typedefs** noch andere Typen abgeleitet werden, ist es oft schwer, die „Verwandtschaftsbeziehungen“ noch nachzuvollziehen und sinnvollen Code zu schreiben.

Nach meiner Erfahrung gibt es zwei Situationen, in denen **typedef** die Arbeit in C tatsächlich leichter macht:

- **structs** sowie Pointer auf **structs**, so wie oben gezeigt
- Funktionszeiger, wie wir sie in Abschnitt 9.4 kennengelernt haben. Die reine Umdefinition bestehender Typennamen dagegen sollte eher vermieden werden.

Im Kontext von Funktionszeigern macht die Verwendung von **typedef** den Code nicht nur kürzer, sondern bedeutend intuitiver lesbar (sofern man sprechende Typennamen definiert). Sehen Sie folgendes Beispiel an:

typedef und Funktionszeiger

```
typedef int (*funcPtr_comp) (const void *, const void *);

/* Originaldefintion wie von der cpp-Referenz
 * void qsort( void * ptr, size_t count, size_t size,
 *            int (*comp)(const void *, const void *) );
 */
void qsort (void * ptr, size_t count, size_t size,
            funcPtr_comp comp);
```

Die Originaldefinition erlaubt zwar, direkt Rückgabetyt und Typ der Parameter zu entnehmen; es ist aber schwierig, hier auf einen Blick zu erfassen, welches der Name des Parameters ist, und was zur Beschreibung des Typs gehört. In der zweiten Version wird vorausgesetzt, dass der/die ProgrammiererIn die Signatur von `comp` kennt; dafür wird die Signatur von `qsort` übersichtlich und entspricht eher einer menschlichen Gedankeneinheit. Die Meinungen hierzu gehen bisweilen aber weit auseinander.

structs können auch Pointer auf Instanzen desselben Typs enthalten:

Beispiel: struct mit Referenz auf Instanz gleichen Typs

```
1 struct linkedList {
2     double          content;
3     struct linkedList * next;
4 };
```

Während das Symbol `struct linkedList` schon „in der Definition der `struct`“ zur Verfügung steht, sind mit `typedef` definierte Symbole erst *nach Abschluss* des Befehls verwendbar:

Beispiel: typedef und struct mit Referenz auf Instanz gleichen Typs (fehlerhaft)

```
1 typedef struct {
2     double          content;
3     // linkedList * next;    // Fehler: linkedList noch nicht definiert
4 } linkedList;
```

Um für solche Strukturen dennoch den Komfort von **typedef** nutzen zu können, arbeitet man hier mit *benannten structs* (im Gegensatz zu *anonymen structs*):

Beispiel: typedef und struct mit Referenz auf Instanz gleichen Typs (korrekt)

```
1 typedef struct LL_struct {    // linkedList struct
2     double          content;
3     struct LL_struct * next;  // okay: struct LL_struct kann aufgelöst werden
4 } linkedList;                // Symbol linkedList steht ab hier zur Verfügung.
```

In Kapitel 17 werden wir uns ausführlicher mit solchen Strukturen beschäftigen.

10.1.3. **structs** und Pointer

Wie Sie bereits festgestellt haben, müssen wir in C häufig mit Pointern arbeiten. Im vorigen Abschnitt wurden bereits Pointer auf **structs** angesprochen. Dazu sei hier zuerst ein Syntaxelement vorgestellt, und weiter einige Beispiele besprochen:

Kombinierter Pointer-Feldzugriffs-Operator

Haben wir einen Pointer auf eine **struct**, so muss für einen Feldzugriff der Pointer zuerst dereferenziert (Operator `*`) werden und anschließend das angesprochene Feld über den Feldzugriffoperator (`.`) ausgewählt werden. Da der Feldzugriffsoperator eine höhere Präzedenz hat als die Dereferenzierung (d. h. `.` wird vor `*` ausgewertet; siehe Tabelle B.7), muss mit Klammern gearbeitet werden:

Beispiel: Dereferenzierung und Feldzugriff

```
1  /* Definition von Karteieintrag und pKarteieintrag wie im vorigen Kapitel */
2  int main () {
3      Karteieintrag    Tiny;
4      pKarteieintrag ptr = &Tiny;
5
6      printf("Jahr: %4d\n", (*ptr).geburtsjahr);
7  }
```

Diese umständliche Form kann durch den kombinierten *Pointer-Feldzugriffs-Operator* `->` ersetzt werden:

Beispiel: kombinierter Operator

```
1  /* Definition von Karteieintrag und pKarteieintrag wie im vorigen Kapitel */
2  int main () {
3      Karteieintrag    Tiny;
4      pKarteieintrag ptr = &Tiny;
5
6      printf("Jahr: %4d\n", ptr->geburtsjahr);
7  }
```

Pointer als Feldelemente

Selbstverständlich können auch die Feldelemente einer **struct** selbst Pointer enthalten. In den obigen Beispielen etwa sind `vorname` und `nachname` **char**-Arrays und damit Pointer. Der Dereferenzierungs-Operator `*` wirkt in diesem Fall auf das Feldelement:

Beispiel: Zugriff auf Pointer-Feldelemente (1)

```
1  /* Definition von Karteieintrag und pKarteieintrag wie im vorigen Kapitel */
2  int main () {
3      Karteieintrag Tiny;
4
5      printf("Erster Buchstabe von Tinys Vornamen: %c\n", *Tiny.vorname);
6  }
```

Ebenso verhält es sich mit dem Array-Index-Zugriffsoperator `[]`:

Beispiel: Zugriff auf Pointer-Feldelemente (2)

```
1  /* Definition von Karteieintrag und pKarteieintrag wie im vorigen Kapitel */
2  int main () {
3      Karteieintrag      Tiny;
4
5      printf("Erster Buchstabe von Tinys Vornamen: %c\n", Tiny.vorname[0]);
6  }
```

Pointer in Basis-Variable und in den Feldelementen

Ist die Variable, über die die **struct** angesprochen wird, selbst eine Pointer-Variable, so wird der kombinierte Pointer-Feldzugriffs-Operator -> zuerst aufgelöst; anschließend wirken * bzw []:

Beispiel: kombinierter Operator und Pointer-Feldelemente

```
1  /* Definition von Karteieintrag und pKarteieintrag wie im vorigen Kapitel */
2  int main () {
3      Karteieintrag      Tiny;
4      pKarteieintrag ptr = &Tiny;
5
6      printf("Erster Buchstabe von Tinys Vornamen: %c\n", *ptr->vorname );
7      printf("Erster Buchstabe von Tinys Vornamen: %c\n", ptr->vorname[0]);
8  }
```

Siehe hierzu auch Tabelle B.7.

Arrays von Structs

Man kann auch *Arrays von structs* anlegen. In diesem Fall setzt man den Array-Zugriffsoperator [] an die Array-Variable. Unabhängig davon können weitere Operationen auf Array-Feldelemente durchgeführt werden:

Beispiel: Array von structs

```
1  typedef struct {
2      char vorname[50];
3      char nachname[50];
4      int geburtsjahr;
5      int geburtsmonat;
6      int geburtstag;
7  } Karteieintrag, * pKarteieintrag;
```

```

8  int main () {
9      Karteieintrag listeGeburtstage[3];
10
11     strcpy(listeGeburtstage[0].vorname, "Ce");
12     strcpy(listeGeburtstage[0].nachname, "Saurus");
13     listeGeburtstage[0].geburtsjahr = 1897;
14     listeGeburtstage[0].geburtsmonat = 9;
15     listeGeburtstage[0].geburtstag = 30;
16
17     strcpy(listeGeburtstage[1].vorname, "Lisl");
18     strcpy(listeGeburtstage[1].nachname, "Kohlenstoff");
19     listeGeburtstage[1].geburtsjahr = 1996;
20     listeGeburtstage[1].geburtsmonat = 9;
21     listeGeburtstage[1].geburtstag = 26;
22
23     strcpy(listeGeburtstage[2].vorname, "Tiny");
24     strcpy(listeGeburtstage[2].nachname, "Mürrisch");
25     listeGeburtstage[2].geburtsjahr = 1994;
26     listeGeburtstage[2].geburtsmonat = 2;
27     listeGeburtstage[2].geburtstag = 3;
28
29     printf("Erster Buchstabe Vorname 3: %c\n", *listeGeburtstage[2].vorname );
30     printf("Erster Buchstabe Vorname 3: %c\n", listeGeburtstage[2]->vorname[0]);
31 }

```

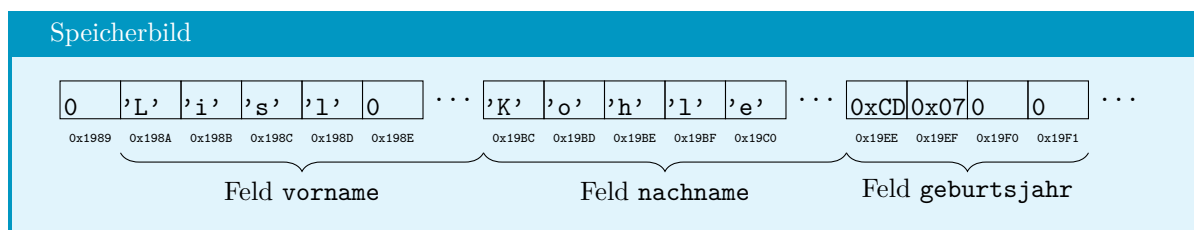
Machen Sie sich hier nochmal die Bedeutung der einzelnen Symbole klar:

- `Karteieintrag` ist ein *Datentyp* und beschreibt nur die Struktur einer Variable; das Symbol selbst beschreibt aber keine Werte.
- `pKarteieintrag` ist ebenfalls ein *Datentyp*. Er beschreibt aber keine *struct*, sondern einen *Pointer* darauf, d. h. die Information, wo im Speicher die tatsächliche Instanz der *struct* zu finden ist. Verbunden mit dieser Adresse ist die Information, welche Art von Werten an der angegebenen Speicherstelle zu finden sind – eben die *struct* `Karteieintrag`.
- `geburtstag` ist ein Element der *struct* `Karteieintrag`, und damit ohne Kontext ein nicht deklariertes Symbol. Nur in Verwendung mit Instanzen von `Karteieintrag` kann dieses Symbol gebraucht werden.
- `listeGeburtstage` ist ein Array vom Typ `Karteieintrag` und damit eine Variable vom Typ `pKarteieintrag`. In dieser Variablen ist nur eine Zahl gespeichert, die die Adresse des ersten Elements des Arrays im Speicher angibt.
- `listeGeburtstage[0]` ist nun tatsächlich ein Objekt vom Typ `Karteieintrag`, d. h. eine Sammlung von fünf Variablen. Man spricht auch von einer *Instanz* von `Karteieintrag`. Es handelt sich um das erste Element der Liste. Dieser Ausdruck kann z. B. an Funktionen übergeben werden, die als Argument ein Objekt vom Typ `Karteieintrag` erwarten (wie im vorigen *Beispiel: Vergleich: Funktion mit struct-Parameter und getrennte Variablen* die Funktion `KarteiZeigenStruct`.)
- `listeGeburtstage[1].geburtsjahr` ist ein *int*-Wert. Er gehört zu der Instanz `listeGeburtstage[1]`, also zur zweiten Instanz des Typs `Karteieintrag` im Array.
- `listeGeburtstage[2].vorname` ist ein `char *`, der zum dritten Element des Arrays `listeGeburtstage` gehört.

- `*listeGeburtstage[2].vorname` ist die Dereferenzierung des oben genannten Objekts, und damit ein `char`.
- `listeGeburtstage->geburtstag` ist gleichbedeutend mit `(*listeGeburtstage).geburtstag` oder `listeGeburtstage[0].geburtstag`. Es handelt sich um einen `int`-Wert, der zum ersten Element des Arrays gehört.
- `listeGeburtstage[3]` ist ein Verweis auf das vierte Element des Arrays, das aber nicht existiert. Es handelt sich hierbei also um eine fehlerhafte Referenz.

10.1.4. `structs` im Speicher

Wir können uns Instanzen von `structs` ähnlich vorstellen wie Arrays. Im Unterschied zu diesen werden die einzelnen Elemente nicht über einen *Index* angesprochen, sondern über einen *Feldnamen*. Außerdem haben die einzelnen Elemente nicht denselben Datentyp. Im Grunde aber wird dieselbe Speicherstruktur angelegt:



Da `structs` einen Datentyp definieren, kann mit `sizeof` der Speicherbedarf *einer Instanz* ermittelt werden. Es handelt sich dabei um die Summe der Speicherbedarfe der einzelnen Elemente:

Beispiel: `sizeof` und `struct`

```

1  struct Karteieintrag {
2      char vorname[50];
3      char nachname[50];
4      int geburtsjahr;
5      int geburtsmonat;
6      int geburtstag;
7  };
8
9  int main () {
10     struct Karteieintrag Tiny;
11
12     printf("%lu\n", sizeof(Karteieintrag));
13     printf("%lu\n", sizeof(Tiny));
14 }

```

Ausgabebeispiel: `sizeof` und `struct`

```

112
112

```

Diese 112 Bytes pro `Karteieintrag` kommen aus jeweils 50 Bytes für die beiden `char`-Arrays `vorname` und `nachname` zusammen sowie dreimal 4 Bytes für die `ints` `geburtsjahr`, `geburtsmonat` und `geburtstag`.

10.1.5. Operationen mit **structs**

Während viele Rechenoperationen mit **structs** denkbar sind, ist in C nur die Wertzuweisung = definiert. Der Grund hierfür ist, dass für zusammengesetzte Typen nicht eindeutig ist, wie etwa eine Addition stattfinden soll, oder nach welchen Kriterien entschieden wird, welche Instanz „größer als“ eine andere ist. Wo solche Aufgaben gelöst werden sollen, schreibt man Funktionen:

Beispiel: `sizeof` und `struct`

```
1  #include <stdio.h>
2  #include <math.h>
3
4  typedef struct {
5      double x;
6      double y;
7  } Punkt2D;
8
9  int vergleichPunkt2D (Punkt2D a, Punkt2D b) {
10     /* Vergleich nach Kriterium 'Entfernung zum Nullpunkt'
11      * Rückgabewerte:
12      * 0, wenn a und b gleich weit entfernt
13      * -1, wenn a näher am Nullpunkt als b
14      * +1, wenn b näher am Nullpunkt als a
15      */
16
17     double la = hypot(a.x, a.y),    // hypot: definiert in math.h
18            lb = hypot(b.x, b.y);    // berechnet sqrt(a.x * a.x + b.x * b.x)
19
20     if (la == lb) {return 0;}
21     else          {return la < lb ? -1 : +1;}
22 }
23
24 int main () {
25     Punkt2D a = {5.0, 5.0}, b = a;
26     printf("%d\n", vergleichPunkt2D(a, b));
27     // printf("%d\n" a == b); Fehler: kein direkter Vergleich von struct-Instanzen
28 }
```

10.1.6. Initializer-Syntax

Ähnlich wie bei Arrays können **struct**-Instanzen bei der Deklaration gleich Werte zugewiesen werden, indem wir diese in {geschweiften Klammern} aufführen:

Beispiel: Initializer-Syntax bei structs

```
1  typedef struct {
2      int  jahr;
3      int  monat;
4      int  tag;
5  } Datum;
6
7  int main () {
8      Datum tag = {1989, 3, 29};
9  }
```

Diese Kurzform ist jedoch *ausschließlich* bei der Deklaration von **struct**-Instanzen zulässig. Im folgenden Beispiel sind Deklaration und Wertzuweisung getrennt; daher funktioniert dort die Initializer-Syntax nicht mehr:

Beispiel: Initializer-Syntax bei structs

```
1  typedef struct {
2      int  jahr;
3      int  monat;
4      int  tag;
5  } Datum;
6
7  int main () {
8      Datum tag;
9      // tag = {1989, 3, 29}; Fehler: Initializer-Syntax außerhalb von Deklaration
10 }
```

Dies liegt daran, dass mit diesem abhängig vom Kontext sehr unterschiedliche Ergebnisse erzielt werden. Enthält die **struct** ein **char**-Array fester Größe (wie im obigen Beispiel das **struct Karteieintrag**: Die Felder **vorname** und **nachname** sind beide **char**-Arrays der Größe 50), so wird ein String-Literal (ein Ausdruck in "doppelten Anführungszeichen") direkt in das Array kopiert; allgemeinen Pointern dagegen wird die Adresse des Literals selbst zugewiesen.

Beispiel: Initializer-Syntax bei Strings

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct {
5      char  constSizeArray[50];
6      char * charPointer;
7  } twoStrings;
```

```

8  int main () {
9      twoStrings x = {"Pure Vernunft Darf Niemals Siegen",    // constSizeArray
10                     "Pure Vernunft Darf Niemals Siegen"};    // charPointer
11
12     printf("Startadresse der Instanz:\t%p\n" , (void *)&x);
13     printf("Adresse von constSizeArray:\t%p\n", (void *)x.constSizeArray));
14     printf("Adresse von charPointer:\t%p\n",    (void *)x.charPointer));
15 }

```

Ausgabebeispiel: Initializer-Syntax bei Strings

| | |
|-----------------------------|----------------|
| Startadresse der Instanz: | 0x7ffc0d4172c0 |
| Adresse von constSizeArray: | 0x7ffc0d4172c0 |
| Adresse von charPointer: | 0x562bd763b808 |

Das erste String Literal (Zeile 9) soll dem Feld `constSizeArray` zugewiesen werden, für das ab Beginn der `struct`-Instanz 50 Byte frei gehalten werden. Die Zeichenkette wird auf genau diese Speicherstelle kopiert. In den Zeilen 12 und 13 wird daher auch dieselbe Adresse für `&x` (die Adresse der Instanz `x`) und für `x.constSizeArray` (die Adresse des `char`-Arrays `x.constSizeArray`) ausgegeben. Zeile 14 nennt dagegen eine ganz andere Adresse, da für das Feld `charPointer` lediglich eine Adresse speichert, nicht aber den String selbst. Entsprechend wird hier die *Adresse des String-Literals* abgelegt; wir können uns dies vorstellen als *die Adresse des String-Literals im Code*.

Diese letzten Überlegungen waren vermutlich schwer zu verfolgen; bitte entnehmen Sie diesem Abschnitt, dass die Initializer-Syntax im Allgemeinen eine schnelle Wertzuweisung zu `struct`-Instanzen erlaubt, dass aber beim Umgang mit Arrays Vorsicht geboten ist. Wo Sie sich nicht sicher sind, können Sie jederzeit die zuerst gezeigten expliziten Wertzuweisungen benutzen.

10.2. Casting-Interface: `unions`

`unions` sind `structs`, bei denen alle Felder auf dieselbe Speicheradresse gelegt werden. Das bedeutet, dass man über verschiedene Bezeichner dieselben Werte bearbeiten kann. Dies kann für manche Elektronik-Anwendungen nützlich sein, in denen Datenpakete in anderen „Einheiten“ versandt und empfangen werden als sie das Steuerprogramm verarbeitet.

Beispiel: Im folgenden Beispiel wird einem `int` ein `char`-Array überlagert:

Beispiel: union aus int und chars

```

1  #include <stdio.h>
2
3  typedef union {
4      int i;
5      char c[4];
6  } intUnion;
7
8  int main () {
9      intUnion u;
10
11     for (int i=254; i<260; i++) {
12         u.i = i;
13
14         printf("Integer: %d\n", u.i);
15         for (int n=0; n<4; n++) {
16             printf("  Byte %d: %02x", n, u.c[n]);
17         }
18         printf("\n");
19     }
20 }

```

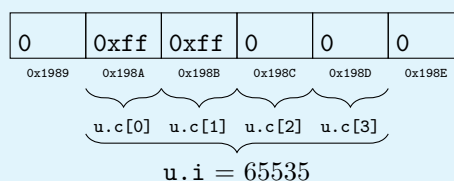
Ausgabebeispiel: union aus int und chars

```

Integer: 254
  Byte 0: fe  Byte 1: 00  Byte 2: 00  Byte 3: 00
Integer: 255
  Byte 0: ff  Byte 1: 00  Byte 2: 00  Byte 3: 00
Integer: 256
  Byte 0: 00  Byte 1: 01  Byte 2: 00  Byte 3: 00
Integer: 257
  Byte 0: 01  Byte 1: 01  Byte 2: 00  Byte 3: 00
Integer: 258
  Byte 0: 02  Byte 1: 01  Byte 2: 00  Byte 3: 00
Integer: 259
  Byte 0: 03  Byte 1: 01  Byte 2: 00  Byte 3: 00

```

Speicherbild



Implizit bildet eine **union** also ein Interface auf eine Variable mit verschiedenen Typecasts.

10.3. Automatisch numerierte Symbole: `enums`

`enums` sind Aliase für `unsigned ints`, bei denen die einzelnen Zahlen neue Symbole zugewiesen bekommen. Die zugewiesenen Symbole können so gewählt werden, dass der Verwendungszweck klar wird. Syntaktisch lehnen sich `enums` an `structs` an; die einzelnen Elemente eines `enums` werden aber durch Kommata getrennt, nicht durch Semikolons:

Syntax: Deklaration einer `enum`

```
typedef enum {
    Symbol0, Symbol1, ...
} Name;
```

`Symbol0` steht dann für den Wert 0, `Symbol1` für 1 und so fort.

Betrachten Sie den folgenden Code: Es wird geprüft, welche von zwei Karten den höheren Rang im Spiel hat, wobei *Herz* als Trumpffarbe zählt. Sticht eine Karte `k1` eine andere Karte `k2`, so ist das Ergebnis `-1`; wenn `k2` den höheren Rang hat, wird `+1` zurückgegeben. Bei Gleichheit wird die Funktion zu 0 ausgewertet¹.

Beispiel: Deklaration einer `struct`

```
1  typedef enum {Herz, Karo, Kreuz, Pik} Kartenfarbe;
2
3  typedef struct {
4      int      Wert;
5      Kartenfarbe Farbe;
6  } Karte;
7
8  int vergleicheKarten (Karte k1, Karte k2) {
9      if (k1.Farbe == Herz) {
10         if (k2.Farbe == Herz) {goto vergleicheKartenWerte;}
11         else {return -1;}
12     }
13
14     if (k2.Farbe == Herz) {return 1;}
15
16     vergleicheKartenWerte:
17     if (k1.Wert == k2.Wert) {return 0;}
18     else {return k1.Wert > k2.Wert ? -1 : 1;}
19 }
```

Intern wird das Feld `Farbe` als `unsigned int` angelegt, und das Symbol `Herz` zum Wert 0 übersetzt. Es ist ebenso zulässig, `k1.Farbe == 0` zu prüfen oder eine Zuweisung `k2.Farbe = 3` zu machen. Jedoch ist mit der Vergabe von `enum`-Symbolen sofort ein Kontext gegeben, der den Code besser interpretierbar macht.

¹Sie sehen hier eine Verwendung von `goto`. Wie erwähnt sollte der Gebrauch dieses Befehls sehr stark hinterfragt werden. An dieser Stelle führt die Alternative – eine `if-else if`-Konstruktion mit verketteten Bedingungen – jedoch zu einer weniger schnell erfassbaren Struktur. Versuchen Sie gerne als Übung, eine Funktion mit gleicher Wirkung ohne `goto` zu schreiben, und vergleichen Sie das Ergebnis.

10.4. C++

Ausflug: C++

In C++ sind **structs** das zentrale Element; sie werden dort um viele Konzepte erweitert. Aus Feinheiten, die hier nicht behandelt werden können, benutzt man in C++ eher das Schlüsselwort **class**, das im Wesentlichen dieselbe Wirkung hat wie das hier gezeigte **struct**.

Wir bilden alle *Objekte* als solche **class** ab und ordnen ihnen *Methoden* zu, d. h. Funktionen, die mit den Feldern einer **class** arbeiten^a. Ein *Objekt* ist eine nahezu beliebige gedankliche Einheit von Werten, die untereinander in einem funktionalen Zusammenhang stehen. *Methoden* stehen nicht mehr „neben“ der **class**, sondern sind gänzlich in diese eingebunden. Über ein Schlüsselwort **this** „sieht“ die Methode alle Felder, die zu einer Instanz der **class** gehören.

Klassen erlauben *Zugriffskontrolle*, verbieten also das direkte Lesen und Schreiben von bestimmten Feldern. Nur die Methoden einer Klasse können alle Felder bearbeiten. Auf diese Weise soll ein versehentliches Ändern von Werten und damit das Entstehen von inkonsistenten (unsinnigen) Zuständen verhindern.

Weiter erlaubt C++ die Definition von *Konstruktoren* und *Destruktoren*, d. h. Programmen, die automatisch aufgerufen werden, wenn eine Instanz der Klasse erzeugt wird, bzw. wenn sie das Ende ihrer Lebensdauer erreicht. Diese speziellen Methoden reservieren i. d. R. Speicher für komplexere Strukturen bzw. geben diesen wieder frei.

Schließlich ist es in C++ auch möglich, eigene Operatoren zu definieren, und so direkt mit dem Symbol + zwei Instanzen einer Klasse zu addieren anstatt hierfür explizit eine Funktion aufrufen zu müssen.

Auch hier soll lediglich auf den Umfang des Themas hingewiesen und ein Anreiz geschaffen werden, den Kurs *C++ und Qt* der Universität Regensburg zu besuchen.

^aUmgangssprachlich wird oft die Bezeichnung *Methode* aufgeführt. Im C++ Standard ist allerdings nicht die Rede von Methoden, sondern von *Member Functions* bzw. *Member Data*.

11. Modulare Programmierung

Everything is awesome!

Emmet Brickowski

Für besonders große Projekte bietet es sich an, den Code auf mehrere Dateien zu verteilen, und diese nach Aufgaben zu sortieren. Dies erlaubt es, eine „übergeordnete“ Kapselung umzusetzen, die ähnlich funktioniert wie schon bei den Funktionen besprochen wurde (siehe Kapitel 9), und die den Einfluss begrenzt, den die einzelnen Programmkomponenten aufeinander haben.

Ein *Modul* ist eine C-Code-Datei, die nur „ihre eigenen“ Variablen und Funktionen „sieht“. Innerhalb eines Moduls müssen alle Namen (von Variablen, Funktionen, Sprung-Labels, ...) eindeutig sein, d. h. dürfen nicht doppelt vergeben werden (es sei denn, sie „leben“ jeweils in ihrem eigenen Scope). Ein Modul erzeugt allerdings wiederum einen umfassenden Scope; daher dürfen in zwei getrennten Modulen dieselben Namen vorkommen.

Verbunden werden Module über *Header*-Dateien, die lediglich Definitionen enthalten. Diese Definitionen teilen gewissermaßen mit, welche Funktionen, Variablen, ... geteilt werden, also über die Modulgrenzen hinaus sichtbar sein sollen.

11.1. Trennung von Header- und Modul-Code

Wir haben Header schon im Kontext von Libraries kennen gelernt: mittels `#include`-Zeilen laden wir die nötigen Definitionen, die wir benötigen, um die Funktionen der Libraries zu benutzen. Dem Compiler haben wir mit Kommandozeilenoptionen wie `-lm` mitgeteilt, dass die im Header definierten Funktionen in einem vorkompilierten Modul zu finden sind. Stellen Sie sich im Folgenden vor, der Befehl `#include` würde per Copy&Paste den Inhalt der inkludierten Datei in Ihren Code einfügen.

Die Organisation eines Projekts in Module geschieht mit derselben Mechanik; jedoch bieten wir hier zum *Linken* keine vorkompilierte Bibliothek, sondern tatsächlichen C-Code. Der Header definiert also eine Schnittstelle zwischen den einzelnen Komponenten.

Eine Header-Datei endet auf die Erweiterung `.h` und darf prinzipiell beliebigen C-Code enthalten. Sinnvoll ist es jedoch, hier nur *Definitionen* zu setzen, also Anweisungen, die keinen Einfluss auf die Speicherstruktur haben. Definitionen sind z. B. Funktions-Signaturen (siehe Abschnitt 9.2.2), `structs` und `typedefs` (siehe Kapitel 10). Deklarationen von *Variablen* dagegen dürfen im Header nicht vorkommen, da die Deklaration auch eine Speicherstelle reserviert, d. h. die Speicherstruktur verändert.

Bislang haben wir nur `#include`-Zeilen gesehen, bei denen der Header-Name von <spitzen Klammern> eingefasst war. Dies teilt dem Compiler mit, dass die Header-Datei in einem Standard-Verzeichnis zu finden ist, das bei der Installation des Compilers angelegt wird. Da sich unsere Header i. d. R. im selben Verzeichnis befinden wie unser Code, fassen wir diese stattdessen mit "doppelten Anführungszeichen" ein.

Als Beispiel wollen wir uns vorstellen, wir wollen eine eigene library anlegen, die den Umgang mit 2D-Punkten erleichtert. Wir definieren zuerst im Header `vector2D.h` einen Datentypen und „machen die Funktionen des Moduls sichtbar“:

Beispiel: Header vector2D.h

```
1  typedef struct {
2      double x;
3      double y;
4  } vector2D;
5
6  vector2D v2_sum      (vector2D a, vector2D b);
7  vector2D v2_inv      (vector2D v);
8  vector2D v2_scale    (vector2D v, double f);
9  double   v2_dotProd  (vector2D a, vector2D b);
10 double   v2_length   (vector2D v);
```

Die Implementierung hierzu speichern wir in der Datei vector2D.c:

Beispiel: Modulcode vector2D.c

```
1  #include "vector2D.h"
2  #include <math.h>
3
4  vector2D v2_sum      (vector2D a, vector2D b) {
5      vector2D reVal = {a.x + b.x, a.y + b.y};
6      return reVal;
7  }
8
9  vector2D v2_inv      (vector2D v)            {
10     vector2D reVal = {-v.x, -v.y};
11     return reVal;
12 }
13
14 vector2D v2_scale     (vector2D v, double f)  {
15     vector2D reVal = {f * v.x, f * v.y};
16     return reVal;
17 }
18
19 double   v2_dotProd   (vector2D a, vector2D b) {return a.x * b.x + a.y * b.y;}
20 double   v2_length    (vector2D v)            {return hypot(v.x, v.y);}
```

Da in diesem Modul mit der Definition des Typs `vector2D` gearbeitet wird, muss diese Definition auch „sichtbar“ sein. Daher laden wir in den Modulcode den Header mittels `#include` (Zeile 1). Weiter benutzen wir die Funktion `hypot` (berechnet den Wert $\sqrt{x^2 + y^2}$, also die Länge eines Vektors $\begin{pmatrix} x \\ y \end{pmatrix}$). Diese Funktion ist im Header `<math.h>` definiert, weswegen dieser ebenfalls geladen wird. Während `vector2D.h` in unserem Arbeitsverzeichnis liegt und daher mit "doppelten Anführungszeichen" eingefasst wird, findet sich `math.h` in einem Standard-Verzeichnis, und ist deswegen durch <spitze Klammern> eingefasst.

Achtung: Die Datei `vector2D.c` besitzt *keine* Funktion `main`! Sie kann vom Compiler zu *Object-Code* umgesetzt werden, d. h. in Maschinensprache übersetzt werden; Linken zu einem *ausführbaren Programm* ist mit diesem Code alleine jedoch nicht möglich.

Das *Hauptmodul* findet sich in einer eigenen Datei `myProgram.c`:

Beispiel: Hauptmodulcode myProgram.c

```
1  #include <stdio.h>
2  #include "vector2D.h"
3
4  int main () {
5      vector2D v = {1,1}, w = {2, 3};
6
7      printf("%lf\n", v2_length(v2_sum(v, w)));
8  }
```

Hier stehen nun alle „Befehle“ zur Verfügung, die wir im Header `vector2D.h` deklariert haben. Wir müssen hier nicht explizit die Bibliothek `<math.h>` einbinden, da keine Funktion aus der math-library im Hauptmodul explizit benutzt wird.

Unser Projekt besteht also aus zwei Modulen, die über einen gemeinsamen Header verbunden sind. Diese beiden Module müssen beim Aufruf des Compilers aufgeführt werden:

Compileraufruf: Projekt aus zwei Modulen

```
gcc -std=c11 -Wall -Wextra -Wpedantic myProgram.c vector2D.c -lm -o myProgram
```

Achtung: der hier gezeigte Aufruf besteht nur aus einer *einzigsten Zeile*. Wenn Sie diesen Aufruf eintippen bedarf es *keines* Zeilenumbruchs nach `-lm`. Beachten Sie weiter auch, dass im Compiler-Aufruf der Header *nicht* erwähnt wird!

Wie üblich findet das Kompilieren über den `gcc` statt. Wir listen zunächst einige Optionen auf (C-Standard von 2011, Anzeigen aller relevanten Warnungen) und nennen dann unsere beiden Module (`myProgram.c` und `vector2D.c`). Den Zusatz `-lm` kennen wir als Compiler Option um die Math-Library einzubinden. Wir können dies als ein *drittes* Modul auffassen, das in unserem Projekt verwendet wird. Schließlich nennen wir mit `-o myProgram` den Namen der ausführbaren Datei, die aus unserem Code erzeugt werden soll.

11.2. Speicherklasse `extern`

Wo globale Variablen über Modulgrenzen hinaus zur Verfügung stehen sollen, kommt das Schlüsselwort `extern` zum Einsatz. Es wird einer Variablen-Deklaration vorangestellt, und teilt dem Compiler mit dass an einer anderen Stelle eine Variable mit diesem Namen deklariert wird, und dass diese Variable im aktuellen Modul sichtbar sein soll. Mit einer `extern`-Deklaration wird also *keine Speicherstelle* reserviert!

Betrachten Sie folgende drei Dateien:

Beispiel: Hauptmodulcode myProg.c

```
1  #include <stdio.h>
2  #include "definitions.h"
3
4  int main () {
5      printf("Globale Variable 'global' = %lf\n", global);
6  }
```

Beispiel: Header `definitions.h`

```
1 extern double global;
```

Beispiel: Modul `module.c`

```
1 #include "definitions.h"
2
3 double global = 5.0;
```

Im Hauptmodul `myProg.c` wird in Zeile 5 Bezug auf eine Variable `global` genommen. Dass eine solche Variable existiert wird über den Header `definitions.h` mitgeteilt. Die dortige Zeile `extern double global;` legt aber noch nicht die Variable an, sondern teilt dem Compiler nur mit, dass es ein Objekt mit dem Namen `global` gibt. Dieses Objekt wird schließlich im Modul `module.c` angelegt und erhält dort in Zeile 3 den Wert 5.0.

extern-Anweisungen sind reine Definitionen

`extern`-Zeilen legen keine Speicherstellen an; sie reservieren lediglich ein Symbol. Zu jedem `extern` muss es ein Gegenstück geben, das tatsächlichen Speicher belegt. Dieses Gegenstück muss *eindeutig* sein, d. h. es darf nur eine einzige „echte“ Variable geben, die den Namen trägt, der in der `extern`-Zeile referenziert wird.

Da `extern` nur ein Symbol benennt, schreiben wir diesen Befehl in die Header-Dateien. Die zugehörige Speicherstelle „lebt“ in (genau) einem Modul.

Wozu dieser Aufwand? Sollte es nicht genügen, eine Variable direkt in einem Header zu deklarieren? Betrachten Sie dazu folgendes Codebeispiel:

Beispiel: Hauptmodulcode `myProg.c`

```
1 #include <stdio.h>
2 #include "definitions.h"
3
4 int main () {
5     printf("Globale Variable 'global' = %lf\n", global);
6 }
```

Beispiel: Header `definitions.h`

```
1 double global = 5;
2
3 void setGlobalToSeven();
```

Beispiel: Modul `module.c`

```
1 #include "definitions.h"
2
3 void setGlobalToSeven() {
4     global = 7;
5 }
```

Dieses Dateipaket lässt sich nicht kompilieren. Der Linker (d. h. der `gcc`) gibt folgende Fehlermeldung

aus:

Fehlermeldung des Linkers

```
/tmp/cccuG8Y5.o:(.data+0x0): Mehrfachdefinition von »global«  
/tmp/cc3AiH0L.o:(.data+0x0): hier zuerst definiert collect2: error: ld returned  
1 exit status
```

Wieso ist das so?

Der Header `definitions.h` wird von beiden Modulen (`myProg.c` und `module.c`) benötigt, da die Variable `global` in beiden zur Verfügung stehen soll. Daher enthalten beide Module auch die Zeile `#include "definition.h"`. Der Header-Code wird also in beiden Dateien „eingefügt“. Diese Header-Code enthält aber auch die *Deklaration* von `global`. Effektiv wird die Variable also zweimal deklariert! Der Compiler kann nicht entscheiden, welche der beiden Deklarationen „die Richtige“ ist; daher bricht der Linking-Vorgang ab.

Keine Deklarationen in Headern

Wie bereits erwähnt sollen in Headern nur Definitionen stehen, d. h. `structs`, `unions`, `enums`, `typedefs`, `externs` und Funktions-Prototypen. Bei all diesen Anweisungen werden noch keine Speicher-Stellen reserviert, sondern lediglich „Formen“ definiert.

Da Header den Zweck haben, mehrere Module miteinander zu verbinden, werden sie in mindestens zwei Dateien inkludiert. Eine Deklaration in einem Header wird also naturgemäß zu doppelten Deklarationen führen. Aus diesem Grund schreiben wir in einen Header *ausschließlich* Definitionen. Alle Anweisungen, die die Struktur des Speichers verändern, „leben“ in Modulen.

11.3. Funktionen mit eingeschränkter Sichtbarkeit: `static`

Wir kennen das Schlüsselwort `static` bereits aus dem Kontext von Funktionen: Variablen, die mit diesem Modifier deklariert werden, verbleiben im Speicher, selbst wenn ihr Symbol durch verlassen des Scopes nicht mehr gültig ist. (Siehe Abschnitt 9.2.8.)

Im Kontext Module hat das Schlüsselwort `static` jedoch noch eine zweite Bedeutung:

Selbst, wenn Funktionen nur innerhalb eines Moduls gebraucht werden und nicht im Header „sichtbar gemacht werden“, müssen ihre Namen über das gesamte Projekt hinweg eindeutig sein, dürfen also nicht doppelt vorkommen. Soll diese Regel aufgehoben werden, kann eine Funktion mit dem Schlüsselwort `static` als außerhalb der Modulgrenzen „unsichtbar“ markiert werden.

Betrachten Sie hierzu folgende Dateistruktur¹:

Beispiel: Hauptmodulcode `myProg.c`

```
1  #include <stdio.h>  
2  #include "definitions.h"
```

¹Wie tatsächlich Zufallszahlen generiert werden können, wird im Abschnitt 15.2 besprochen.

```

3 // https://xkcd.com/221/
4 static int getRandomNumber () {
5     return 4; // chosen by a fair dice roll.
6               // guaranteed to be random.
7 }
8
9 int main () {
10     printf("Random Number %d\n", getRandomNumber());
11     printf("Random Event: %s\n", getRandomEvent ());
12 }

```

Beispiel: Header definitions.h

```

1 char * getRandomEvent ();

```

Beispiel: Modul module.c

```

1 #include <stdlib.h>
2
3 static int getRandomNumber() {return rand() % 3;}
4
5 char * getRandomEvent () {
6     switch (getRandomNumber()) {
7         case 0 :
8             return "Zeppelin!"; // https://xkcd.com/73/
9         case 1 :
10            return "Wild Ratata!";
11         case 2 :
12            return "Tree-Powers, activate!";
13     }
14     return "The inexorable passage of time";
15 }

```

Hier haben sowohl myProg.c als auch module.c eine Funktion mit Namen `getRandomNumber`. Da aber beide als `static` ausgezeichnet sind, „kommen sie sich nicht in die Quere“. Wird bei nur einer der beiden das Schlüsselwort `static` entfernt, ergeben sich ebenfalls noch keine Probleme: Die Situation hier ist grob vergleichbar mit zwei sich umschließenden Scopes. Wenn aber beide Instanzen von `getRandomNumber` ohne die Auszeichnung `static` im Code auftauchen, meldet der Linker eine doppelte Definition:

Fehlermeldung des Linkers: Mehrfachdefinition von `getRandomNumber`

```

/tmp/ccXCopXE.o: In Funktion >getRandomNumber<:
module.c:(.text+0x0): Mehrfachdefinition von >getRandomNumber<
/tmp/cc3SiANo.o:myProgram.c:(.text+0x0): hier zuerst definiert
collect2: error: ld returned 1 exit status

```

11.4. Weitere Speicherklassen und Modifier

Die folgenden Typen-Modifier sind hier nur der Vollständigkeit halber aufgeführt und können z. T. nur mit vertieften Kenntnissen über Prozessorarchitektur wirklich verstanden werden.

- auto** Die Speicherklasse **auto** ist die Standard-Speicherklasse und wird überall dort verwendet, wo nicht explizit **static** oder **extern** gesetzt wird.
- register** wird für Werte, die sehr häufig gelesen oder geändert werden und die daher aus Performance-Gründen „im Prozessorspeicher“ behalten werden sollten.
- volatile** zeichnet Variablen aus, die nicht nur vom laufenden Programm geändert werden können, sondern beispielsweise den Zustand eines laufenden Geräts widerspiegeln. **volatile** teilt dem Compiler mit, dass die Variable nicht in den Registern des Prozessors gehalten werden soll, sondern immer ein Zugriff auf die Speicherstelle erfolgen muss (**volatile** entspricht in etwa dem Gegenteil von **register**).
- const** markiert Werte, die nach der ersten Zuweisung nicht mehr geändert werden dürfen. Zum einen überprüft der Compiler, ob diese Regel eingehalten wird, bietet also eine Sicherheitsprüfung des Codes gegen unbemerkte Fehler. Zum anderen erlaubt das Wissen, dass sich ein Variablen-Wert nicht ändert dem Compiler zusätzliche Optimierungsmöglichkeiten.
- restrict** wird im Kontext von Pointern benutzt, und ist Ihr „Versprechen“ an den Compiler, dass der entsprechende Speicherbereich nicht auch über einen anderen Pointer adressierbar ist (d.h. entsprechende Speicherbereiche „überlappen“ garantiert nicht). Dies gibt dem Compiler diverse Möglichkeiten den Code zu optimieren.

12. Anbindung an das Betriebssystem

Operating Systems are like underwear – nobody really wants to look at them

Bill Joy

Während unsere Programme „für sich alleine stehen können“, werden sie in den meisten Fällen auf einem Rechner ausgeführt, der ein Betriebssystem zur Verfügung stellt. Dieses stellt einige Schnittstellen zur Verfügung, die wir hier kennenlernen wollen.

12.1. Errorcodes

Ein Errorcode ist ein Wert zwischen 0 und 255, der angibt, welcher Fehler bei der Ausführung aufgetreten ist. Lief das Programm ohne Fehler ab, so ist dieser Errorcode üblicherweise gleich 0.

12.1.1. Rückgabewert der main

In Kapitel 9 haben wir bereits festgestellt, dass formal die `int main` eine Funktion ist, die einen `int` zurück gibt. Bisher haben wir von dieser Funktionalität keinen Gebrauch gemacht. Ab hier wollen wir aber dieses Feature nutzen, und beenden unsere Programme mit einer `return`-Anweisung:

Beispiel: Hello World mit Rückgabewert

```
1  #include <stdio.h>
2
3  int main() {
4      printf("Hello World!\n");
5      return 0;
6  }
```

Zeile 5 beendet das Programm und gibt einen Wert 0 zurück – aber wo wird dieser Wert entgegen genommen? Wie Sie aus dem Thema dieses Kapitels sicher schon erahnen, nimmt das Betriebssystem den Rückgabewert entgegen. Im Allgemeinen wird dort der Wert ignoriert; er kann aber wiederum von anderen Programmen aufgegriffen werden, die so auf den Ablauf unseres Programms reagieren. Üblicherweise wird ein *Errorcode* zurück gegeben.

Eine Möglichkeit, den Rückgabewert eines Programms abzufragen ist, in der Konsole das Symbol `$?` zu benutzen. Geben Sie die Zeile

Kommandozeile: Rückgabewert eines Programms anzeigen

```
echo $?
```

ein. Sie werden eine Zahl sehen, die eben genau den Rückgabewert des zuletzt gestarteten Programms beschreibt.

Beispiel: Der Linux-Befehl `ls` listet den Inhalt des aktuellen Arbeitsverzeichnisses auf. `progDoesNotExist.c` sei der Name einer Datei, die nicht existiert. Mit diesem Wissen können sie den folgenden Konsolen-Output verstehen:

Beispiel: Ausgabe von Rückgabewerten verschiedener Programme auf der Konsole

```
blue-chameleon@blue-chameleon:~/Codes$ ls
a.out          gtk_helloworld.c  playground.c
BosonScattering ImageLibrary      ProcessFilesVBA.bas
CasseBriques  Java              project
CB            make_play        QS_10-Hartinger_backup.c
'ChiZeta Default CTors.cpp' Mockepon          QS_10-Hartinger.c
CircleAreaApprox.c Mockepon_old_stuff tmp
debug-out.c    module.c          vector2D.c
definitions.h  myProgram         vector2D.h
findroot_sqrt.c myProgram.c       VivadoProjects
freeglut_test.c Playground

blue-chameleon@blue-chameleon:~/Codes$ echo $?
0
blue-chameleon@blue-chameleon:~/Codes$ gcc progDoesNotExist.c
gcc: error: progDoesNotExist.c: Datei oder Verzeichnis nicht gefunden
gcc: fatal error: no input files
compilation terminated.
blue-chameleon@blue-chameleon:~/Codes$ echo $?
1
```

`ls` listet den Inhalt des aktuellen Arbeitsverzeichnisses auf und wird ohne Fehler beendet. Entsprechend wird hier der Errorcode 0 gesetzt. Da die Datei `progDoesNotExist.c` nicht existiert, bricht der gcc seinen Prozess mit Fehler ab; wir erhalten den Error Code 1.

Errorcodes sind teilweise normiert. Das bedeutet, dass die Bedeutung einiger Fehlercodes festgelegt ist; ab einem bestimmten Code sind diese Werte aber frei zu vergeben, d. h. der/die ProgrammiererIn kann selbst entscheiden, was dieser Wert bedeuten soll.

Für die vordefinierten Fehlercodes sind im Header `<errno.h>` bereits Symbole angelegt. Siehe <http://man7.org/linux/man-pages/man3/errno.3.html> für eine Liste aller Symbole mit zugeordneter Bedeutung. Unter http://www-numi.fnal.gov/offline_software/srt_public_context/WebDocs/Errors/unix_system_errors.html finden Sie auch die Zuordnung der Symbole zu den Zahlen.

Selbst definierte Errorcodes sollten größer als 130 sein, um keine Bedeutungskollision mit den vordefinierten Codes zu ergeben.

Allgemeiner Errorcode -1

In den meisten Anwendungsfällen reicht es, zu unterscheiden: *Programm wurde erfolgreich abgeschlossen* und *ein Fehler ist aufgetreten*. Wo diese Unterscheidung reicht, wird als Errorcode entweder 0 bei Erfolg und -1 bei Fehler zurück gegeben. Vom Betriebssystem wird dies automatisch in den Errorcode 255 übersetzt.

Im Kurs *Linux* der Universität Regensburg lernen Sie Möglichkeiten kennen, diese Errorcodes auszuwerten.

12.1.2. Programm instantan beenden: exit

Neben der Möglichkeit, die `main` mit `return errorcode` zu verlassen, ist es auch möglich, den Befehl `exit` aus der `<stdlib.h>` zu benutzen. Dieser beendet die Programmausführung *von jedem beliebigen Punkt aus*, also auch, wenn gerade eine Funktion ausgeführt wird. Der Befehl `exit` nimmt als Argument den Errorcode an, der an das Betriebssystem zurück gegeben werden soll.

Beispiel: Programm beenden mit `exit`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  int main() {
6      double number;
7
8      printf("Bitte geben Sie eine positive Zahl ein:");
9      scanf("%lf", &number);
10
11     if (number < 0) {
12         printf("Fehler: Zahl war kleiner als 0!\n");
13         exit(EXIT_FAILURE); // Fehlercode -1
14     }
15
16     printf("Die Wurzel der Eingabe war %lf.\n", sqrt(number));
17     return EXIT_SUCCESS;    // Fehlercode 0
18 }
```

12.1.3. Befehle an die Kommandozeile: system

Der Befehl `system` aus der `<stdlib.h>` startet ein externes Programm und gibt den Errorcode dieses Aufrufs zurück. Wir können das Beispiel *Ausgabe von Rückgabewerten verschiedener Programme auf der Konsole* also auch direkt aus unserem C-Code heraus laufen lassen:

Beispiel: Aufrufe mit `system` und Auswertung der Errorcodes

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      printf("Inhalt des aktuellen Arbeitsverzeichnisses:");
6      int errCode_ls = system("ls");
7      printf("ls schließt ab mit Error Code %d\n", errCode_ls);
8
9      printf("Versuch der Kompilation von 'progDoesNotExist.c':");
10     int errCode_pdne = system("gcc progDoesNotExist.c");
11     printf("Error Code %d\n", errCode_pdne);
12
13     return 0;
14 }
```

Ausgabebeispiel: Aufrufe mit `system` und Auswertung der Errorcodes

```
a.out          gtk_helloworld.c    playground.c
BosonScattering ImageLibrary      ProcessFilesVBA.bas
CasseBriques   Java              project
CB             make_play         QS_10-Hartinger_backup.c
'ChiZeta Default CTors.cpp' Mockepon  QS_10-Hartinger.c
CircleAreaApprox.c Mockepon_old_stuff tmp
debug-out.c    module.c          vector2D.c
definitions.h  myProgram         vector2D.h
findroot_sqrt.c myProgram.c       VivadoProjects
freeglut_test.c Playground

Inhalt des aktuellen Arbeitsverzeichnisses:ls schließt ab mit Error Code 0
gcc: error: progDoesNotExist.c: Datei oder Verzeichnis nicht gefunden
gcc: fatal error: no input files
compilation terminated.
Versuch der Kompilation von 'progDoesNotExist.c':Error Code 256
```

12.1.4. Fehlerbeschreibung ermitteln: `strerror`

Niemand will sich Fehlercodes im Kopf behalten. Um hilfreiche Unterstützung bei der Fehlersuche zu geben, existiert die Funktion `strerror` aus dem Header `<string.h>`. Dieser wird ein Fehlercode übergeben; der Rückgabewert ist ein Pointer auf einen String, der diesen Fehler menschenlesbar beschreibt:

Beispiel: Errorcodes mit `strerror` beschreiben

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main() {
6      printf("Versuch der Ausführung von 'progDoesNotExist':");
7      int errCode_pdne = system("gcc progDoesNotExist.c");
8      printf("Error Code %d\n", errCode_pdne);
9      printf("Fehlerbeschreibung: %s\n", strerror(errCode_pdne));
10
11     return 0;
12 }
```

12.2. Kommandozeilenparameter

Dem `gcc` können wir über die Kommandozeile mitteilen, welche Codes wir kompilieren wollen. Außerdem ist es möglich, über Kommandozeilenparameter wie `-std=c11` das Verhalten des Compilers weiter zu beeinflussen. Für unsere eigenen Programme gibt es ebenfalls Möglichkeiten, auf solche Kommandozeilenparameter zu reagieren.

Wird ein Programm von der Konsole aus gestartet, so zerlegt das Betriebssystem die komplette Zeile in ihre Bestandteile und kopiert diese in den Arbeitsspeicher¹. Zerlegt wird an denselben Trennzeichen, die

¹Vermutlich sind Sie gewohnt, Programme über ein graphisches Userinterface zu starten, also durch Anklicken. Tatsächlich erzeugen diese Maus-Aktionen „unter der Oberfläche“ ebenfalls Kommandozeilen-Befehle.

auch von `scanf` akzeptiert werden: Leerzeichen und Tabulatoren. Ein Programmaufruf wie

Programmaufruf mit Kommandozeilenparametern

```
./myProg parameter1, parameter2 parameter3 "parameter4 with whitespaces"
```

wird damit zerlegt in:

Zerlegung des Programmaufrufs

```
./myProg  
parameter1,  
parameter2  
parameter3  
parameter4 with whitespaces
```

Wie Sie sehen, bleiben Kommata erhalten; Leerzeichen werden entfernt, es sei denn, sie befinden sich in "doppelten Anführungszeichen".

Diese Zerlegung wird unserem Programm zugänglich gemacht, indem wir die Signatur der `main` abändern:

Syntax: `main` mit Zugriff auf Kommandozeilenparameter

```
int main (int argc, char ** argv) {...}
```

Das Symbol `argc`² (kurz für *argument count*) enthält die Anzahl der „Blöcke“ dieser Zerlegung – im obigen Beispiel also den Wert 5. Über `argv` (kurz für *argument value*) kann auf die einzelnen Blöcke selbst zugegriffen werden. Machen Sie sich hierzu bitte zuerst den Datentyp `char **` klar: Die Zerlegung des Kommandozeilenaufruf ist eine *Liste von Strings*. Ein String wiederum ist eine *Liste von Zeichen*. Damit ist die Zerlegung eine *Liste von Listen von Zeichen* – ein Pointer zweiter Ebene, also ein `char **`. Mit diesen Symbolen können wir nun Code schreiben, und beispielsweise mit `if` und `strcmp` auf bestimmte Parameter reagieren.

Die oben gezeigte Zerlegung wurde mit folgendem Code ausgegeben:

Beispiel: Abfrage von Kommandozeilenparametern

```
1  #include <stdio.h>  
2  
3  int main(int argc, char ** argv) {  
4      printf("argc: %d\n", argc);  
5  
6      for (int i=0; i<argc; i++) {printf("%02d\t%s\n", i, argv[i]);}  
7  
8      return 0;  
9  }
```

²Technisch spricht nichts dagegen, andere Symbole als `argc` und `argv` zu verwenden, solange die Signatur einen `int`- und einen `char **`-Parameter beschreibt. Es hat sich jedoch etabliert, genau diese Namen zu vergeben und sollte im Sinne des Austauschs mit anderen ProgrammiererInnen so beibehalten werden.

Wert von `argv[0]`

Achtung: Der „nullte“ Parameter `argv[0]` existiert immer und enthält den Namen der ausführbaren Datei, die gerade läuft. Ein schneller Weg zu prüfen, ob überhaupt Parameter übergeben wurden ist der Vergleich `argc > 1`.

12.3. Spezielle Befehle für unixoide Konsolen

Die verschiedenen Konsolen-Programme, die auf unixoiden³ Systemen verbreitet sind können bestimmte „Kommandos“ interpretieren, die ihnen mit einem regulären Druck-Befehl mitgeteilt werden. Das bedeutet, dass wir im Format-String von `printf` Zeichenketten setzen können, die besondere Effekte auf der Konsole haben.

12.3.1. Konsolenbildschirm leeren

Um einen „leeren Bildschirm“ anzuzeigen, kann die Sequenz `\033[H\033[J` gedruckt werden.

Beispiel: Abfrage von Kommandozeilenparametern

```
1  #include <stdio.h>
2
3  int main() {
4      char password[20];
5
6      printf("Bitte Passwort eingeben:\n");
7      scanf("%19s", password);
8
9      // Bildschirm leeren, um Passwort nicht unnötig lange anzuzeigen
10     printf("\033[H\033[J");
11
12     return 0;
13 }
```

Anmerkung: Benutzen Sie in der Praxis nie diese Methode, um Passwörter zu verbergen!

Alternative Wege

Sowohl unter unixoiden Systemen als auch unter Windows existiert ein Systemkommando, das denselben Effekt hat. Der Aufruf hierfür ist

```
system("clear");
```

für unixoide Systeme und

```
system("cls");
```

für Windows.

³d. h.. Linux, Mac, BSD, ...

12.3.2. Cursor auf der Konsole platzieren

Bisher konnten wir auf der Konsole nur „von oben nach unten“ und „von links nach rechts“ schreiben. Dies erfordert manchmal einige Klimmzüge, ein bestimmtes Bild der Darstellung zu erzeugen. Leichter wird diese Aufgabe, wenn wir den Cursor an beliebigen Koordinaten platzieren können. Dies ermöglicht die Sequenz `\033[<Zeile>;<Spalte>H`, wobei `<Zeile>` und `<Spalte>` jeweils als Dezimalzahlen angegeben werden.

Beispiel: Sternchen in Tabelle setzen

```
1  #include <stdio.h>
2
3  void print_at(char * text, int row, int col) {
4      printf("\033[%d;%dH", (row), (col));
5      printf("%s", text);
6  }
7
8  int main () {
9      // Tabelle zeichnen
10     for (int row = 0; row < 7; row++) {
11         if (row % 2) {
12             printf("| | | |\n");
13         } else {
14             printf("+--+--+\n");
15         }
16     }
17
18     print_at("*", 4, 4); // Koordinaten der mittleren Zelle
19     print_at("", 8, 1); // Cursorposition auf "Ende" setzen
20
21     return 0;
22 }
```

Ausgabebeispiel: Sternchen in Tabelle setzen

```
+--+--+
| | | |
+--+--+
| |*| |
+--+--+
| | | |
+--+--+
```

12.3.3. Farben setzen

Der Cursor hat eine *Schriftfarbe*, die sich aus einer *Vordergrund-* und einer *Hintergrundfarbe* zusammensetzt. Das bedeutet, dass jedes Zeichen in einer aktuell eingestellten Farbe gedruckt wird. Diese Farbe ist standardmäßig grau auf schwarz; wir können jedoch über bestimmte Sequenzen diese Voreinstellungen ändern. Diese Sequenzen sind im Anhang in Tabelle B.5 aufgelistet. Hinzu kommen einige *Schrifteffekte* wie Fettdruck, Unterstreichung oder Blinken, die jedoch nicht von allen Konsolenprogrammen unterstützt werden. Die entsprechenden Sequenzen finden Sie in Tabelle B.6.

Beispiel: Farbige Textausgabe

```
1  #include <stdio.h>
2
3  int main () {
4      printf("normaler Text\n");
5
6      printf("\x1b[92m"); // Schriftfarbe Hellgrün
7      printf("Hellgrün auf schwarz\n");
8      printf("Immer noch Hellgrün auf Schwarz\n");
9
10     printf("\x1b[44m"); // Hintergrundfarbe Blau
11     printf("Hellgrün auf blau\n");
12
13     return 0;
14 }
```

Ausgabebeispiel: Farbige Textausgabe

```
normaler Text
Hellgrün auf schwarz
Immer noch Hellgrün auf Schwarz
Hellgrün auf blau
```

Nachdem die Konsolenfarbe gesetzt wurde, halten sich *alle* Druck-Befehle an die zuletzt gesetzten Schriftattribute. Das bedeutet, dass insbesondere auch die Sequenz *Konsolenbildschirm leeren* keinen schwarzen Bildschirm erzeugt, sondern mit der gesetzten Hintergrundfarbe den Bildschirm ausfüllt.

Die genannten Farbnamen sind Standardeinstellungen der üblichen Konsolen-Programme, können jedoch in den Einstellungen leicht geändert werden. Gegebenenfalls stellen Sie mit den genannten Sequenzen also andere Farben ein als Sie erwarten.

Farbe setzen unter Windows

Unter Windows existiert der Konsolenbefehl `color`, dem eine zweistellige Hexadezimalzahl als Parameter mitgegeben wird. Die höherwertige Hexadezimalziffer stellt die Hintergrundfarbe ein, während das zweite Zeichen die Vordergrundfarbe bestimmt. Die übliche Zuordnung von Ziffern zu Farben ist:

| | | | | | | | |
|---|---------|---|---------|---|------------|---|---------|
| 0 | Schwarz | 4 | Rot | 8 | Dunkelgrau | C | Hellrot |
| 1 | Blau | 5 | Magenta | 9 | Hellblau | D | Pink |
| 2 | Grün | 6 | Braun | A | Hellgrün | E | Gelb |
| 3 | Cyan | 7 | Grau | B | Helcyan | F | Weiß |

Damit bewirkt zum Beispiel unter Windows die C-Code-Zeile:

```
system("color 0a");
```

dass die Schriftfarbe „Hellgrün auf Schwarz“ eingestellt wurde. Diese Einstellung gilt wie auch unter unixoiden Systemen solange, bis eine neue Farbe eingestellt wird. Weitere Attribute wie Fettdruck sind unter Windows-Konsolen nicht möglich.

13. Dateizugriff

I have files, I have computer files and, you know, files on paper. But most of it is really in my head. So God help me if anything ever happens to my head!

George R. R. Martin

Ein- und Ausgabe fand bisher rein über den Bildschirm und die Tastatur statt. Selbstverständlich können wir aber auch Daten über die Festplatte oder andere Medien austauschen. Hier wollen wir Möglichkeiten kennenlernen, dies umzusetzen.

13.1. File-Handle und Zugriffsmodi

Um Dateien zu bearbeiten muss zunächst klar sein, welche Datei bearbeitet werden soll. Im Weiteren wird auch relevant, welcher Teil der Datei geschrieben oder gelesen werden soll, sowie eine ganze Reihe weiterer Informationen, die tief in die Organisation des Betriebssystems verwickelt sind. Für das Programmieren in C sind alle diese Informationen in einer `struct FILE` verpackt, die wir im Detail nicht kennen müssen¹.

Dateizugriff kann in verschiedenen *Modi* stattfinden. Modi sind in diesem Fall *Lese-* oder *Schreibzugriff* (sowie einige Unterarten davon, die wir gleich im Detail kennenlernen werden). Wir können also nicht im selben Schritt aus einer Datei lesen und in diese schreiben.

Wenn wir eine Datei bearbeiten wollen, erfragen wir zunächst beim Betriebssystem Zugriff auf diese Datei. Dies geschieht mit der Funktion `fopen`, welche im Header `<stdio.h>` deklariert wird. `fopen` erwartet als Parameter einen Dateinamen und eine Modusbezeichnung; zurück gegeben wird ein Pointer auf eine Instanz von `struct FILE`. Diesen Pointer bezeichnen wir im Weiteren als *File-Handle*².

Syntax: `fopen`

```
FILE * handle = fopen(filename, mode);
```

Dabei bedeuten:

filename Ein `char *` auf einen beliebigen, gültigen³ Dateinamen. Dieser *String* darf auch Pfadangaben beinhalten⁴. Wird kein Pfad angegeben, so geht das System vom aktuellen Arbeitsverzeichnis aus⁵.

¹Unter <https://elixir.bootlin.com/linux/latest/source/include/linux/fs.h> finden Sie die Definition dieser Datenstruktur; der dort abgedruckte Header ist 3537 Zeilen lang, was die eigentliche Komplexität des Themas *Dateizugriff* zeigt. Wie bereits erwähnt müssen wir zur Anwendung diese Details aber nicht kennen.

²*Handle* ist hierbei bildlich zu verstehen: Es handelt sich um einen „Angriffspunkt“, ein „Griff“, an dem wir die Datei „anfassen“ können.

³Wie Sie sicher wissen, sind bestimmte Zeichen in Dateinamen nicht erlaubt. Solche Zeichen sind beispielsweise `*` oder `?`

⁴Unter unixoiden Systemen ist das Trenn-Symbol für Pfade der *forward slash* `/`; für Windows wird der *backward slash* `\` verwendet

⁵Dies ist im Allgemeinen dasselbe Verzeichnis, in dem auch die ausführbare Datei liegt. Befehle wie `system("cd ..");` können dieses Arbeitsverzeichnis natürlich wechseln.

mode Ebenfalls ein String (also ein **char ***), der angibt, ob die Datei im Lese- oder Schreibmodus geöffnet werden soll. Tabelle 13.1 listet die wichtigsten Modi auf.

| Modus | Wirkung |
|-------|---|
| "r" | Lesezugriff (<i>read</i>) |
| "w" | Schreibzugriff: Datei neu anlegen (<i>write</i>) |
| "a" | Schreibzugriff: Daten an das Dateiende anhängen (<i>append</i>) |

Tabelle 13.1.: Liste der Dateimodi

Beispiel: Datei `file.txt` im Schreibmodus öffnen (1)

```
1  #include <stdio.h>
2
3  int main () {
4      FILE * handle = fopen("file.txt", "w")
5      return 0;
6  }
```

Hier wird die Datei `file.txt` im *Schreibmodus* geöffnet. Existiert die Datei noch nicht, so wird sie im Dateisystem angelegt. Wenn es dagegen schon eine Datei mit dem Namen `file.txt` gibt, so wird sie durch eine zunächst leere Datei ersetzt.

Kann die Datei nicht geöffnet werden (z.B. weil der Dateiname unzulässige Zeichen enthält, oder weil der Dateipfad nicht erreichbar ist), so ist der Rückgabewert von `fopen` gleich 0.

Eine geöffnete Datei muss auch wieder geschlossen werden, und der Speicher für das File-Handle sollte wieder freigegeben werden. Beides geschieht in einem Schritt über den Befehl `fclose` (deklariert im Header `<stdio.h>`), der als Parameter lediglich den File-Handle erwartet:

Syntax: `fclose`

```
fclose(handle);
```

Beispiel: Datei `file.txt` im Schreibmodus öffnen (2)

```
1  #include <stdio.h>
2
3  int main () {
4      FILE * handle = fopen("file.txt", "w")
5      fclose(handle);
6      return 0;
7  }
```

Wenn Sie im obigen Beispiel in Zeile 4 den Modus `"w"` durch ein `"a"` ersetzen, wird eine bereits existierende Datei `file.txt` geöffnet und Daten können an das Dateiende angehängt werden. Sollte die Datei `file.txt` allerdings noch nicht existieren so wird diese neu erstellt und geöffnet.

Eine Datei kann nur dann im Lesemodus geöffnet werden, wenn diese auch tatsächlich existiert. Ersetzen wir in Zeile 4 also das `"w"` durch ein `"r"`, so wird nur dann ein `struct FILE` angelegt, wenn das Öffnen erfolgreich war. Andernfalls ist der Rückgabewert von `fopen` gleich 0. Im Lesemodus wird keine Datei angelegt.

Prüfen ob Datei bereits vorhanden

Es gibt Situationen, in denen Sie nur prüfen wollen, ob eine Datei mit einem gegebenen Namen überhaupt existiert. Dies können Sie über den Rückgabewert von `fopen` leicht prüfen:

Beispiel: Prüfen ob Datei bereits vorhanden

```
1  #include <stdio.h>
2
3  int fileExists(char * filename) {
4      FILE * handle = fopen(filename, "r");
5      if (handle) {fclose(file); return 1;}
6      else      {          return 0;}
7  }
8
9  int main () {
10     FILE * handle = NULL;
11     if (fileExists("myFile.txt")) {
12         // Dialog anzeigen: Datei überschreiben?
13     } else {
14         handle = fopen("myFile.txt", "w");
15     }
16
17     if (handle) {fclose(handle);}
18     return 0;
19 }
```

Daten nicht versehentlich überschreiben

Öffnen Sie eine Datei die im Modus "`w`", so wird entweder eine leere Datei angelegt, oder eine bestehende Datei durch eine leere Datei ersetzt. Wollen Sie versehentliches Überschreiben verhindern, so können Sie als Modus auch "`wx`" angeben: dies führt automatisch eine Prüfung durch, ob eine Datei mit gegebenem Namen bereits existiert und bricht den Vorgang ab, wenn dies der Fall ist. Als Handle wird in diesem Fall `NULL` zurück gegeben:

Beispiel: Datei im Schreibmodus öffnen ohne versehentliches Überschreiben

```
1  #include <stdio.h>
2
3  int main () {
4      FILE * handle = fopen("myFile.txt", "wx");
5      if (handle) {
6          // Datei schreiben...
7      } else {
8          printf("Fehler: Datei existiert bereits!\n");
9          return -1;
10     }
11
12     if (handle) {fclose(handle);}
13     return 0;
14 }
```

13.2. Text-Zugriff – Lesen und Schreiben in menschlichem Format

In die mit `fopen` geöffnete Datei können wir nun mit `fprintf` schreiben, ganz so wie wir es schon von `printf` gewohnt sind. Als zusätzlichen, ersten Parameter müssen wir angeben, in welche Datei geschrieben werden soll: Dies geschieht selbstverständlich über das File-Handle:

Syntax: `fprintf`

```
fprintf(handle, formatString, ...)
```

Beispiel: Ausgabe in eine Datei

```
1  #include <stdio.h>
2
3  int main () {
4      FILE * handle = fopen("myFile.txt", "w");
5
6      fprintf(handle, "hello world!\n");
7      fprintf(handle, "29 * 3 = %d\n", 29 * 3);
8
9      fclose(handle);
10     return 0;
11 }
```

Dieses Codebeispiel sorgt dafür, dass nach Programmablauf im Arbeitsverzeichnis eine Datei mit Namen `myFile.txt` vorliegt. Der Inhalt dieser Datei lautet:

Inhalt der Datei `myFile.txt` nach Ablauf des Beispiels: *Ausgabe in eine Datei*

```
hello world!
29 * 3 = 87
```

(Der letzte Zeilenumbruch im obigen Abdruck ist kein Versehen – die Datei hat drei Zeilen, wobei die letzte eine „leere Zeile“ ist.)

Dieselbe Datei `myFile.txt` kann nun mit dem Befehl `fscanf` wieder gelesen werden. Ähnlich wie `fprintf` ist dieser Befehl ein Analogon zu `scanf` für Dateien: In beiden Fällen steht das erste `f` für *file*.

Syntax: `fscanf`

```
fscanf(handle, formatString, pointer1, ...);
```

Aus einer zuvor geöffneten Datei mit `handle` lesen wir also nacheinander Werte, wie sie ein `formatString` vorgibt. Dabei können wir uns vorstellen, wie ein Cursor in der Datei langsam nach vorne wandert, und für jedes `%` im `formatString` einen Informationsblock weiter springt. Die gelesenen Daten werden dann wie bei `scanf` an jenen Speicheradressen abgelegt, auf welche die `pointer` zeigen.

Während die Syntax für uns inzwischen leicht zu verstehen ist, gestaltet sich die Anwendung manchmal schwierig, wie die folgenden Beispiele verdeutlichen:

Beispiel: Datei myFile.txt zurück lesen

```
1  #include <stdio.h>
2
3  int main () {
4      FILE * handle = fopen("myFile.txt", "r");
5
6      if (!handle) {
7          printf("Fehler: Konnte Datei 'myFile.txt' nicht öffnen.");
8          return -1;
9      }
10
11     char    helloWorldLine[1024],
12            asterisk      [2],
13            equals        [2];
14     int     twentynine = -1, three = -1, eightyseven = -1;
15
16     fscanf(handle, "%1023[^\n]", helloWorldLine);
17     fscanf(handle, "%d"          , &twentynine);
18     fscanf(handle, "%1s"         , asterisk);
19     fscanf(handle, "%d"          , &three);
20     fscanf(handle, "%1s"         , equals);
21     fscanf(handle, "%d"          , &eightyseven);
22
23     printf("%s\n", helloWorldLine);
24     printf("%d %s %d %s %d\n",
25           twentynine, asterisk, three, equals, eightyseven
26     );
27
28     fclose(handle);
29     return 0;
30 }
```

Ausgabebeispiel: Datei myFile.txt zurück lesen

```
hello world!
29 * 3 = 87
```

Wie bekannt öffnen wir in Zeile 4 unsere Datei und überprüfen anschließend (Zeile 6) ob das erfolgreich war.

In den Zeilen 10 bis 13 deklarieren wir einige Variablen, in denen wir den Dateiinhalt speichern wollen. `helloWorldLine` ist sehr groß gewählt, um mit Sicherheit eine ganze Zeile speichern zu können. `asterisk` und `equals` enthalten jeweils ein Zeichen und einen Null-char. Die anderen Variablen werden schließlich die Zahl aufnehmen, nach der sie benannt sind.

In Zeile 15 lesen wir die erste Zeile der Datei ein. Der Format-String `%1023[^\n]` übersetzt sich zu „maximal 1023 Zeichen, die alles außer ein Zeilenumbruch sein dürfen“. Wie immer wird bei String-Lesebefehlen ein Null-char mit geschrieben, daher dürfen nur 1023 Zeichen gelesen werden, auch wenn `helloWorldLine` 1024 Zeichen aufnehmen kann. Beim Lesen der ersten Zeile wird durch den Formatstring der Zeilenumbruch nicht mit eingelesen; der Cursor steht also noch *vor* dem Zeilenumbruch.

In der folgenden Zeile 16 soll nun eine Ganzzahl gelesen werden (Formatstring `%d`). Daher überspringt `fscanf` alle Nicht-Zahlen-Zeichen. In unserem Fall ist das also der Zeilenumbruch. Danach trifft der

Dateicursor auf den Text 29, der korrekt als Zahl eingelesen wird.

Als nächstes soll das Sternchen `*` gelesen werden. Zeile 17 liest genau dieses eine Zeichen ein, und speichert es unter der Adresse von `asterisk`. Wir benutzen dazu den Format-String `%1s`, der eben einen String mit einer Länge von einem Zeichen beschreibt, wobei Zeilenumbrüche, Leerzeichen und Tabulatoren ignoriert werden.

Die Idee liegt nahe, nur ein `char` lesen zu wollen, da ja nur ein einzelnes Sternchen gelesen werden soll. Hierbei würde man aber vergessen, dass vor diesem Zeichen noch ein Leerzeichen steht. Ersetzen wir Zeile 17 durch `fscanf(handle, "%c", asterisk);`, so enthält `*asterisk` jetzt den Wert 32, also den ASCII-Code eines Leerzeichens. Der Dateicursor ist nur um eine Stelle weiter gewandert, und steht weiterhin *vor* dem Sternchen, weswegen hier auch das Einlesen der nächsten Zahl in Zeile 18 fehlschlägt.

In den folgenden Zeilen 18-20 wird dieses Pattern wiederholt. Schließlich erfolgt in den Zeilen 22-25 die Ausgabe der eingelesenen Werte mit `printf`, bevor Zeile 27 die Datei schließt und Zeile 28 das Programm beendet.

13.3. Binär-Zugriff – Lesen und Schreiben in Binärformat

Der Titel des vorherigen Abschnitts war *Lesen und Schreiben in menschlichem Format*. Alle Werte, die in die Datei geschrieben wurden, werden von jedem Texteditor so dargestellt, dass ein Mensch diese interpretieren kann. Sie sind inzwischen aber mit der Tatsache vertraut, dass die interne Speicherdarstellung von diesem Menschenlesbaren Format stark abweicht.

Es ist möglich, Werte so in Dateien abzulegen, wie sie auch im Speicher stehen. Dies hat zwei Vorteile:

- Da weder eine Umrechnung vom menschenlesbaren Format in die interne Speicherstruktur stattfinden muss, noch auf Trennzeichen wie Zeilenumbrüche acht gegeben werden muss, funktioniert das Einlesen solcher *Binärdaten* bedeutend schneller. Für große Datensätze (z. B. Bilder, wissenschaftliche Messreihen, Musikdaten, ...) bedeutet das erheblichen Performance-Gewinn.
- Im Allgemeinen ist die Binäre Darstellung von Zahlen kompakter. Die Zahl 12345678 braucht als menschenlesbarer Text 8 Zeichen und damit auch 8 Bytes. Hinzu kommt noch mindestens ein Byte für ein Trennzeichen. Ein `long`-Wert dagegen ist immer 4 Byte lang, und kann daher ohne Trennzeichen abgelegt werden. Im Binärformat gespeicherte Dateien benötigen also im Allgemeinen weniger Speicherplatz.

Zu diesem Zweck existieren die Befehle `fread` und `fwrite`, beide deklariert im Header `<stdio.h>`⁶.

Da die binäre Darstellung besonders für große Datenmengen günstig ist, sind die Befehle `fread` und `fwrite` auf die Benutzung mit Arrays ausgelegt. Die Syntax der beiden Befehle ist sehr ähnlich:

Syntax: `fread` und `fwrite`

```
fread (pointer, size, count, handle);  
fwrite(pointer, size, count, handle);
```

Dabei stehen die Parameter für folgende Konzepte:

pointer Die Speicheradresse, an der die gelesenen Werte abgelegt werden sollen (`fread`) bzw. von der die Daten stammen, die in die Datei geschrieben werden sollen (`fwrite`).

size Die Größe eines zu lesenden/schreibenden Elements in Bytes.

⁶TODO: Endianness

count Die Anzahl der Elemente, die gelesen/geschrieben werden sollen.

handle Ein File-Handle, wie er von **fopen** zurückgegeben wurde.

Betrachten wir dazu das folgende Beispiel:

Beispiel: fwrite und fread

```

1  #include <stdio.h>
2
3  int main () {
4      FILE * handle;
5      handle = fopen("myFile.dat", "w");
6      if (!handle) {
7          printf("Fehler: Konnte Datei 'myFile.dat' nicht öffnen.");
8          return -1;
9      }
10
11     unsigned int outArray[26];
12     unsigned int N = sizeof(outArray) / sizeof(*outArray);
13
14     for (int i = 0; i < N; i++) {
15         // einige große Zahlen erzeugen
16         outArray[i] = 'A' + i          // = 65 + i
17                     + (('A' + i) << 8) // = (65 + i) * 256
18                     + (('A' + i) << 16) // = (65 + i) * 65536
19                     + (('A' + i) << 24); // = (65 + i) * 16777216
20     }
21     fwrite(outArray, N, sizeof(*outArray), handle);
22
23     fclose(handle);
24     // ..... //
25     handle = fopen("myFile.dat", "r");
26     if (!handle) {
27         printf("Fehler: Konnte Datei 'myFile.dat' nicht öffnen.");
28         return -1;
29     }
30
31     unsigned int inArray[N+1] = {0};
32     fread (inArray, N, sizeof(* inArray), handle);
33
34     for (int i = 0; i < N; i++) {
35         printf("%2d\t%u\t%u\t%s\n",
36             i,
37             inArray[i] , outArray[i],
38             inArray[i] == outArray[i] ? "gleich" : "ungleich"
39         );
40     }
41
42     printf("\nDarstellung derselben Daten in der Datei:\n");
43     printf("%s\n", (char *)inArray);
44
45     fclose(handle);
46     return 0;
47 }

```

Wie gewohnt öffnen wir in Zeile 6 eine Datei im Schreibmodus und überprüfen anschließend, ob dies erfolgreich war. Zeile 12 deklariert ein Array `outArray` aus 26 positiven Ganzzahlen, die wir später in die Datei `myFile.dat` schreiben werden. Die Variable `N` enthält die Anzahl der Elemente von `outArray`.

(Obwohl wir wissen, dass dieses Array 26 Elemente hat, bietet es sich manchmal an, die Arraygröße explizit zu berechnen. Wenn wir zu späterer Zeit unseren Code anpassen und die Arraygröße ändern, müssen wir nicht den ganzen Code überarbeiten.)

Wir befüllen `outArray` in den Zeilen 15 bis 21 mit einigen großen Zahlen. Für dieses Beispiel sind diese Zahlen so konstruiert, dass ihre Speicherdarstellung dieselbe Form hat wie ein String aus den Buchstaben A-Z. Vorerst ist es aber egal, welche Zahlen in diesem Array stehen.

Zeile 23 schreibt dann dieses Array *als Ganzes* in die Datei `myFile.dat`. Wie bereits erwähnt werden die Daten in *binärer Form* geschrieben, d. h. ein Mensch wird in dieser Datei keine Zahlen erkennen, wenn diese mit einem Texteditor geöffnet wird. Entsprechend können wir den Inhalt auch nicht mit `fscanf` zurück lesen, sondern müssen mit `fread` arbeiten.

Um die Daten zurück zu lesen, muss die Datei `myFile.dat` zuerst geschlossen werden, da wir sie bis dato im *Schreibmodus* geöffnet hatten. Wir setzen daher also in Zeile 25 ein `fclose` und öffnen die Datei erneut in Zeile 29, diesmal im Lesemodus.

Zeile 35 deklariert ein neues Array `inArray`, das wir nun um ein Element größer wählen, als es für unsere Datei nötig wäre. Dies dient hier dem Zweck, die eingelesenen Daten auch als String ausgeben zu können – siehe dazu gleich mehr.

Ebenso, wie wir ein gesamtes Array in einem einzigen Befehl schreiben konnten, ist es in Zeile 36 auch möglich, eine gesamte Liste mit einem einzigen `fread` zu lesen. Die ersten 26 Elemente des Arrays enthalten jetzt den Dateiinhalt; das „überzählige“ 27. Element wird nicht überschrieben und behält seinen Startwert 0.

Die Zeilen 38-44 geben sowohl `inArray` als auch `outArray` aus, sowie einen Vergleich der gelesenen und geschriebenen Werte. Wie zu erwarten sind alle Array-Elemente von `inArray` und `outArray` gleich.

In Zeile 47 schließlich geben wir nochmals den Inhalt von `inArray` auf dem Bildschirm aus, interpretieren ihn hier jedoch als Zeichenkette. Dies ist auch der Grund, warum wir ein überzähliges Element für das Feld angelegt haben: das letzte Element des Arrays hat den Wert 0 und markiert so das Ende eines Strings. Diesen String werden Sie auch sehen, wenn Sie die Datei `myFile.dat` mit einem Texteditor öffnen!

Alle Daten, mit denen Sie arbeiten, können auf verschiedene Art und Weise interpretiert werden. Obwohl Ihre Arrays `inArray` und `outArray` positive Ganzzahlen enthalten, werden diese im Speicher wie alles als Bitmuster behalten, die in Dateien wie ein Text aussehen können!

Ausgabebeispiel: `fwrite` und `fread`

| | | | |
|----|------------|------------|--------|
| 0 | 1094795585 | 1094795585 | gleich |
| 1 | 1111638594 | 1111638594 | gleich |
| 2 | 1128481603 | 1128481603 | gleich |
| 3 | 1145324612 | 1145324612 | gleich |
| 4 | 1162167621 | 1162167621 | gleich |
| 5 | 1179010630 | 1179010630 | gleich |
| 6 | 1195853639 | 1195853639 | gleich |
| 7 | 1212696648 | 1212696648 | gleich |
| 8 | 1229539657 | 1229539657 | gleich |
| 9 | 1246382666 | 1246382666 | gleich |
| 10 | 1263225675 | 1263225675 | gleich |

| | | | |
|----|------------|------------|--------|
| 11 | 1280068684 | 1280068684 | gleich |
| 12 | 1296911693 | 1296911693 | gleich |
| 13 | 1313754702 | 1313754702 | gleich |
| 14 | 1330597711 | 1330597711 | gleich |
| 15 | 1347440720 | 1347440720 | gleich |
| 16 | 1364283729 | 1364283729 | gleich |
| 17 | 1381126738 | 1381126738 | gleich |
| 18 | 1397969747 | 1397969747 | gleich |
| 19 | 1414812756 | 1414812756 | gleich |
| 20 | 1431655765 | 1431655765 | gleich |
| 21 | 1448498774 | 1448498774 | gleich |
| 22 | 1465341783 | 1465341783 | gleich |
| 23 | 1482184792 | 1482184792 | gleich |
| 24 | 1499027801 | 1499027801 | gleich |
| 25 | 1515870810 | 1515870810 | gleich |

Darstellung derselben Daten in der Datei:

```
AAAABBBBCCCCDDDEEEEEFFFFGGGGHHHHIIIIJJJKKKKLLLLMMMMNNNNOOOOPPPPPQQQRRRRSSSSTTT
UUUUUVVVVWWWWWXXXXYYYYZZZZ
```

13.4. Cursorposition in Dateien: `fseek` und `ftell`

Nach dem Öffnen einer Datei befindet sich der Cursor entweder am Dateianfang (Modi `r` und `w`) oder am Dateiende (Modus `a`). Nachfolgende Lese- und Schreibbefehle bewegen den Cursor. Wir können aber auch die Cursorposition frei wählen, ohne erst elementweise durch die Datei zu springen. Dazu dient der Befehl `fseek`.

Syntax: `fseek`

```
fseek(filehandle, offset, origin);
```

Dabei bedeuten:

filehandle Das Handle, wie wir es von `fopen` erhalten haben.

offset Die Zahl von Bytes (d.h. Schriftzeichen), um die der Cursor verschoben werden soll.

origin Eine Codeziffer, die angibt, von wo weg gezählt werden soll. Mögliche Werte sind:

SEEK_SET Verschiebung relativ zum Dateianfang. (**offset** ist dann die Nummer des Bytes in der Datei.)

SEEK_CUR Verschiebung relativ zur aktuellen Cursorposition

SEEK_END Verschiebung relativ zum Dateiende.

Analog dazu existiert der Befehl `ftell`, der die aktuelle Cursorposition in der Datei zurück gibt:

Syntax: `ftell`

```
long int position = ftell(filehandle);
```

Mit diesen beiden Befehlen können wir beispielsweise die Größe einer Datei in Bytes ermitteln:

Beispiel: Länge einer Datei mit `fseek` und `ftell` ermitteln

```
1  #include <stdio.h>
2
3  long int lenght_of_file (FILE * filehandle) {
4      long int oldPos = ftell(filehandle);
5
6      fseek(filehandle, 0L, SEEK_END);
7      long int endPos = ftell(filehandle);
8
9      fseek(filehandle, oldPos, SEEK_SET);
10     return endPos;
11 }
12
13 int main () {
14     FILE * filehandle = fopen("file.txt", "r");
15     printf(
16         "Die Datei 'file.txt' ist %ld Bytes groß.\n",
17         lenght_of_file(filehandle)
18     );
19     fclose(filehandle);
20 }
```

Wir speichern zuerst die aktuelle Position in der Datei (Zeile 4). Danach springen wir „0 Bytes vor das Dateiene“ (Zeile 6). Wir ermitteln nun die Position, die ja auch zugleich die Dateilänge angibt (Zeile 7). Schließlich springen wir an den Ausgangspunkt in der Datei zurück, also an die Position, die wir zu Anfang gespeichert hatten (Zeile 9).

Schließlich gibt es noch einen schnellen Weg, zu ermitteln, ob wir bereits das Dateiene erreicht haben: Die Funktion `feof` (deklariert im Header `<stdio.h>`; steht für *file operation: End of File*) nimmt als Argument ein File-Handle, und gibt 1 zurück, sobald das Dateiene erreicht wurde:

Beispiel: Datei bis ans Ende lesen mit `feof`

```
1  #include <stdio.h>
2
3  int main () {
4      FILE * filehandle = fopen("file.txt", "r");
5
6      char line[1024];
7      while (!feof(filehandle)) {
8          fscanf(filehandle, "%1023[^\n]", line);
9          printf("%s\n", line);
10     }
11
12     fclose(filehandle);
13 }
```

14. Der Präprozessor

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are—by definition—not smart enough to debug it.

Brian Kernighan

Bisher haben wir uns damit beschäftigt, wie Code in Maschinsprache übersetzt wird. Der C-Standard sieht aber auch ein Konzept vor, das noch vor dieser Übersetzung angreift, und Ersetzungen auf Code-Ebene durchführt. Dieses Konzept wird vom *Präprozessor* behandelt, der im `gcc` integriert ist. Wir können uns vorstellen, dass unser Code durch Copy/Paste-Anweisungen verändert wird, bevor das eigentliche Kompilieren stattfindet. Zu diesem Zweck steht ein eigens (sehr kleines) Set an *Präprozessor-Direktiven* zur Verfügung, die alle mit einer Raute `#` beginnen.

Tatsächlich haben Sie schon im ersten Kapitel die erste solche Präprozessor-Direktive kennen gelernt: `#include` verändert Ihren Quellcode, bevor das Kompilieren beginnt, indem eine Header-Datei an die Stelle kopiert wird, an der in Ihrem Original-Code `#include` steht.

Kommandozeilenoption `-E`: Nur Präprozessor

Um zu sehen, welche Ersetzungen der Präprozessor macht, können Sie mit dem folgenden Kommandozeilenbefehl den „tatsächlichen“ C-Code sehen, der kompiliert wird:

```
gcc -E ./myProgram.c
```

Die Ausgabe dieser sogenannten *Code-Expansion* erfolgt direkt auf dem Bildschirm. Üblicherweise enthält Ihr Code `#include`-Zeilen, die in diesem Schritt durch den Dateiinhalt ersetzt werden. Da die Ausgabe dadurch sehr lang werden kann, ist es ggf. günstiger, die Ausgabe in eine Datei umzuleiten. Dies können Sie mit der folgenden Zeile bewerkstelligen:

```
gcc -E ./myProgram.c > ./myPreprocessedCode.c
```

Die Datei `myPreprocessedCode.c` enthält dann Ihren Code, mit sämtlichen Ersetzungen, die durch die Präprozessor-Direktiven vorgegeben wurden.

Trend: Abwendung von Präprozessor-Lösungen

Der Präprozessor ist fester Bestandteil des C/C++-Standards. Jedoch bringt dieses Element eine Reihe von Nachteilen mit sich, wegen derer inzwischen von der Verwendung des Präprozessors weitgehend abgeraten wird. Um mit Library-Lösungen umzugehen müssen Sie die Funktionsweise des Präprozessors kennen; wo möglich sollten Sie aber auf die bereits bekannten Mittel zurück greifen.

Selbstverständlich sind nicht alle Einsatzgebiete des Präprozessors ersetzbar. Der Befehl `#include` beispielsweise hat keine Entsprechung und wird in den Sprachen C und C++ immer von Bedeutung bleiben. Wo die hier gezeigten Techniken immer noch sinnvoll sind, wird besonders darauf hingewiesen.

14.1. Konstante Ausdrücke und vordefinierte Symbole

14.1.1. Konstante Ausdrücke

Der Befehl `#define` wird dazu verwendet, einfache Ersetzungen zu machen. Die Syntax lautet:

Syntax: `#define`

```
#define SYMBOL Ersetzung
```

Dabei ist `SYMBOL` eine beliebige Folge von alphanumerischen Zeichen inklusive des Unterstrichs `_`, die bis zu 40 Zeichen umfassen darf. Nach dem ersten Leerzeichen rechnet der Compiler alle folgenden Zeichen bis zum Zeilenende als Teil von `Ersetzung`. Alles, was hier steht, wird vom Präprozessor anstelle von `Symbol` eingefügt.

`#define`-Symbole werden auch **Macros** genannt.

Sie können `#define` beispielsweise benutzen, um Naturkonstanten zu definieren. Dies ist immer noch gängige Praxis für das Programmieren in C:

Beispiel: `#define` für Naturkonstanten

```
#define PI 3.141592653589793238462643383279502884197169399375
```

Konvention: Konstante Ausdrücke in Großbuchstaben

Konstanten verhalten sich anders als Variablen, sind aber im Code nicht mehr auf Anhieb von diesen zu unterscheiden. Es hat sich daher durchgesetzt, Macros in Großbuchstaben zu setzen.

Häufig wird `#define` auch eingesetzt, um wiederkehrenden Werten ein klareres Symbol zu geben. Dies ermöglicht es, mit einer einzigen Änderung im Code das gesamte Verhalten des Programms zu ändern und sorgt i. d. R. für besser lesbaren Code. Beispielsweise können Sie für eine wissenschaftliche Simulation ein Symbol `SYSTEMGROESSE` definieren und mehrfach schnell hintereinander Ergebnisse für verschiedene Systeme erhalten, ohne Ihren gesamten Code durchsuchen zu müssen.

Der Text in `Ersetzung` darf mehr als ein einzelnes „Wort“ sein, d. h. darf auch Leerzeichen enthalten. Es ist möglich, damit ganze „Mini-Befehle“ zu schreiben:

Beispiel: Komplexer Ausdruck mit #define

```
1  #include <stdio.h>
2
3  #define COUNT_TO_TEN for(unsigned int i=1; i<11; i++) {printf("%i ", i);}
4
5  int main () {
6      COUNT_TO_TEN;
7  }
```

Ausgabebeispiel: Komplexer Ausdruck mit #define

```
1 2 3 4 5 6 7 8 9 10
```

Keine Semikolons nach Präprozessor-Direktiven

Nach dem Macro wird die gesamte folgende Zeile als Teil von **Ersetzung** aufgefasst. Ein Abschließendes Leerzeichen ist *nicht* nötig. Setzt man diess trotzdem, wird der Code teilweise richtig umgesetzt, oft führt dies aber zu Problemen. Sehen Sie sich dazu das folgende Beispiel an:

#define mit Semikolon

```
1  #include <stdio.h>
2
3  #define N 10;
4
5  int main () {
6      int x = N;
7
8      printf("%d", x);
9      printf("%d", N);
10 }
```

Beim Versuch, diesen Code zu Kompilieren wirft gcc eine Fehlermeldung:

Compiler-Fehlermeldung

```
myProgram.c: In function 'main':
myProgram.c:3:13: error: expected ')' before ';' token
    #define N 10;
               ^
myProgram.c:9:16: note: in expansion of macro 'N'
    printf("%d", N);
```

Der Fehler wird klar, wenn wir uns die Umsetzung des Macros N (die *Expansion*) in Zeile 9 anzeigen lassen:

Expansion des Symbols N in Zeile 9

```
printf("%d", 10;);
```

Offensichtlich ist das Semikolon hinter der 10 hier zuviel!

In Zeile 6 dagegen gibt dies keine Probleme:

Expansion des Symbols N in Zeile 6

```
x = 10;;
```

Das doppelte Semikolon wird wie eine Leerzeile interpretiert und ignoriert.

Um unerwartete Fehler zu vermeiden, setzen Sie *keine* Semikolons in Präprozessor-Direktiven.

Kein Zuweisungsoperator in der #define-Direktive

Beachten Sie, dass Sie *kein* Gleichheitszeichen = bei der Definition von Macros brauchen. Zeilen wie

```
#define N = 10
```

sind zwar gültiger C-Code, im allgemeinen aber unsinnig. Das Symbol N wird hier durch den gesamten Text „= 10“ ersetzt.

Ausdrücke in Klammern

Manchmal will man zwar einen konstanten Ausdruck in seinem Programm verwenden, der allerdings aus mehreren Komponenten besteht. Beispielsweise könnte SYSTEMGROESSE in einer Simulation die Summe aus RANDSTUECK und KERN sein. Es bietet sich also an, die Konstante als genau diese Summe zu definieren.

In diesem Fall aber müssen Sie immer bedenken, dass das Symbol nicht durch einen Wert, sondern durch seinen Code-Text ersetzt wird. Betrachten Sie folgendes Beispiel^a:

Beispiel: Summen in #define-Symbolen

```
1  #include <stdio.h>
2
3  #define SIX  1 + 5
4  #define NINE 8 + 1
5
6  int main () {
7      printf("SIX multiplied by NINE equals %d.\n", SIX * NINE);
8  }
```

Ausgabebeispiel: Summen in #define-Symbolen

```
SIX multiplied by NINE equals 42.
```

Das Problem wird wieder klar, wenn wir uns die Expansion der Symbole SIX und NINE in Zeile 7 ansehen:

Expansion von Zeile 7

```
printf("SIX multiplied by NINE equals %d.\n", 1 + 5 * 8 + 1);
```

Wie bekannt beachtet der Compiler Punkt vor Strich und berechnet somit einen unerwarteten Wert. Um diesem Problem zu entgehen, sollten Sie also ihre Symbole mit Klammern definieren:

Beispiel: Summen in #define-Symbolen

```
1  #define SIX  (1 + 5)
2  #define NINE (8 + 1)
```

^aFrei nach Douglas Adams.

Sie können die Wirkung von Macros begrenzen. Mit der Direktive `#undef` setzen Sie einen Endpunkt, ab dem ein zuvor definiertes Macro nicht mehr durch anderen Code ersetzt wird.

Beispiel: #define und #undef

```
1 #define PI 3.14159265359
2 // ... normaler C-Code...
3
4 #undef PI
5 // printf("%lf\n", PI); -- Fehler: PI nicht (mehr) definiert
6
7 #define PI 3
8 // ... unnormaler C-Code ...
```

Direktiven mit Variablen

Prinzipiell können Sie in Direktiven auch Variablen aus ihrem Code verwenden. Dies sorgt aber für undurchsichtigen Code und ist generell zu vermeiden. Betrachten Sie das folgende Beispiel:

Beispiel: Direktiven mit Variablen

```
1 #include <stdio.h>
2
3 #define COUNT_AND_TELL i++; printf("Zähler: %d\n", i)
4
5 void foo() {
6     for (int i=0; i<10; i++) {
7         printf("i = %d\n", i);
8         COUNT_AND_TELL;
9     }
10 }
11
12 void bar() {
13     COUNT_AND_TELL;
14 }
15
16 int main () {
17     int i=0;
18
19     foo();
20     COUNT_AND_TELL;
21
22     bar();
23     COUNT_AND_TELL;
24 }
```

Zunächst einmal können Sie dieses Beispiel gar nicht erst kompilieren, da in der Funktion `bar` keine Variable `i` existiert, die durch `COUNT_AND_TELL` benutzt werden könnte. Doch selbst, wenn wir auf die Verwendung von `COUNT_AND_TELL` in Zeile 13 verzichten, erhalten wir ein unerwartetes Verhalten: Die Schleife in `foo` wird nur fünfmal durchlaufen, da `COUNT_AND_TELL` jedes Mal die Zählvariable `i` mit erhöht. Dies ist aus dem Symbolnamen nicht sofort ersichtlich, und liefert so schlecht wartbaren Code.

Verzichten Sie daher auf die Verwendung von Variablen in Macros.

Empfehlung: globale `const`-Variablen

Macros haben keinen explizit zugeordneten Datentyp. Das bedeutet, dass der Compiler keine Prüfung auf die „Sinnhaftigkeit“ der Verwendung unserer Symbole machen kann.

In modernen Lehrbüchern wird daher empfohlen, konstante Ausdrücke als `const`-Variablen zu speichern. Ein weiterer Vorteil ist, dass in diesem Fall wirklich mit einem Wert gerechnet wird; Probleme mit Klammersetzung ergeben sich also nicht mehr. Für manche Ausdrücke kann dies sogar Rechenzeit sparen, da der Ausdruck nur einmal ausgewertet werden muss, nicht jedes Mal, wenn das Symbol benutzt wird.

Im folgenden Beispiel greifen wir wieder den Code mit `SIX * NINE` auf. Zusätzlich ersetzen wir zur Veranschaulichung den Wert 1 durch `cos(0)`, was zu 1 ausgewertet wird.

Beispiel: `const`-Ausdrücke

```
1  #include <stdio.h>
2  #include <math.h>
3
4  const int ONE = cos(0);
5  const int SIX = ONE + 5;
6  const int NINE = 8 + ONE;
7
8  int main () {
9      printf("SIX multiplied by NINE equals %d.\n", SIX * NINE);
10 }
```

Ausgabebeispiel: `const`-Ausdrücke

```
SIX multiplied by NINE equals 54.
```

Zunächst bemerken wir, dass der korrekte Wert berechnet wird, obwohl die Definition von `SIX` und `NINE` ohne Klammern gesetzt wurde. In Zeile 9 wird tatsächlich mit den Werten 6 und 9 gerechnet.

Die Berechnung des Cosinus ist relativ aufwendig und kostet merkbar viel Zeit, wenn dies häufig geschehen soll. Obwohl sowohl in `SIX` als auch in `NINE` der Ausdruck `ONE` vorkommt, der ja gleich `cos(0)` ist, wird diese Berechnung nur ein einziges Mal ausgeführt, gleich wie oft im folgenden Code die Symbole `ONE`, `SIX` oder `NINE` vorkommen: Das *Ergebnis* ist in `ONE` gespeichert.

Das Schlüsselwort `const` sorgt hier lediglich dafür, zu verhindern, dass die Ausdrücke `ONE`, `SIX` oder `NINE` versehentlich überschrieben werden. Zeilen wie

```
ONE = 2;
```

werden vom Compiler als unzulässig erkannt und erzeugen eine Fehlermeldung.

Empfehlung: `inline`-Funktionen

Macros, die für kleine Code-Stücke stehen werden manchmal benutzt, um die Codeeffizienz zu steigern. Wie wir wissen wird bei einem Funktionsaufruf einiger Overhead betrieben. Wird der Code dagegen über ein Macro direkt eingefügt, ersparen wir dem Prozessor diese zusätzlichen Verwaltungsschritte. Wir haben aber bereits einige Nachteile der Methode gesehen.

Besser ist es, solchen Code stattdessen in `inline`-Funktionen auszulagern. Dies wurde bereits in Abschnitt 9.3 gezeigt.

14.1.2. Mehrzeilige Symbole

Macros enden mit einem Zeilenumbruch. Wo sich diese Ausdrücke über mehrere Zeilen erstrecken sollen, muss in der Präprozessordirektive mit einem Backslash \ markiert werden, dass der Ersetzungstext fortgesetzt wird. Im Ersetzungstext wird dieser Backslash dann durch einen Zeilenumbruch ersetzt.

Beispiel: Mehrzeilige Präprozessor-Direktive

```
1  #include <stdio.h>
2
3  #define PRINT_HEADLINE \
4      printf("+-----+\n"); \
5      printf("/ headline      /\n"); \
6      printf("+-----+\n")
7
8  int main () {
9      PRINT_HEADLINE;
10 }
```

Ausgabebeispiel: Mehrzeilige Präprozessor-Direktive

```
+-----+
| headline      |
+-----+
```

14.1.3. Vordefinierte Symbole

Unabhängig davon, welche Header Sie in Ihren Code einbinden, stellt der gcc einige weitere Symbole zur Verfügung, die Ihnen Informationen über den Compilerprozess bieten. Einige Symbole von besonderer Bedeutung sind im Anhang in Tabelle B.10 aufgeführt.

Eine ausführliche Liste finden Sie hier: <https://gcc.gnu.org/onlinedocs/cpp/Predefined-Macros.html>. Weitere Symbole, die nur von manchen C-Compilern unterstützt werden, sind unter http://nadeausoftware.com/articles/2012/01/c_c_tip_how_use_compiler_predefined_macros_detect_operating_system aufgelistet.

Konvention: Unterstrich nicht als erstes Zeichen

Vordefinierte Symbole beginnen i. d. R. mit einem Unterstrich _. Da der Sprachumfang von C und C++ beständig erweitert wird, kann es sein, dass in einigen Jahren neue vordefinierte Symbole hinzugefügt werden. Um zukünftige Namens-Kollisionen zu vermeiden, sollten Sie ihre Macro-Namen nicht mit einem Unterstrich beginnen lassen.

14.2. Bedingte Kompilierung

Die Präprozessor-Direktive `#if...#endif` kann dazu benutzt werden, Code-Passagen nur unter bestimmten Bedingungen zu kompilieren. Alles, was zwischen diesen beiden Direktiven steht, wird vom Compiler nur „gesehen“, wenn eine bestimmte Bedingung erfüllt ist. Wie schon bei `if` ist die Bedingung ein Ausdruck, der in einen Wahrheitswert übersetzt werden kann. Im Ausdruck dürfen jedoch keine „normalen Variablen“ vorkommen, sondern nur Konstanten sowie mit `#define` definierte Symbole

(sowie auch die vom Compiler vordefinierten Symbole). Anders als bei `if` muss die Bedingung nicht in Klammern gesetzt werden, und die geschweiften Klammern entfallen ebenfalls.

Beispiel: Debug-Modus (1)

```
1 // #define DEBUG 1
2 /* Debug-Modus aktivieren durch entfernen des Kommentar-Zeichens Zeile oben */
3
4 #include <stdio.h>
5
6 int main () {
7     #if DEBUG == 1
8         printf("Debug-Modus aktiviert\n");
9     #endif
10
11     // ... normaler Code ...
12 }
```

`#if...#endif` kann noch um `#else` und `#elif`-Blocks erweitert werden. `#else` funktioniert dabei wie schon von `if` bekannt; `#elif` ist eine Kurzform von `else if` und funktioniert wie diese.

Bedingungs-Direktiven werden oft mit dem Operator `defined` eingesetzt. Dieser wird zu `true` ausgewertet, wenn ein Symbol existiert; ansonsten ist das Ergebnis `false`. Das obige Beispiel lässt sich damit auch so formen:

Beispiel: Debug-Modus (2)

```
1 // #define DEBUG
2 /* Debug-Modus aktivieren durch entfernen des Kommentar-Zeichens Zeile oben */
3
4 #include <stdio.h>
5
6 int main () {
7     #if defined(DEBUG)
8         printf("Debug-Modus aktiviert\n");
9     #endif
10
11     // ... normaler Code ...
12 }
```

Eine häufige Anwendung ist die Adaption von Code für bestimmte Betriebssysteme:

Beispiel: Betriebssystem-Spezifische Passagen

```
1 #if defined(__linux__)
2     // Linux-spezifischer Code
3 #elif defined(__APPLE__)
4     // Apple-spezifischer Code
5 #elif defined(__WIN64__) or defined(__WIN32__)
6     // Windows-spezifischer Code
7 #else
8     #error Betriebssystem nicht unterstützt.
9 #endif
```

(Die Direktive `#error` sorgt dafür, dass der Kompiliervorgang abgebrochen wird und der nachfolgende Text als Fehlermeldung ausgegeben wird.)

Für `#if defined(x)` und `#if !defined(x)` existieren die Kurzformen `#ifdef(x)` und `#ifndef(x)`.

#include-Sperre

Schon mittelgroße Projekte benutzen oft viele Module. Damit verbunden ist auch die Notwendigkeit, `#include`-Zeilen zu verwenden. Manchmal ergeben sich aus einer solchen Aufteilung aber auch Gegenseitige Abhängigkeiten: Modul A benötigt Definitionen aus B, während aber Modul B auf Definitionen von A aufbaut. Das bedeutet, dass der Header von A ein `#include "B.h"` benötigt, während der Header von B ein `#include "A.h"` enthalten wird. Dies alleine würde zu einer Endlosschleife im Präprozessor führen (bzw. zu doppelten Definitionen, sobald der von A inkludierte Header B.h wieder auf `#include "A.h"` trifft).

Ein Ausweg aus dieser Situation ist die Einführung von *Include-Sperren*: In der ersten Zeile des Headers wird mit `#if` geprüft, ob ein Sperrsymbol noch nicht definiert wurde. Die zweite Zeile definiert gerade dieses Sperrsymbol mit `#define`. Das `#if` umfasst den kompletten Code des Headers. Wird nun versucht, denselben Header zweimal (oder öfter) in ein Modul zu inkludieren, so wird nur das erste `#include` umgesetzt; alle folgenden solchen direktiven scheitern am `#if` im Header:

Beispiel: #include-Sperre im Header `my_header.h`

```
1  #ifndef(MY_HEADER_H)
2  #define MY_HEADER_H
3
4  // Header-Code
5
6  #endif
```

Diese Technik wird noch heute genutzt und ist auch in C++-Projekten üblich.

14.3. Parametrisierte Macros

Wir haben die Direktive `#define` bereits für einfache Ersetzungen kennen gelernt. Es ist möglich, diese Ersetzungen von einem oder mehreren Parametern abhängig zu machen. Statt immer denselben Ersetzungstext zu erzeugen wird ein Symbol dann durch Code ersetzt, der einen Parameter enthält.

Syntax: `#define` mit Parametern

```
#define SYMBOL(Parameterliste) Ersetzung_mit_Parametern
```

Das folgende Beispiel Definiert ein Macro, das die größere von zwei Zahlen zurück gibt:

Syntax: #define mit Parametern

```
#include <stdio.h>

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

int main () {
    int i = 8 , j = 7;
    double x = 8.0, y = 7.0;

    printf("%d\n" , MAX(i, j));
    printf("%lf\n", MAX(x, y));
}
```

Macros mit komplexeren Argumenten

Während Macros schnelle Lösungen für kleine Probleme erlauben, ist ihre Anwendung wie so häufig eine Gefahrenquelle. Betrachten Sie das folgende Beispiel:

Beispiel: Direktiven mit Variablen

```
1  #include <stdio.h>
2
3  #define MAX(a, b) (a > b ? a : b)
4
5  int main () {
6      int i = 8, j = 7;
7      printf("MAX: %2d\n" , MAX(i++, j++));
8      printf(" i : %2d\n", i);
9      printf(" j : %2d\n", j);
10 }
```

Naiv könnte man annehmen, dass die Ausgabe dieser Codezeilen 7 und 8 beide Male 9 sein sollte. Tatsächlich aber lautet die Ausgabe:

Ausgabebeispiel: Direktiven mit Variablen

```
MAX:  9
 i : 10
 j :  8
```

Um dies zu verstehen betrachten wir die Expansion des Macros:

Expansion von Zeile 7

```
7      printf("%d\n" , (i++ > j++ ? i++ : j++));
```

Sowohl *i* als auch *j* werden miteinander verglichen *und dann* inkrementiert. Da *i* vor der Inkrementierung größer als *j* vor der Inkrementierung ist ($8 > 7$) wird der erste Teil des ternären Operators ausgewertet, also *i++*. Der *Rückgabewert* dieser Operation ist der Wert von *i* vor dem Inkrement; gleichzeitig wird aber der Wert von *i* erhöht.

Sie sehen also wieder, dass in der Expansion von Macros auch Nebenwirkungen in der Anwendung von Parametern stecken können.

Macros in Blockstrukturen

Ein weiteres Beispiel soll Sie auf eine unerwartete Wechselwirkung von Macros mit Blockstrukturen wie `if` aufmerksam machen:

Beispiel: Macro in if-Block

```
1  #include <stdio.h>
2
3  #define SAFE_DIVIDE(result, x, y)    if (y) result = x/y
4
5  int main () {
6      int a = 1, b = 0, x = 0;
7      int something = 1;
8
9      if (something) SAFE_DIVIDE(x, a, b);
10     else printf("Something is not set...\n");
11
12     printf("x = %d\n", x);
13 }
```

Naiverweise würde man davon ausgehen, dass `SAFE_DIVIDE` dafür sorgt, dass die Division `a / b` verhindert wird, wenn `b==0`. Da hier `something == 1` gilt, sollte die `else`-Zeile nicht ausgeführt werden. Stattdessen aber lautet die Ausgabe:

Ausgabebeispiel: Macro in if-Block

```
Something is not set...
x = 0
```

Wir erkennen das Problem wieder, wenn wir uns die Expansion des Macros ansehen:

Expansion: Macro in if-Block

```
9      if (something) if (y) x = a/b;
10     else printf("Something is not set...\n");
```

Die `else`-Zeile wird der Bedingung `y != 0` zugeordnet und gehört nicht mehr zu `something`. An dieser Stelle kann das Problem umgangen werden, indem der Macro-Code in {geschweifte Klammern} gesetzt wird:

```
#define SAFE_DIVIDE(result, x, y)    {if (y) result = x/y;}
```

Besser jedoch ist es, auf Macro-Lösungen dieser Form komplett zu verzichten.

Alternative zu Macros: inline-Funktionen

Wie bereits mehrfach erwähnt sind Funktionen Macros vorzuziehen. Wo es auf Geschwindigkeit ankommt, kann das Schlüsselwort `inline` verwendet werden. Die beiden obigen Beispiele (Maximum und sichere Division) lassen sich folgendermaßen umsetzen:

Beispiel: inline-Funktionen für Maximum und sichere Division

```
1  #include <stdio.h>
2
3  inline double max      (double x, double y);
4      double max      (double x, double y) {return x > y ? x : y;}
5  inline double save_divide(double x, double y);
6      double save_divide(double x, double y) {return y ? x / y : 0;}
7
8  int main () {
9      double a = 1, b = 0, x = -1;
10     int something = 1;
11
12     if (something) {x = save_divide(a, b);}
13     else          {printf("Something is not set...\n");}
14
15     printf("x = %lf\n", x);
16     printf("Maximum : %lf\n", max(a++, b++));
17     printf("(a, b) = (%lf, %lf)\n", a, b);
18 }
```

Ausgabebeispiel: inline-Funktionen für Maximum und sichere Division

```
x = 0.000000
Maximum : 1.000000
(a, b) = (2.000000, 1.000000)
```

14.4. Stringify-Operator

Der Operator `#` verwandelt einen nachfolgenden Ausdruck in einen String, und kann zu Debug-Zwecken genutzt werden:

Beispiel: Stringify-Operator

```
1  #include <stdio.h>
2
3  #define DebugInt(x) printf("%s = %d\n", #x, x)
4
5  int main () {
6      int myInt = 5;
7
8      DebugInt(myInt);
9      DebugInt(myInt + 7);
10 }
```


Ausgabebeispiel: Stringify-Operator

```
myInt = 5
myInt + 7 = 12
```

14.5. Concatenation-Operator

Mit dem Concatenation-Operator (Englisch: *to concatenate*, aneinanderreihen) kann man zwei Code-Elemente zu einem einzigen Ausdruck aneinander hängen. Ein Anwendungsfall erspart etwas Tipparbeit:

Stellen Sie sich vor, Sie wollen ein Menüsystem umsetzen. Jeder Menüeintrag hat einen Titel und eine zugeordnete Aktion. Die Aktion wird von einer Funktion ausgeführt, die wir als Funktionszeiger speichern. Alle Menüeinträge legen wir in einem Array ab. Wir können also den folgenden Code setzen:

Beispiel: Menü-Infrastruktur (1)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct {
5      char * name;
6      void (*function)(void);
7  } command;
8
9  void proc_Beenden() {exit(0);}
10 void proc_Hilfe() {printf("Hilfetext...\n");}
11
12 int main () {
13     command menuItems[] = {
14         {"Beenden", proc_Beenden},
15         {"Hilfe" , proc_Hilfe}
16     };
17 }
```

Hier sind nur zwei Menüeinträge gezeigt. Wir können uns aber auch beliebig große Menüs vorstellen. Wenn jeweils Titel und Funktionsname sich jeweils nur um den Präfix `proc_` unterscheiden, bedeutet das, dass wir viel Code doppelt tippen müssen. Stattdessen können wir mit dem Concatenation-Operator ein Code-Template erstellen:

Beispiel: Menü-Infrastruktur (2)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define COMMAND(X) {#X, proc_ ## X}
```

```

5  typedef struct {
6      char * name;
7      void (*function)(void);
8  } command;
9
10 void proc_Beenden() {exit(0);}
11 void proc_Hilfe() {printf("Hilfetext...\n");}
12
13 int main () {
14     command menuItems[] = {
15         COMMAND(Beenden),
16         COMMAND(Hilfe)
17     };
18 }

```

Hier wird wieder derselbe Code erzeugt. In den Augen mancher ProgrammiererInnen ist die letzte Form mit Macro übersichtlicher. Natürlich wird verborgen, dass eine Verbindung zu `proc_X` besteht, und eine Trennung von Namen des Menüeintrags und Funktionsnamen ist nicht mehr möglich. Außerdem lassen sich leicht Fälle konstruieren, in denen das Macro unbrauchbaren Code erzeugt. Kommt im Macro-Parameter `X` beispielsweise ein Leerzeichen vor, so ist dieses zwar im Menütitel enthalten, gehört aber natürlich nicht zum Namen der aufzurufenden Funktion.

15. Miscelaneous

Real stupidity beats artificial intelligence every time.

Terry Pratchett

An diesem Punkt haben wir bereits die wichtigsten Elemente der Sprache kennen gelernt. Hier möchte ich Ihnen noch einige nützliche Dinge zeigen, die für sich keine geschlossene Einheit mehr bilden, bevor wir in den letzten beiden Kapiteln interessante Anwendungen der erlernten Techniken betrachten werden.

15.1. Zeitpunkte

Für einen Menschen ist ein Zeitpunkt gegeben durch sechs Zahlen: Jahr, Monat, Tag, Stunde, Minute und Sekunde. Diese Unterteilung ist im Alltag zwar nützlich, macht das Rechnen mit Zeitpunkten aber kompliziert. Wie viele Sekunden lagen zwischen 2019-06-21, 20:26:35 (dem Zeitpunkt, an dem diese Zeile geschrieben wurde) und 1989-03-29, 00:03:15 (dem Geburtstag des Autors)? Der Header `<time.h>` definiert Funktionen, die bei solchen Aufgaben hilfreich sind. Lesen Sie hierzu im Detail unter den Links von <https://en.cppreference.com/w/c/chrono>.

15.1.1. UNIX Epoch Time

Es ist leichter, mit zwei Zahlen zu rechnen, als mit zwölf¹. Daher werden Zeitpunkte intern für den Rechner auf eine einzige Zahl herunter gebrochen: Die Zahl der Sekunden, die seit 1970-01-01, 00:00:00 vergangen sind. Dieser genormte Zeitpunkt wird *UNIX Epoch Time* oder auch *die UNIX-Epoche* genannt. Kennt man diese Zeitdifferenz, so kann man die Zeit zwischen zwei Punkten einfach als Differenz berechnen. Natürlich bedeutet es einen gewissen Aufwand, diese Differenz wieder in ein menschenlesbares Format zu bringen; hierzu stehen aber Funktionen bereit, die die komplizierteren Schritte (Einbeziehen von Schaltjahren, Schaltsekunden, Zeitzonen, ...) übernehmen.

Der C-Standard definiert den Datentyp `time_t` für die Arbeit mit Zeitpunkten. Das bedeutet insbesondere, dass die Funktionen, die Sie hier kennen lernen werden, Argumente vom Typ `time_t` annehmen werden. Die Definition von `time_t` legt jedoch nur Eigenschaften fest, nicht, wie der Datentyp exakt umgesetzt werden soll. Üblicherweise (und insbesondere beim `gcc`) handelt es sich um einen Alias für einen `long long int`, also eine *vorzeichenbehaftete* 64bit-Ganzzahl. Auf älteren Systemen kann der Typ aber auch zu einem `long int`, also einer 32bit-Ganzzahl umgesetzt werden.

Beide Zahlentypen können nur Werte einer bestimmten Größe speichern, d. h. es gibt einen spätesten Zeitpunkt, den unixoide Systeme verarbeiten können. Für 64bit-Werte liegt dies in der fernen Zukunft², aber bei 32bit wäre keine Zeit nach 2038-01-19, 03:14:07 darstellbar. Eine neue Epoche zu definieren wirft Kompatibilitätsprobleme auf – woraus soll ein Programm ermitteln, auf welche Epoche sich ein Zeitwert bezieht? Für IT-Spezialisten war dieses *Jahr 2038-Problem* lange Zeit ein großes Problem, bis es durch die flächendeckende Einführung von 64bit-Rechnern umgangen werden konnte.

¹Citation needed

²Genauer: am vierten Dezember des Jahres 292 277 026 596. Quelle: https://ximalas.info/2015/03/10/when-does-the-64-bit-unix-time_t-really-end/

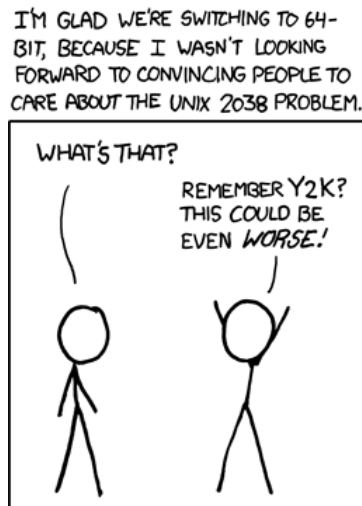


Abbildung 15.1.: Das Jahr 2038-Problem. Quelle: <https://xkcd.com/607/>

15.1.2. Zeitwerte finden und umrechnen

Die aktuelle UNIX-Zeit wird von der Funktion `time` ermittelt. Dieser kann ein Pointer auf ein Objekt vom Typ `time_t` übergeben werden, an den die aktuelle Zeit (also die Zahl der Sekunden seit 1970-01-01, 00:00:00) geschrieben werden soll. Alternativ ist auch der Wert `NULL` als Parameter zulässig – dann wird kein Wert in den Speicher geschrieben. In beiden Fällen enthält der Rückgabewert die Information der UNIX-Zeit.

Beispiel: `time`

```

1  #include <stdio.h>
2  #include <time.h>
3
4  int main () {
5      time_t now = time(NULL);
6
7      // time(&now); // alternative Methode
8
9      printf("%ld Sekunden seit 1970-01-01, 00:00:00.\n", now);
10 }
```

Diese UNIX-Zeit kann nun in „menschliche Zeitrechnung“ übersetzt werden, also in Jahre, Monate, ...aufgebrochen werden. Hierzu bedient man sich der Funktion `gmtime`, die einen Pointer auf ein Objekt vom Typ `time_t` annimmt, und ein Objekt vom Typ `struct tm` zurück gibt. Dieser struct ist beschrieben unter <https://en.cppreference.com/w/c/chrono/tm> und enthält Felder für Jahr, Monat, Tag, Mit `asctime` kann ein solches Objekt vom Typ `struct tm` in einen mit `printf` druckbaren Text umgewandelt werden (siehe <https://en.cppreference.com/w/c/chrono/asctime>).

Beispiel: time in ein Datum übersetzen

```
1 #include <stdio.h>
2 #include <time.h>
3
4 int main () {
5     time_t now = time(NULL);
6     printf("Now: %s", asctime(gmtime(&now)));
7 }
```

Ausgabebeispiel: time in ein Datum übersetzen

```
Now: Fri Jun 21 19:50:01 2019
```

Um Objekte vom Typ `struct tm` zu erzeugen, sollte die Funktion `mktime` verwendet werden (<https://en.cppreference.com/w/c/chrono/mktime>). Die Funktion `strftime` erlaubt außerdem auch die Darstellung in anderen Zeit-Formaten als dem Amerikanischen (<https://en.cppreference.com/w/c/chrono/strftime>).

15.1.3. Genaue Zeitmessung und Ticks

Zeitangaben, die im Zusammenhang mit der UNIX-Zeit stehen sind nur auf eine Sekunde genau. Wo Genauigkeit auf unter eine Sekunde (und bis zu Nanosekunden hinab) gefragt ist, kann mit dem Datentyp `clock_t` gearbeitet werden. Auch hier handelt es sich um ein Alias für einen Ganzzahl-Datentyp. Mit Objekten dieses Typs werden aber nicht die Sekunden seit der UNIX-Epoche gezählt, sondern die *Prozessortakte* seit einer bestimmten, nicht näher spezifizierten Referenzzeit. Üblicherweise ist diese Referenzzeit der Beginn der Ausführung des Programms.

Kennt man die Zahl der Prozessortakte, die in einer Sekunde umgesetzt werden, kann dies also leicht durch Division in eine Zeitdauer übersetzt werden. Genau diese Information – Prozessortakte pro Sekunde – ist in dem Macro `CLOCKS_PER_SEC` zugänglich:

Beispiel: clock

```
1 #include <stdio.h>
2 #include <time.h>
3
4 int main (void) {
5     clock_t start = clock();
6
7     // Zeitaufwändige Simulation
8
9     clock_t end = clock();
10
11     double cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
12     printf("Die Simulation lief für %df Sekunden.\n", cpu_time_used);
13 }
```

15.1.4. Kurze Wartezeiten

Manchmal möchte man den Ablauf seines Programms absichtlich verzögern. Dies könnte beispielsweise der Fall sein, wenn Sie ein Spiel programmieren, in dem Sie Ihren Spielern nicht Reaktionsvermögen einer CPU abverlangen wollen.

Zu diesem Zweck existiert die Funktion `nanosleep`, die im POSIX-Standard von 1993 definiert wurde. Der Standard kann als Erweiterung zum C-Standard aufgefasst werden. Befehle, die hiervon abgedeckt werden gehören nicht mehr zum normalen Sprachumfang und werden nicht alleine durch Einbinden der richtigen Bibliotheken frei geschaltet. Stattdessen muss eine Präprozessor-Konstante `_POSIX_C_SOURCE` dem Compiler mitteilen, dass diese erweiterten Funktionen mit beachtet werden sollen³.

Sobald `nanosleep` in Ihrem Code verfügbar gemacht wurde, können Sie es nach folgendem Beispiel benutzen:

Beispiel: `clock`

```
1  #define _POSIX_C_SOURCE 199309L    // this enables nanosleep
2  #include <time.h>
3
4  void wait_ms(unsigned long milliseconds) {
5      struct timespec sleeptime;
6
7      if(milliseconds > 999) {
8          sleeptime.tv_sec = (int) (milliseconds / 1000);
9          sleeptime.tv_nsec = (milliseconds
10                               - ((long) sleeptime.tv_sec * 1000)) * 1000000;
11      } else {
12          sleeptime.tv_sec = 0;
13          sleeptime.tv_nsec = milliseconds * 1000000;
14      }
15
16      nanosleep(&sleeptime, NULL);
17  }
```

(Quelle: <https://stackoverflow.com/questions/7684359>)

Eine genaue Beschreibung des Befehls finden Sie unter <http://man7.org/linux/man-pages/man2/nanosleep.2.html>

Es gäbe noch mehr zu sagen ...

In diesem Abschnitt sollte Ihnen nur ein Überblick in die Logik hinter der time-library gegeben werden. Ich ermutige Sie also, selbstständig die unter <https://en.cppreference.com/w/c/chrono> aufgelisteten Funktionen selbstständig zu erkunden.

15.2. Zufallszahlen

Computer sind *deterministische* Maschinen. Das bedeutet, dass alle Ergebnisse, die mit einem Computer gewonnen werden können, bereits durch seinen Ausgangszustand vorgegeben sind. Kennt man den

³Im Header `<time.h>` finden Sie ein `##if`, das dafür sorgt, dass einige Definitionen nur umgesetzt werden, wenn die Präprozessor-Konstante einen geeigneten Wert hat.

gesamten Speicherinhalt eines Rechners, so lässt sich daraus das Ergebnis jeder Berechnung und jedes Algorithmus ableiten. Dies schließt somit *Zufall* bereits vollkommen aus.

Während *echter* Zufall einem Computerprogramm nicht zugänglich ist, können wir aber zumindest Zahlenreihen erzeugen, die zufällig *wirken*. Es ist einem Menschen i. d. R. nicht möglich, die nächste Zahl einer solchen *Pseudozufallsreihe* vorherzusagen. In der Regel sind solche Zufallsreihen *rekursiv* definiert, d. h. eine Pseudozufallszahl wird aus ihrem Vorgänger berechnet. Damit gilt wieder, dass mit der ersten Zahl die komplette Zufallsreihe vorherbestimmt ist. Durch geschickte Wahl des Startwerts kann man aber echtem Zufall für die praktische Anwendung nahe genug kommen.

Im Header `<stdlib.h>` sind die Funktionen `rand` und `srand` deklariert, die zu diesem Zweck dienen.

`rand` gibt die nächste Zahl einer Zufallsreihe aus. Berechnet wird eine positive Ganzzahl zwischen 0 und `RAND_MAX`. Letzteres wiederum ist ein Symbol, das ebenfalls in `<stdlib.h>` definiert ist, und dessen Wert von der Version des Compilers abhängt. I. d. R. handelt es sich um die größte Zahl, die mit dem Datentyp `int` dargestellt werden kann.

`srand` legt den Startwert fest. Als Argument wird eine vorzeichenlose Ganzzahl erwartet.

Es bietet sich an, als Startwert beispielsweise die aktuelle Systemzeit zu verwenden, wie sie von `time` zurück gegeben wird. Da dieser Wert im Allgemeinen bei jeder Programmausführung ein anderer ist, erhält man einen für praktische Zwecke gut geeigneten Pseudo-Zufallsgenerator.

In der praktischen Anwendung kann dies so aussehen:

Beispiel: Pseudozufallszahlen

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  int main () {
6      srand(time(NULL));    // "zufälligen" Startwert wählen
7
8      printf("drei zufällige Ganzzahlen:\n");
9      for (int i=0; i<3; i++) {
10         printf("%d\n", rand());
11     }
12
13     printf("\ndrei zufällige Ganzzahlen zwischen 0 und 5:\n");
14     for (int i=0; i<3; i++) {
15         printf("%d\n", rand() % 6);
16     }
17
18     printf("\ndrei zufällige Fließkommazahlen zwischen 0 und 1:\n");
19     for (int i=0; i<3; i++) {
20         printf("%lf\n", (double) rand() / RAND_MAX);
21     }
```

```

22     double lo = 5;
23     double hi = 15;
24     printf("drei zufällige Fließkommazahlen zwischen %lf und %lf:\n", lo, hi);
25     for (int i=0; i<3; i++) {
26         printf("%lf\n", lo + ((double) rand() / RAND_MAX) * (hi-lo));
27     }
28 }

```

Ausgabebeispiel: Zufallszahlen

drei zufällige Ganzzahlen:

1726242556

1893651755

1463512462

drei zufällige Ganzzahlen zwischen 0 und 5:

3

1

1

drei zufällige Fließkommazahlen zwischen 0 und 1:

0.662454

0.426405

0.427842

drei zufällige Fließkommazahlen zwischen 5.000000 und 15.000000:

5.544294

5.964819

9.742072

Linear Congruential Generator – der C-Standard-Pseudozufallsgenerator

Der C-Standard schreibt nicht vor, wie der Algorithmus hinter `rand` genau auszusehen hat. Üblicherweise wird aber ein *Linear Congruential Generator* implementiert. Diese funktionieren nach dem Prinzip:

$$x_i = (ax_{i-1} + c) \mod m$$

Hierbei sind x_i die berechneten Pseudozufallszahlen, d. h. x_{i-1} steht für den Vorgänger der gerade berechneten Zahl. Die Parameter a , c und m sind mehr oder minder beliebige Konstanten. Der Wert von m legt den größten Wert fest, den diese Methode generieren wird. Eine geschickte Wahl von a und c sorgen für eine möglichst gleichmäßige Verteilung der Zufallswerte.

Der `gcc` verwendet:

$$m = 2^{31}$$

$$a = 1\,103\,515\,245$$

$$c = 12345$$

(Siehe https://en.wikipedia.org/wiki/Linear_congruential_generator)

Statistische Qualität von Zufallszahlen

Für allgemeine Anwendungen – etwa die Programmierung von Spielen – genügt der Standard-Zufallsgenerator des `gcc` vollkommen. Zu wissenschaftlichen Simulationen aber ist dieser *nicht geeignet*. Die einzelnen Werte sind noch zu stark korreliert, und die Periode (d. h. die Zahl von Zufallswerten, bevor sich die Wertereihe wiederholt) ist zu gering.

Für Wissenschaftliche Arbeiten stehen viele Bibliotheken zur Verfügung, die ausgereifere (aber auch langsamer arbeitende) Pseudozufalls-Generatoren anbieten. Ein Beispiel hierfür ist die *Gnu Scientific Library* (GSL).

Siehe <https://www.gnu.org/software/gsl/>

15.3. nan – not a number

Fließkommazahlen – also `floats` und `doubles` – haben eine interessante Eigenschaft: Sie können auch mathematische Objekte darstellen, die streng genommen *keine Zahlen* sind. Bestimmte Bitmuster werden nicht als Zahl ausgewertet sondern stehen für das Ergebnis einer fehlerhaften Berechnung wie etwa die Wurzel aus einer negativen Zahl⁴. Wir nennen diese Bitmuster *not a number* oder kurz *NAN*.

Je nach Datentyp und Compiler werden unterschiedliche NANs unterschieden. Allen ist gemeinsam, dass jede Fließkomma-Rechnung, an denen eine NAN beteiligt ist, wieder eine NAN gleicher Art erzeugt. Soll eine solche NAN in eine Ganzzahl-Variable übertragen werden, so erzeugt dies, abhängig von Ziel-Datentyp und Compiler, entweder den Wert 0 oder -2^b – wobei b die Breite des Datentyps in bits ist. Für `ints` (32bit) ergibt sich so der Wert `-2 147 483 648`.

Bei der Ausgabe mit `printf` werden alle NANs durch die Zeichenkette `nan` dargestellt:

Beispiel: NANs

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main () {
5      double nan = sqrt(-1.0);
6
7      char      nan_char  = nan;
8      short     nan_short = nan;
9      int       nan_int   = nan;
10     unsigned int nan_uint = nan;
11     long       nan_long  = nan;
12
13     printf("nan als double: %lf\n" , nan      );
14     printf("nan als char  : %hhd\n", nan_char );
15     printf("nan als short : %hd\n" , nan_short);
16     printf("nan als int   : %d\n" , nan_int   );
17     printf("nan als uint  : %d\n" , nan_uint  );
18     printf("nan als long  : %ld\n" , nan_long );
19
20     return 0;
21 }
```

⁴C bietet zwar eine Bibliothek zur Arbeit mit komplexen Zahlen an. Für den „normalen“ Betrieb gelten solche Rechnungen aber als nicht lösbar. Siehe dazu auch Abschnitt 15.6

Ausgabebeispiel: NaNs

```
nan als double: nan
nan als char   : 0
nan als short  : 0
nan als int    : -2147483648
nan als uint   : 0
nan als long   : -9223372036854775808
```

Um zu überprüfen, ob ein Fließkommawert ein NAN ist (gleich welcher Art) kann das Macro `isnan` (definiert im Header `<math.h>`) benutzt werden:

Beispiel: `isnan`

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main () {
5      printf("sqrt(+1.0)      ist %snan\n", (isnan(sqrt(+1.0)) ? "" : "kein "));
6      printf("sqrt(-1.0)      ist %snan\n", (isnan(sqrt(-1.0)) ? "" : "kein "));
7      printf("sqrt(-1.0) + 1 ist %snan\n", (isnan(sqrt(-1.0) + 1) ? "" : "kein "));
8
9      return 0;
10 }
```

Ausgabebeispiel: `isnan`

```
sqrt(+1.0)      ist kein nan
sqrt(-1.0)      ist nan
sqrt(-1.0) + 1 ist nan
```

Schließlich ist im Header `<float.h>` die Makro-Konstante `NAN` definiert, die zur schnellen bzw. klaren Erzeugung eines solchen Fehlerwerts geeignet ist.

15.4. `atexit`

Manche Aufgaben sollen erst ausgeführt werden, wenn das Programm beendet wird. Dies kann beispielsweise das Schließen von Logfiles sein oder die Freigabe von Speicherbereichen mit `free`. Damit wir nicht jeden Pfad, auf den unser Programm enden kann, nachverfolgen müssen, können wir mit der Funktion `atexit` (deklariert im Header `<stdlib.h>`) eine oder mehrere eigene Funktionen registrieren, die bei Ende unseres Programms aufgerufen werden sollen.

Als Parameter wird ein Funktionszeiger auf eine `void`-Funktion ohne Parameter erwartet. (Da bei Programmende keine Stelle mehr einen Rückgabewert empfangen kann, ist der Rückgabotyp `void` naheliegend. Bedingt durch den automatischen Aufruf können auch keine Parameter übergeben werden. Werden in der Funktion dennoch zusätzliche Informationen gebraucht, muss dies über globale Variablen gelöst werden.)

Beispiel: atexit

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  FILE * hDebug = NULL;
5
6  void handler_quit () {
7      if (hDebug) {
8          fclose(hDebug);
9          printf("Debug Log geschlossen.\n");
10     }
11 }
12
13 int main () {
14     atexit(handler_quit);
15     hDebug = fopen("debuglog.txt", "w");
16
17     // restlicher Code...
18 }
```

15.5. Variadische Funktionen

Wie wir wissen, können die Funktionen `printf` und `scanf` beliebig viele Parameter entgegen nehmen. Dies ist möglich, indem sie als *variadische Funktionen* deklariert sind. Das bedeutet, dass ihre Signatur nur eine gewisse Anzahl an festen Parametern vorsieht und danach durch drei Punkte angedeutet wird, dass hier beliebige Werte folgen dürfen:

Syntax: Prototyp einer variadischen Funktion

```
Rückgabetyt Funktionsname (Parameterliste_fest, ...);
```

Um auf die so übergebenen *variadischen Parameter* zuzugreifen, bietet der Header `<stdarg.h>` einige Macros an über die geeignete Pointer erhalten werden können.

Um die variadischen Parameter auszulesen benötigen wir zunächst ein Objekt vom Typ `va_list`, das als Handle auf die Parameter fungiert. Im folgenden soll diese Variable `args` genannt werden.

`args` wird über das Macro `va_start` initialisiert, dem wir dazu `args` als ersten Parameter übergeben müssen und zusätzlich den letzten Parameter, der „regulär“ an unsere variadische Funktion übergeben wurde.

Einmal initialisiert liest das Macro `va_arg` nun aus `args` Werte beliebigen Typs aus. Als Parameter wird `arg` sowie der Datentyp des nächsten zu lesenden Wertes übergeben. Ist die Arbeit mit der Liste beendet, muss mit `va_end` Speicher freigegeben werden.

Dieses abstrakte Vorgehen wird klarer, wenn wir ein Beispiel aus der CPP-Referenz betrachten:

Beispiel: Variadische Funktion zum Aufsummieren beliebig vieler ints

```
1  #include <stdio.h>
2  #include <stdarg.h>
3
4  int add_nums(int count, ...) {
5      int result = 0;
6
7      va_list args;                // Liste variadischer Parameter
8      va_start(args, count);       // Liste vorbereiten, Startpunkt nach count
9
10     for (int i = 0; i < count; ++i) {
11         result += va_arg(args, int); // int-Werte aus der Liste lesen
12     }
13
14     va_end(args);
15     return result;                // Handle auf variadische Liste freigeben
16 }
17
18 int main() {
19     printf("%d\n", add_nums(4, 25, 25, 50, 50));
20 }
```

Besteht die variadische Liste aus Parametern unterschiedlichen Typs, so müssen die festen Parameter die Information enthalten, welche Datentypen an welcher Stelle der Liste stehen. Üblicherweise geschieht dies über einen *Format String*. Die Interpretation eines solchen Strings nennt man auch *parsen*.

15.6. Komplexe Zahlen

Im Kontext von naturwissenschaftlichen Simulationen kommen häufig auch Rechnungen mit komplexen Zahlen vor. Der Header `<complex.h>` stellt einige Deklarationen und Makros bereit, die die Arbeit hiermit erleichtern.

Sobald der Header eingebunden ist, stehen die neuen Datentypen `complex float`, `complex double` und `complex long double` zur Verfügung. Wie ihre „Cousins“ in den reellen Zahlen unterscheiden sich die drei Datentypen in der *Rechengenauigkeit*, also in der Zahl der signifikanten Ziffern, die zur Verfügung stehen.

Mit Ausdrücken vom Typ `complex XXX` können die vier Grundrechenarten direkt mit den bekannten Operatoren (+, -, *, /) durchgeführt werden. Auch hier werden die gängigen Rechenregeln (Punkt vor Strich, Behandlung von Real- und Imaginärteil, ...) beachtet.

Der Header `<complex.h>` definiert auch das Symbol `I`, mit dem die imaginäre Einheit dargestellt wird. Mit den Funktionen `creal` und `cimag` kann der Real- bzw. Imaginärteil eines Ausdrucks vom Typ `complex XXX` ausgelesen werden.

Damit funktioniert dann der folgende Code:

Beispiel: Grundrechenarten mit komplexen Zahlen

```
1  #include <stdio.h>
2  #include <complex.h>
3
4
5  int main() {
6      complex double z1 = 1.0 + 2.0 * I;
7      complex double z2 = 5.0 - 2.7 * I;
8
9      complex double z3 = z1 * z2 + z1;
10
11     printf("Ergebnis: %lf%+lfi\n", creal(z3), cimag(z3));
12 }
```

Ausgabebeispiel: Grundrechenarten mit komplexen Zahlen

Ergebnis: 11.400000+9.300000i

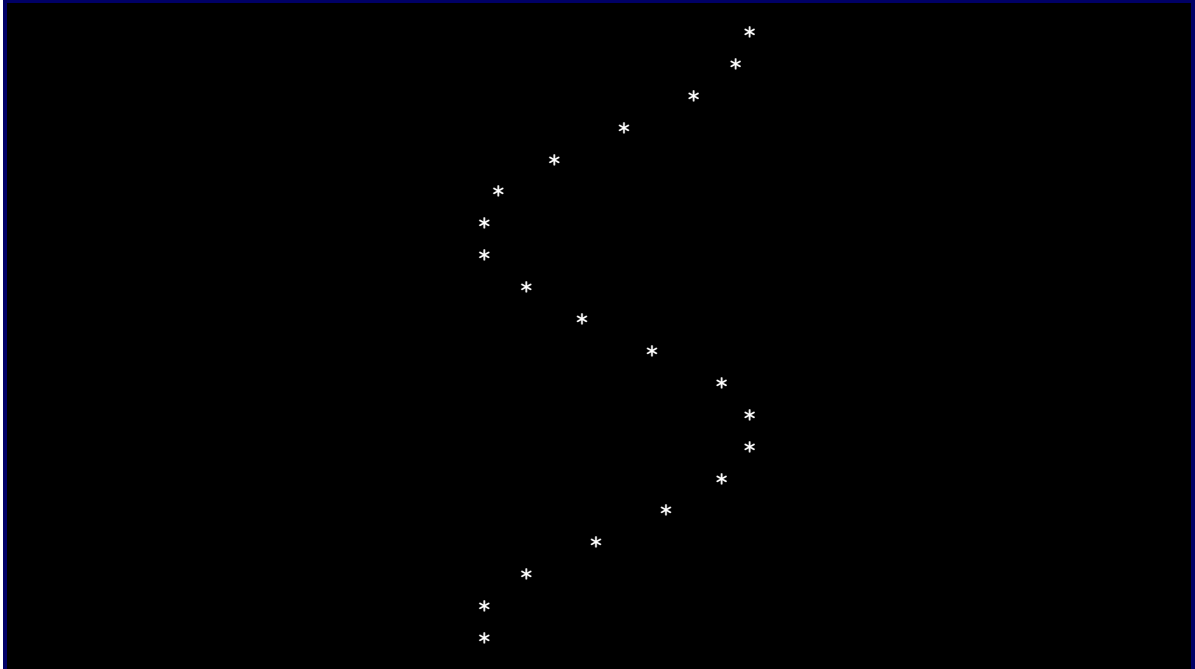
Für die wichtigsten aufwändigeren Operationen (Wurzel, Exponentialfunktion, Logarithmus, trigonometrische Funktionen, ...) existieren Analoga zu den bekannten Funktionen, die mit komplexen Zahlen funktionieren. Diese Funktionen haben denselben Namen wie die aus der `<math.h>` bekannten Routinen, beginnen aber mit dem Präfix `c`. Das bedeutet z. B., dass die Funktion `csin` den Sinus einer komplexen Zahl berechnet.

Um diese Funktionen benutzen zu können muss (wie auch schon bei den reellen mathematischen Funktionen) gegen die math-library gelinkt werden. Das heißt, der Compiler muss mit der Kommandozeilenoption `-lm` gestartet werden.

Beispiel: Funktionen mit komplexen Zahlen

```
1  #include <stdio.h>
2  #include <complex.h>
3
4  int main() {
5      const int    lines    = 20;
6      const double offset   = 40.0;
7      const double height   = 10.0;
8      const double frequency = 0.5;
9
10     complex double result;
11
12     for (int i=0; i<lines; i++) {
13         result = cexp(i * frequency * I);
14
15         for (int j=0; j<offset + creal(result) * height; j++) {
16             printf(" ");
17         }
18         printf("*\n");
19     }
20 }
```

Ausgabebeispiel: Funktionen mit komplexen Zahlen



Siehe <https://en.cppreference.com/w/c/numeric/complex> für eine Übersicht der komplexen Funktionen.

15.7. Debugger

Bis zu diesem Punkt haben Sie sicher schon festgestellt, wie häufig die ersten Zeilen Code, die wir schreiben fehlerhaft sind und nicht das Ergebnis erzielen, das wir uns erhoffen. Je größer und komplexer unsere Projekte werden, desto schwieriger wird es auch, einen Fehler ausfindig zu machen und zu korrigieren. Die Schwierigkeit besteht darin, die Entwicklung von den vielen Variablen nachzuverfolgen, mit denen wir arbeiten.

Um diese Aufgabe zu erleichtern steht das Tool **gdb** (GNU debugger) zur Verfügung. Das Programm wird von der Kommandozeile aufgerufen und bietet einen gewissen Einblick in unser laufendes Programm. Damit der Debugger diesen Einblick gewähren kann, müssen wir dem Compiler allerdings mitteilen, dass beim Übersetzen in Maschinsprache die Variablennamen und andere Informationen über unseren Quellcode erhalten bleiben sollen. Dies erreichen wir, indem wir den Compiler mit der Kommandozeilenoption **-g** aufrufen.

Performanz

Programme, die mit der Debug-Option **-g** kompiliert wurden sind größer und laufen langsamer. Während der Entwicklung ist es sinnvoll, die Option zu nutzen, um eben den Support des **gdb** nutzen zu können. Vergessen Sie jedoch nicht, die finale Version ohne diese Option zu kompilieren, um das Maximum an Leistungsfähigkeit herauszuholen.

Wenn Sie den **gdb** starten, finden Sie sich in einer neuen Kommandozeilen-Umgebung wieder. Das bedeutet, dass Sie, wie auch bei der Linux-Konsole, Kommandos eingeben und mit ENTER bestätigen können. Diese Kommandos sind jetzt jedoch andere, als dies in der normalen Konsole wäre.

Als ersten Schritt in der **gdb**-Umgebung können Sie mit dem folgenden Befehl festlegen, welches Programm Sie debuggen wollen:

```
gdb: Programm festlegen
```

```
file <executable>
```

Dabei steht **<executable>** für den Dateinamen Ihres *ausführbaren* Programms. In den vorigen Beispielen wäre das also jeweils `./myProgram` gewesen.

Der Befehl **run** führt Ihr Programm aus. Sie können auch Kommandozeilenparameter an Ihr Programm übergeben, indem Sie diese hinter **run** setzen:

```
gdb: Programm mit Kommandozeilenparametern starten
```

```
run param1 param2 ...
```

Sie können im Vorfeld Punkte festlegen, an denen die Ausführung des Programms pausiert werden soll. Dies geschieht mit dem Befehl **break**. Hinter **break** können Sie entweder einen Funktionsnamen oder eine Zeilennummer angeben.

Ist ein solcher *Breakpoint* erreicht, kann mit **print** der Wert von Variablen ausgegeben werden. Der folgende Befehl wird beispielsweise den Wert der Variablen `x` im aktuellen Zustand des Programms ausgeben:

```
gdb: Wert von Variablen ausgeben
```

```
print x
```

Schließlich lässt sich mit **continue** die Ausführung fortsetzen. Das Programm läuft weiter, bis ein neuer Breakpoint erreicht wird, bis das Programm regulär zu seinem Ende kommt oder bis es abstürzt.

Unter <https://web.eecs.umich.edu/~sugih/pointers/summary.html> finden Sie weitere Details zum GNU-Debugger.

Weitere Features

Aus Gründen der Übersichtlichkeit können hier nur die wichtigsten Features des **gdb** vorgestellt werden. Dies soll aber nicht den Eindruck erwecken, dass der Funktionsumfang des Debuggers komplett durch einige zusätzliche **printf**-Zeilen im Code ersetzt werden könnte. Lesen Sie beispielsweise unter <https://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf> über weitere Features des **gdb**.

16. Rekursion

To understand recursion, you must first understand recursion.

Anonymous

Viele beim Programmieren auftauchende Probleme lassen sich in kleinere Unter-Probleme zerlegen, die dieselbe Form haben wie die ursprünglich zu lösende Aufgabe. Solche zerlegbaren Aufgaben lassen sich so weit aufröseln, bis man bei einem *elementaren Problem* angekommen ist, das mit einfachen Mitteln zu lösen ist.

Beispiel: Sie wollen die komplette Ordnerstruktur auf Ihrem Rechner auflisten. Dazu müssen Sie, ausgehend von einem Stammverzeichnis, ...

- alle Dateien des Stammverzeichnisses auflisten
- alle Ordner im Stammverzeichnis auflisten
- *die Ordnerstruktur jedes Unterverzeichnisses auflisten*

In dieser Auflistung der Teilprobleme finden Sie also eine Unteraufgabe, die gleichlautend mit der ursprünglichen Aufgabe ist: Um eine Ordnerstruktur aufzulisten, müssen Sie eine *Sub*-Ordnerstruktur auflisten. Die Teilaufgabe ist aber „leichter“ als die Lösung des gesamten Problems, da nun nur eine Teilmenge des gesamten Ordnerstruktur aufgezählt werden muss. Schließlich wird man bei einer Teilaufgabe ankommen, die keine Sub-Ordnerstruktur mehr verlangt, da der Ausgangs-Ordner keine Unterordner mehr enthält. Die Auflistung eines solchen Ordners ohne Unterordner ist (verhältnismäßig) einfach.

Wir nennen Probleme, die sich auf gleichförmige Unterprobleme reduzieren lassen *rekursiv*. Die entsprechende Lösungsstrategie nennt sich dementsprechend *Rekursion*.

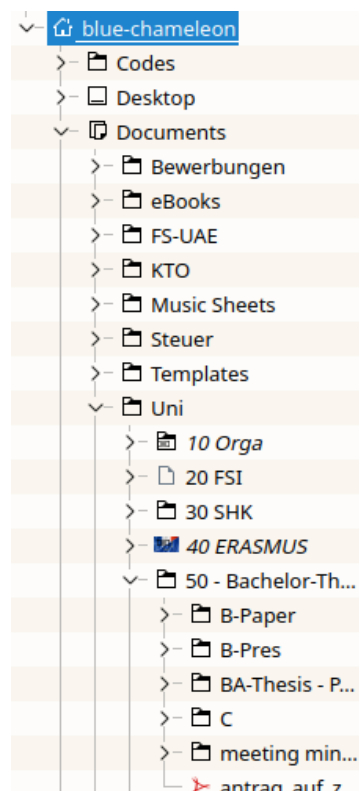


Abbildung 16.1.: Die Auflistung einer Ordnerstruktur besteht aus Auflistungen von Sub-Ordnerstrukturen

16.1. Funktionen, die sich selbst aufrufen

Behandeln wir zunächst ein einfaches Beispiel: Wir wollen alle Elemente eines `int`-Arrays aufsummieren. Natürlich können wir dies inzwischen schnell mit einer `for`-Schleife lösen. Für diesen Abschnitt wollen wir aber eine *rekursive Strategie* wählen:

Die Summe aller Elemente einer Liste ist gleich dem ersten Listenelement plus der Summe der verbleibenden Liste ohne ihr erstes Element.

Diese Aufgabe ist direkt lösbar, wenn die verbleibende Liste nur ein Element hat. Ansonsten erkennen wir wieder die rekursive Struktur des Problems.

Zur Lösung setzen wir eine Funktion an, die als Parameter einen Pointer auf den Listenanfang sowie die Länge der Liste erwartet. Diese Funktion wird *sich selbst aufrufen*, also *in die Rekursion gehen*.

Beispiel: Rekursives Aufsummieren

```
1  #include <stdio.h>
2
3  int listsum_recursive(int * list, unsigned int N) {
4      if (N == 1) {
5          return list[0];
6
7      } else if (N == 2) {
8          return list[0] + list[1];
9
10     } else {
11         return list[0] + listsum_recursive(list + 1, N - 1);
12     }
13 }
14
15 int main () {
16     int list[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
17
18     printf("Summe: %d\n", listsum_recursive(list, sizeof(list) / sizeof(*list)));
19     return 0;
20 }
```

Ausgabebeispiel: Rekursives Aufsummieren

Summe: 45

Wie nun funktioniert dieses Verfahren?

Die Zeilen 1 bis 8 sind unkompliziert. Der Fall einer Liste mit nur einem bzw. zwei Elementen wird mit elementaren Mitteln bearbeitet. Für den interessanten Fall, dass die Unterliste mehr als ein Element hat (**else** in Zeile 10) beschreiben wir die Rekursionsstrategie: Wir addieren das erste Listenelement `list[0]` zur Summe der Unterliste ohne dieses erste Element. Der Ausdruck `list + 1` beschreibt einen **int**-Pointer, der um ein **int**-Element gegenüber der ursprünglichen Liste verschoben ist. `list + 1` zeigt also auf eine „neue“ Liste, die mit dem *zweiten* Element von `list` beginnt.

In der Umsetzung werden in Zeile 11 die beiden Summanden getrennt voneinander ausgewertet. Das heißt, der Rechner liest den Wert `list[0]` aus dem Speicher. Die Auswertung des zweiten Summanden sieht (in *Pseudo-Syntax*, d. h. in nicht gültigem C-Code!) so aus:

Auswertung: Rekursionsstufe 0 in Pseudosyntax

```
return 1 + listsum_recursive({2, 3, 4, 5, 6, 7, 8, 9}, 8);
```

Als **Rekursionstiefe** oder *Rekursionsebene* bezeichnen wir die Anzahl der „verschachtelten“ Aufrufe. Da wir aus der Funktion `listsum_recursive` wieder dieselbe Funktion `listsum_recursive` aufrufen, haben wir also gerade Rekursionsebene 1 erreicht.

Natürlich muss jetzt noch der Funktionsaufruf von `listsum_recursive` aufgelöst werden. Dies geschieht nach demselben Prinzip wie schon der erste Aufruf, der von der `main` aus gestartet wurde. Wir gehen also wieder eine Rekursionsebene tiefer, wenn wir wieder auf Zeile 11 treffen. Die Zeile wird dann ausgewertet zu:

Auswertung: Rekursionsstufe 1 in Pseudosyntax

```
return 2 + listsum_recursive({3, 4, 5, 6, 7, 8, 9}, 7);
```

Auch wenn wir immer dieselbe Funktion aufrufen, wird jeweils ein neuer Scope betreten. Die Bezeichner `N` und `list` stehen also auf jeder Rekursions-Ebene für andere Speicherstellen. Sie können sich dies wie verschachtelte Boxen:

Rekursionsstufe 0: Symbol und Werte

`N = 9, list = {1, 2, 3, 4, 5, 6, 7, 8, 9}`

Rekursionsstufe 1: Symbole und Werte

`N = 8, list = {2, 3, 4, 5, 6, 7, 8, 9}`

Rekursionsstufe 2: Symbole und Werte

`N = 7, list = {3, 4, 5, 6, 7, 8, 9}`

Rekursionsstufe 3-6

...

Rekursionsstufe 7: Symbole und Werte

`N = 2, list = {8, 9}`

Obwohl wir formell in derselben Funktion bleiben, hat das Symbol `N` in Ebene 7 und Ebene 6 nichts miteinander zu tun. Dasselbe gilt für das Symbol `list`.

In Ebene 7 treffen wir endlich auf einen Fall, den unser Algorithmus ohne weiteren Rekursionsschritt auskommt. Es wird die Summe `8 + 9` berechnet und zurück gegeben. Dieser Rückgabewert wird nun in Zeile 11 der jeweils übergeordneten Rekursionsebene eingesetzt, um auch dort einen Rückgabewert zu berechnen. Der Stapel wird sozusagen *von innen heraus* aufgelöst:

Rekursionsstufe 0: Rückgabewerte

Rekursionsstufe 1: Rückgabewerte

Rekursionsstufe 2: Rückgabewerte

Rekursionsstufe 3-6

Rekursionsstufe 7: Rückgabewerte

```
return 8 + 9;
```

...

```
return 3 + 39;
```

```
return 2 + 42;
```

```
return 1 + 44;
```

Pseudocode und tatsächliche Werte

Im obigen Beispiel wurde der Anschaulichkeit halber `list` mit einer Menge von Werten ersetzt (Darstellung in {geschweiften Klammern}). Dies ist keine gültige C-Syntax!

In der Realität wird jeweils ein *Pointer* an die untergeordnete Rekursionsebene weitergeleitet. Alle diese Pointer zeigen auf *überlappende* Speicherbereiche, sind aber jeweils um die Speicherbreite einer Zahl gegeneinander verschoben. Da man nun verschiedene Speicherpunkte als Listen-Anfang interpretiert, und zugleich die Listenlänge N in jeder Rekursionsebene um 1 reduziert, ergeben sich effektiv die oben gezeigten Listen.

Während die oben gewählte Darstellung die Vorgänge klarer macht, sollten Sie sich bewusst sein, was tatsächlich im Speicher abgelegt wird.

16.2. Kommunikation über Rekursionsebenen hinweg

Wie üblich bei Funktionsaufrufen können Sie Werte zwischen den Rekursionsebenen austauschen, indem Sie diese als Parameter übergeben. Dies ist aber nicht immer wünschenswert. Stellen Sie sich etwa vor, Sie wollen eine Sicherheitssperre einbauen und die Rekursionstiefe auf 10 begrenzen. Dies ist möglich über folgenden Ansatz:

Beispiel: Rekursionstiefe mit Parameter

```
1  int recursiveFunc(unsigned int depth) {
2      if (depth > 10) {return 0;}
3
4      depth++;
5      recursiveFunc(depth)
6
7      // nützlicher Code...
8
9      return 1;
10 }
```

Allerdings müssen Sie hier beim Aufruf der Funktion `recursiveFunc` (z. B. aus der `main` heraus) den dort bedeutungslosen Parameter `depth` angeben. Um dem zu entgehen, können Sie auch das Schlüsselwort `static` benutzen.

Wie Sie noch aus Abschnitt 9.2.8 wissen, sorgt `static` dafür, dass die Speicherzelle für ein so deklariertes Symbol nicht freigegeben wird, wenn die Funktion verlassen wird, dass also der Wert des Symbols zwischen Funktionsaufrufen erhalten bleibt. In diesem Fall bedeutet das also, dass der Wert eines `static` Symbols auch über die Rekursionsebenen hinweg erhalten bleibt. Somit können wir das obige Beispiel auch *ohne* Parameter umsetzen:

Beispiel: Rekursionstiefe ohne Parameter

```
1  int recursiveFunc() {
2      static int depth = 0;
3
4      if (depth > 10) {return 0;}
5
6      depth++;
7      recursiveFunc()
8      depth--;
9
10     // nützlicher Code...
11
12     return 1;
13 }
```

Zeile 2 – die Deklaration und Wertzuweisung von `depth` wird nur ein einziges Mal¹ ausgeführt. Ab dann greift `depth` von jeder Rekursionsebene auf *dieselbe* Speicherstelle zu. Die Verwendung ist also ähnlich einer globalen Variablen, die aber nur innerhalb von `recursiveFunc` „sichtbar“ ist.

16.3. Laufzeitverhalten von rekursiven Methoden

Wie Sie wissen, bedeuten Funktionsaufrufe einigen Verwaltungsaufwand, der in Summe spürbar Zeit kosten kann. Hinzu kommt, dass die maximale Rekursionstiefe begrenzt ist, da für jeden Rekursionsschritt gespeichert werden muss, wo die Programmausführung nach Ende eines Rekursionsschritts fortgesetzt werden soll. Diese *Rücksprung-Adresse* wird auf dem Stack abgelegt, der nur begrenzt Speicherplatz anbietet. Zu viele Rekursionsebenen² führen zu einem sogenannten *Stack overflow*.

¹und tatsächlich bereits zur Kompilierzeit

²mehrere hundert

Es lässt sich beweisen, dass jeder *rekursive Algorithmus* sich auch in verschachtelten Schleifen schreiben lässt. Wo dies *einfach* möglich ist, ist eine solche Lösung ohne Rekursion auch zu bevorzugen. Das obige Beispiel der Summe über ein Array sollte also aus Effizienzgründen besser so implementiert werden:

Beispiel: Iteratives Aufsummieren

```
1  #include <stdio.h>
2
3  int listsum_iterative(int * list, unsigned int N) {
4      int reVal = 0;
5
6      for (unsigned int i=0; i<N; i++) {
7          reVal += list[i];
8      }
9
10     return reVal;
11 }
12
13 int main () {
14     int list[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
15
16     printf("Summe: %d\n", listsum_iterative(list, sizeof(list) / sizeof(*list)));
17     return 0;
18 }
```

Häufig macht eine rekursive Formulierung den Algorithmus aber für *Menschen* leichter verständlich, und damit auch besser wartbar, erweiterbar oder anpassbar. Dies gilt insbesondere dann, wenn die Problembeschreibung bereits rekursive Elemente enthält, bzw. wenn sich die Aufgabe in *Hierarchie-Ebenen* einteilen lässt. Im folgenden Abschnitt werde ich Ihnen eine rekursive Lösung zum Beispiel am Anfang des Kapitels zeigen. Ich lade Sie dazu ein, diese Lösung in eine nicht-rekursive Form umzuschreiben.

16.4. Beispiel: Rekursives Auflisten der Ordnerstruktur

Der folgende Code ist länger und verwendet einige Befehle, die hier noch nicht besprochen wurden. Sie werden aber feststellen, dass Sie inzwischen die Funktionsweise des Codes erfassen können, auch wenn Sie nicht jedes Detail verstehen. Die Namen der einzelnen Symbole sind dabei so gewählt, dass Ihnen das Verständnis erleichtert wird, wie es auch allgemein gute Praxis ist. Kommentare ergänzen dabei schwerer erfassbare Elemente in knapper Form.

Beginnen Sie beim Nachvollziehen mit der Funktion `showtree` ab Zeile 186. Diese enthält die Rekursion und ist für dieses Kapitel die interessanteste Funktion. Die Funktion `print_indented` ab Zeile 178 sollte Ihnen keine Probleme bereiten.

Fahren Sie dann fort mit der Definition von `stringlist` und den zugeordneten Funktionen `make_stringlist`, `free_stringlist`, `append_to_stringlist`, `get_stringlist_item` und `get_stringlist_length` in der abgedruckten Reihenfolge. Diese führen einen eigenen Datentyp mit zugehörigen Funktionen ein und illustrieren eine gängige Herangehensweise in der Programmietechnik: in ihren Eigenschaften eine Einheit bildende Informationen werden zu einem *Objekt* zusammengefasst. Die daran auftretenden Aufgaben werden von allgemein gefassten *Methoden* (Funktionen) erledigt³.

³Man spricht auch von *Reifikation*, oder „Verdinglichung“. In Kursen zur *Objektorientierten* Programmierung wie in C++ ist dies ein zentrales Konzept

Die Funktion `get_directory_entries` ist auf Anhieb vermutlich schwer zu erfassen, da hier die meisten bisher unbekannten Befehle auftauchen. Ich werde nach dem Code einige Erläuterungen dazu geben. Versuchen Sie aber zuerst, die Zusammenhänge selbst zu erraten.

16.4.1. Code

Beispielprogramm: `tree.c`

```
1  #define _GNU_SOURCE          // use all features from unistd.h
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <unistd.h>          // some file system functions...
7  #include <dirent.h>          // ... and some more ...
8  #include <sys/stat.h>        // ... and even more.
9
10 // ===== //
11 // handle a list of strings
12
13 typedef struct {
14     unsigned int N;           // number of strings in the list
15     char ** items;           // the list itself
16 } * stringlist;
17
18 // ----- //
19
20 stringlist make_stringlist() {
21     /* Create a well-defined initial state so that the other methods don't have
22      * to do excessive error-checking
23      */
24
25     stringlist reVal = malloc(sizeof(stringlist));
26
27     if (reVal) {
28         reVal->N = 0;
29         reVal->items = NULL;
30     } else {
31         printf("Error in make_stringlist: Could not allocate memory.\n");
32         return NULL;
33     }
34
35     return reVal;
36 }
37
38 // ..... //
```

```

39 void free_stringlist(const stringlist list) {
40     /* frees all memory occupied by the components of a stringlist.
41        */
42
43     // first, free the items themselves
44     for (unsigned int i=0; i<list->N; i++) {
45         if (list->items[i]) {free(list->items[i]);}
46     }
47
48     // then free the list of items
49     free(list->items);
50
51     // and finally, free the struct
52     free(list);
53 }
54
55 // ..... //
56
57 int append_to_stringlist(const stringlist list, const char * item) {
58     /* adds an item to the end of a stringlist.
59        * returns number of items in the list on success or -1 on failure.
60        */
61
62     // check whether a NULL pointer was passed
63     if (!list) {
64         printf("Error in append_to_stringlist: Invalid list.\n");
65         return -1;
66     }
67
68     // make a copy of the item to be added to the list
69     char * newItem = malloc(strlen(item) + 1);
70     if (!newItem) {
71         printf("Error in append_to_stringlist: Not enough memory for new item.\n");
72         return -1;
73     }
74     strcpy(newItem, item);
75
76     // make the list one item longer
77     char ** longerList = realloc(list->items, (list->N+1) * sizeof(*longerList));
78     if (!longerList) {
79         printf("Error in append_to_stringlist: Could not expand list.\n");
80         free(newItem);
81         return -1;
82     }
83
84     list->items = longerList;
85     list->items[list->N] = newItem;
86
87     return ++list->N;
88 }
89
90 // ..... //

```

```

91 char * get_stringlist_item(const stringlist list, const unsigned int i) {
92     /* access to stringlist items with error checks
93        */
94
95     // check whether a NULL pointer was passed
96     if (!list) {
97         printf("Error in get_stringlist_item: Invalid list.\n");
98         return NULL;
99     }
100
101     // check if index out of boundaries
102     if (i < list->N) {
103         return list->items[i];
104     } else {
105         printf("Error in get_stringlist_item: Invalid index.\n");
106         return NULL;
107     }
108 }
109
110 // ..... //
111
112 unsigned int get_stringlist_length(const stringlist list) {
113     if (!list) {
114         printf("Error in get_stringlist_length: Invalid list.\n");
115         return 0;
116     }
117
118     return list->N;
119 }
120
121 // ===== //
122 // create lists of files and directories in the current working directory
123
124 typedef enum {listtype_files, listtype_directories} listtypes;
125
126 stringlist get_directory_entries (const listtypes type) {
127     stringlist reVal = make_stringlist();
128
129     DIR * filesystem_handle = opendir(".");
130     // "." represents current work directory
131     // opendir returns NULL if it couldn't open the directory
132
133     struct dirent * directory_entry;
134     // will hold name of one item in the directory
135     struct stat      statbuffer;
136     // will hold kind of one item in the directory (file or folder)
137
138     if (filesystem_handle == NULL) {
139         printf("Error in get_subdirectories: Could not open current directory\n");
140         free_stringlist(reVal);
141         return NULL;
142     }
143 }

```



```

144 // read new entries from the directory as long as there are any.
145 while ( (directory_entry = readdir(filesystem_handle)) != NULL ) {
146     // skip special file system elements
147     if ((strcmp(directory_entry->d_name, "." ) == 0) ||
148         (strcmp(directory_entry->d_name, "..") == 0)
149     ) {continue;}
150
151     // analyze type of directory enty
152     if( stat(directory_entry->d_name, &statbuffer) == -1 ) {
153         printf("Error in get_subdirectories: "
154             "Could not determine kind of entry %s\n",
155             directory_entry->d_name);
156         free_stringlist(reVal);
157         return NULL;
158     }
159
160     int entry_is_directory = S_ISDIR(statbuffer.st_mode);
161     int condition_to_add =
162         ( entry_is_directory && (type == listtype_directories)) ||
163         (!entry_is_directory && (type == listtype_files      ));
164
165     if (condition_to_add) {
166         append_to_stringlist(reVal, directory_entry->d_name);
167     }
168
169 }
170
171 closedir(filesystem_handle);
172 return reVal;
173 }
174
175 // ===== //
176 // recursive traversal of file system
177
178 void print_indented(const char * text, const unsigned int n) {
179     for (unsigned int i=0; i<n; i++) {printf(" ");}
180
181     printf("%s\n", text);
182 }
183
184 // ----- //
185
186 void showtree(char * startDir) {
187     static unsigned int indent_level = 0;
188     int flag_free_startDir = 0;
189
190     if (!startDir) {
191         // use current work directory if nothing else was specified.
192         startDir = get_current_dir_name();
193         // this function uses malloc, thus a free() is needed later
194         flag_free_startDir = 1;
195         // store information: free is needed.
196     }

```

```

197     if ( chdir(startDir) ) {
198         // switch to directory startDir. Return 0 on success, -1 otherwise
199
200         printf("\x1b[91m"); // switch to colour red for error output
201         print_indented("(invalid directory)", indent_level);
202         printf("\033[m"); // restore normal colours
203         if (flag_free_startDir) {free(startDir);}
204         return;
205     }
206
207     printf("\x1b[96m"); // show directories in bright cyan
208     print_indented(startDir, indent_level);
209     printf("\033[m"); // restore normal colours
210
211     // get list of directories and go into recursion
212     stringlist subdirs = get_directory_entries(listtype_directories);
213     if (subdirs) {
214         for (unsigned int i=0; i<get_stringlist_length(subdirs); i++) {
215             indent_level++;
216             showtree(get_stringlist_item(subdirs, i));
217             chdir("../");
218             indent_level--;
219         }
220         free_stringlist(subdirs);
221     }
222
223     // get list of files
224     stringlist files = get_directory_entries(listtype_files);
225     if (files) {
226         for (unsigned int i=0; i<get_stringlist_length(files); i++) {
227             print_indented(get_stringlist_item(files, i), indent_level + 1);
228         }
229         free_stringlist(files);
230     }
231
232     if (flag_free_startDir) {free(startDir);}
233 }
234
235 // ===== //
236
237 int main (int argc, char ** argv) {
238     if (argc) {showtree(argv[1]);}
239     else      {showtree(NULL );}
240 }

```

16.4.2. Anmerkungen zu showtree

Als Parameter wird dieser Funktion ein **char *** übergeben, also ein String. Dieser benennt das Ausgangsverzeichnis, von dem ab die Auflistung der Ordnerstruktur beginnen soll. Neben einem „sinnvollen“ Pointer auf eine echte Zeichenkette kann aber auch der Wert **NULL** übergeben werden. In diesem speziellen Fall wird der Rechner angewiesen, vom aktuellen Arbeitsverzeichnis auszugehen. Der Name desselben

wird mit der Funktion `get_current_dir_name` ermittelt. Die Funktion ist im Header `<unistd.h>` deklariert, sobald das Macro `_GNU_SOURCE` definiert wird.

Wie Sie unter <http://man7.org/linux/man-pages/man3/getcwd.3.html> nachlesen können, reserviert `get_current_dir_name` für den Namen des aktuellen Arbeitsverzeichnisses Speicher, der später wieder mit `free` freigegeben werden muss. Wir *setzen eine Flag*, d. h. speichern die Information, dass wir dieses `free` später noch ausführen müssen in einer Variablen `flag_free_startDir`.

Die `static unsigned int indent_level` speichert die gegenwärtige Rekursionstiefe. Wir benutzen diese Information auch, um damit Einrückungen zu machen, und die hierarchische Struktur des Ordnersystems grafisch wiedergeben.

In Zeile 197 wird `chdir` aufgerufen. Auch diese Funktion ist im Header `<unistd.h>` deklariert, und setzt ein neues Arbeitsverzeichnis. Da `startDir` auch auf „unsinnige“ Strings zeigen kann, ist es möglich, dass dieser Verzeichniswechsel fehlschlägt. In diesem Fall ist der Rückgabewert von `chdir` gleich `-1`⁴. Lesen Sie hierzu mehr unter <http://man7.org/linux/man-pages/man2/chdir.2.html>.

Beachten Sie Zeile 201: Obwohl wir die Funktion `print_indented` bislang noch nicht besprochen haben, dürften Sie intuitiv erfassen, was hier geschieht: Der Text (`invalid directory`) wird auf dem Bildschirm ausgegeben, und zwar mit einer Einrückung, die der aktuellen Tiefe im Ordnerbaum entspricht. Dass der Effekt ohne genauere Betrachtungen erfasst werden kann ist eine Konsequenz aus der sinnvollen Benennung von Funktionen und Variablen.

Ähnlich verhält es sich mit Zeile 212:

```
stringlist subdirs = get_directory_entries(listtype_directories);
```

Eine neue Variable vom Typ `stringlist` mit Namen `subdirs` wird angelegt und speichert nun den Wert, der von `get_directory_entries(listtype_directories)` berechnet wurde. Obwohl wir bislang weder wissen, wie dieser Typ `stringlist` aufgebaut ist, noch wie exakt `get_directory_entries` funktioniert, können wir diese Zeile *semantisch* erfassen: gespeichert werden alle Einträge des aktuellen Arbeitsverzeichnisses, die dem `listtype_directories` entsprechen, also alle Unterordner des aktuellen Arbeitsverzeichnisses.

Wenn diese `stringlist` nicht leer war (Zeile 213), so wird jedes Element der Liste abgearbeitet. Da dies bei Unterordnern bedeutet, dass wir eine Ebene tiefer in die Rekursion gehen, erhöhen wir zuerst den Zähler `indent_level`, bevor wir rekursiv wieder `showtree` aufrufen. Statt im aktuellen Arbeitsverzeichnis zu starten, geben wir nun aber als Parameter `get_stringlist_item(subdirs, i)` an: der *i*-te Wert aus der Liste `subdirs`, also der Name des Unterordners, den wir gerade analysieren wollen. In diesem Rekursionsschritt wird unsere Funktion ein `chdir` ausführen, und so in das Unterverzeichnis wechseln. Wenn die Struktur des Unterverzeichnisses ausgegeben wurde, müssen wir natürlich wieder ins Ausgangsverzeichnis zurückkehren. Dazu dient in Zeile 217 der Befehl `chdir("..")`; Mit dem Wechsel ins Ausgangsverzeichnis sind wir auch wieder eine Hierarchieebene aufgestiegen; daher reduzieren wir den Zähler `indent_level`. Die Liste `subdirs` hat Speicherplatz beansprucht, der von hier an nicht mehr gebraucht wird. Da anscheinend die Struktur des Typs `stringlist` etwas komplexer ist, existiert zu diesem Zweck eine eigene Funktion `free_stringlist`, die für uns diesen „Aufräumvorgang“ erledigt.

Für die Ausgabe der Dateien passiert nochmals dasselbe, nur dass hier kein Rekursionsschritt mehr erfolgt. Stattdessen werden alle Dateien auf dem Bildschirm mit Hierarchieebene `indent_level + 1` ausgegeben, um zu symbolisieren, dass sie eben *unter* dem gerade analysierten Verzeichnis zu finden sind.

⁴Daneben sind auch andere Wege möglich, wegen derer der Verzeichniswechsel fehlschlagen kann. Beispielsweise kann der Zugriff auf einzelne Verzeichnisse nur dem Administrator gestattet sein, oder ein Netzlaufwerk ist wegen schlechter Verbindung zeitweise nicht erreichbar.

16.4.3. Anmerkungen zur stringlist

Wir haben bereits gesehen, dass es leicht nachzuvollziehen ist *was* die **stringlist** für uns leistet. Nun wollen wir verstehen, *wie* die umgesetzt wird.

In den Zeilen 13 bis 16 wird der Typ **stringlist** über ein **typedef** definiert. Es handelt sich um einen *Pointer auf* eine **struct** mit zwei Feldern: **N**, das die Zahl der Listenelemente speichert und **items**, welches die eigentliche Liste enthält. Da es sich um eine Liste von *Strings* handelt, muss der Datentyp von **items** **char **** sein.

Es ist in C üblich, bei komplexen Objekten den zugehörigen Datentyp gleich als Pointer anzulegen. Dies erlaubt, dass Funktionen Änderungen am Objekt durchführen (z.B. Listenelemente einfügen), ohne dass hierfür explizit der Adressoperator **&** benutzt werden müsste, und erlaubt eine komfortablere Anwendung.

Um ein Objekt vom Typ **stringlist** zu erzeugen, das „sinnvolle“ Werte hat, finden wir in Zeile 20 die Funktion **make_stringlist**. Sie reserviert zuerst den Speicherplatz für die Struktur selbst. War dies erfolgreich, setzen wir die Startwerte auf **N=0** und **items** auf einen den Wert **NULL**. Andernfalls wird auf dem Bildschirm eine Fehlermeldung ausgegeben und der Rückgabewert der Funktion **NULL** gesetzt. Damit kann auch nicht versehentlich mit zufällig im Speicher befindlichen Werten weiter gearbeitet werden.

Verwandt mit **make_stringlist** ist **free_stringlist**: Diese Funktion gibt allen Speicher frei, der von unserer Liste in Anspruch genommen wurde. Dies umfasst den Speicher für die einzelnen Strings, für die Liste der Strings, und für die Datenstruktur **stringlist** selbst.

Das Paar **make_stringlist** und **free_stringlist** kann man als *Constructor und Destructor* auffassen. Beides sind Begriffe aus C++ bzw. der objektorientierten Programmierung; dem Konzept nach erledigen diese beiden Funktionen aber dieselbe Aufgabe, nämlich das Erstellen und Bereinigen einer Datenstruktur vor bzw. nach Gebrauch.

Spannend ist nun die Funktion **append_to_stringlist**: Nach einer Sicherheitsprüfung in Zeile 63 wird zunächst eine Kopie des Elements angelegt, das wir an unsere Liste anhängen wollen (ab Zeile 69). Dies müssen wir tun, da unser Destructor explizit auch den Speicherplatz für die Strings in der Liste wieder freigibt. Wenn wir nun Pointer auf Strings übergeben, die nicht freigegeben werden dürfen (z.B. String Literals), würde dies zu einem Programmabsturz führen. Stattdessen gestalten wir unsere Routinen so, dass sie eine *geschlossene Einheit* bilden. Parameter, die *von außen* übergeben werden, also aus anderen Programmteilen stammen werden nur gelesen, nicht aber verändert. Wir sagen, *die stringlist verwaltet sich selbst*.

In Zeile 77 wird nun versucht, die Liste zu erweitern. Wir benutzen dazu **realloc** und speichern das Ergebnis dieses Versuchs zunächst in **longerList** zwischen. Dies hat folgenden Grund:

Stellen Sie sich vor, die Erweiterung schlägt fehl. Es kann zwar kein Speicher für **N+1 char *** gefunden werden; die alte Liste bleibt aber dennoch bestehen. Hätten wir also **items** direkt überschrieben, so hätte dieses Element jetzt den Wert **NULL**, und der Zugriff auf die alte Liste wäre verloren. Insbesondere könnte auch der hierfür reservierte Speicher nicht mehr freigegeben werden.

Wenn die Liste nicht erweitert werden kann, so ist auch unsere Kopie **newItem** überflüssig geworden und muss mit **free** freigegeben werden.

Haben dagegen alle Speicher-Allozierungen funktioniert, so können in unserer **stringlist list** die Änderungen eingetragen werden: **items** erhält nun die neue Adresse, die wir in **longerList** zwischengespeichert hatten. An das Ende der Liste wird **newItem** gestellt. Da wir den Zähler **N** noch nicht erhöht haben, ist **N** auch Index des letzten Elements.

In Zeile 87 erhöhen wir endlich auch diesen Zähler und geben die neue Anzahl der Elemente in der Liste zurück.

`get_stringlist_item` und `get_stringlist_length` bieten lediglich ein komfortableres Interface auf unsere Datenstruktur: Einige Sicherheitsprüfungen werden durchgeführt, außerdem sind die Namen leichter interpretierbar als `N` und `items`. Ein Anwender unserer Datenstruktur muss hier weniger gedankliche Energie investieren, um „richtig“ damit umzugehen, und kann auch nicht versehentlich den Wert von `N` ändern.

16.4.4. Anmerkungen zu `get_directory_entries`

Dieser Abschnitt beginnt mit der Definition einer `enum`, die zwei Symbole `listtype_files` und `listtype_directories` festlegt. Sie haben schon gesehen, dass diese benutzt werden, um zu unterscheiden, welche Art von Liste angelegt werden soll.

Die Funktion beginnt damit, eine `stringlist` zu initialisieren. Weiter wird in Zeile 130 auch ein Objekt `filesystem_handle` vom Typ `DIR` durch `opendir` erstellt. Ähnlich wie schon `FILE` handelt es sich hierbei um ein sehr komplexes Objekt, das wir nicht näher verstehen wollen. Es stellt eine Schnittstelle zum Betriebssystem dar, und symbolisiert einen Zugriff auf ein Verzeichnis – in diesem Fall das aktuelle Arbeitsverzeichnis. Dies wird durch den Parameter `"."` zum Ausdruck gebracht. Wenn ein anderes Verzeichnis als das aktuelle Arbeitsverzeichnis analysiert werden sollte, könnten wir den Namen des Verzeichnisses an dieser Stelle einsetzen. Lesen Sie hierzu mehr unter <http://man7.org/linux/man-pages/man3/opendir.3.html>.

Über dieses `filesystem_handle` können nacheinander die einzelnen Elemente im gewählten Verzeichnis gelesen werden. Die hierbei erhaltenen Informationen speichern wir in Form eines `struct dirent *`. Die Information, ob es sich um einen Ordner oder eine Datei handelt, wird in der `struct stat` zu finden sein⁵. Lesen Sie hierzu mehr unter <http://man7.org/linux/man-pages/man3/readdir.3.html>.

In Zeile 145 benutzen wir nun `readdir` aus dem Header `<dirent.h>`, um nacheinander die Namen aller Dateien und Ordner im gewählten Verzeichnis zu erfahren. Die Funktion gibt `NULL` zurück, wenn alle Verzeichnisinhalte einmal bearbeitet wurden und markiert damit das Ende für die Schleife in dieser Zeile. Da die Strings `"."` für das aktuelle Arbeitsverzeichnis und `".."` für das übergeordnete Verzeichnis stehen, wollen wir diese nicht in unserer Auflistung beachten. In den Zeilen 147 bis 149 werden sie daher einfach übersprungen.

In Zeile 152 fragt schließlich die Funktion `stat` (Header `<sys/stat.h>`) weitere Informationen zu dem gerade erhaltenen Datenobjekt ab. Ob es sich dabei um ein Verzeichnis handelt, kann mit dem Macro `S_ISDIR` in Erfahrung gebracht werden. Lesen Sie hierzu mehr unter <http://man7.org/linux/man-pages/man2/stat.2.html> und <http://man7.org/linux/man-pages/man7/inode.7.html>.

⁵Mit dieser `struct` können bei weitem mehr Informationen zugänglich gemacht werden, auf die wir hier leider nicht eingehen können. Da tatsächlich so viele Informationen vorliegen, sind wir zum Umgang mit diesen unhandlichen Strukturen gezwungen.

17. Linked Lists

Where did you come from, where did you go? // Where did you come from, Cotton-Eye Joe?

Rednex

Unsere Arbeit mit Listen involvierte bislang immer Arrays. Dies ist günstig, solange wir die Liste nicht vergrößern oder verkleinern wollen. Soll aber ein Element an die Liste angehängt werden, kann dies ein sehr ungünstiges *Laufzeitverhalten*¹ erzeugen. Noch schwieriger wird es, wenn ein neues Element *in der Mitte der Liste* eingefügt werden soll.

In diesem Kapitel werden wir eine Datenstruktur kennenlernen, die bei Einfügungen und Löschung aus der Liste ein wesentlich besseres Laufzeitverhalten zeigt.

17.1. Ausgangslage: Klassische Arrays

Beispiel: Einfügen in eine Liste: Lösung mit Arrays

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int * insert_into_list(int * list, int N, int newVal, int pos) {
5      if (pos<0) {printf("Fehler: Einfügung vor dem Listenanfang\n"); return NULL;}
6      if (pos>N) {printf("Fehler: Einfügung hinter dem Listenende\n"); return NULL;}
7
8      int * newlist = malloc((N + 1) * sizeof(*newlist));
9      if (!newlist) {printf("Fehler: Allokierung fehlgeschlagen\n"); return NULL;}
10
11     for (int i=0; i<pos; i++) {
12         newlist[i] = list[i];
13     }
14
15     newlist[pos] = newVal;
16
17     for (int i=pos+1; i<N+1; i++) {
18         newlist[i] = list[i-1];
19     }
20
21     free(list);
22     return newlist;
23 }
```

¹Das Laufzeitverhalten ist ein Maß für den Zeitbedarf eines Algorithmus, in Abhängigkeit von der zu verarbeitenden Datenmenge. Wenn eine Liste mit n Elementen mit dem Befehl `realloc` vergrößert werden soll, kann es sein, dass diese n Werte an eine andere Stelle im Speicher „umgezogen“ werden müssen. Dieses Umziehen bedeutet Kopieren und kostet Zeit, und zwar für jedes Listenelement. Je länger die Liste ist, desto mehr Zeit kann das `realloc` also beanspruchen.

```

24 void print_list(int * list, int N) {
25     for (int i=0; i<N; i++) {
26         printf("Element #d: %3d\n", i, list[i]);
27     }
28 }
29
30 int main() {
31     int N = 5;
32     int * list = malloc(N * sizeof(*list));
33
34     if (!list) {
35         printf("Fehler: Allokierung fehlgeschlagen\n");
36         return -1;
37     }
38
39     for (int i=0; i<N; i++) {
40         list[i] = i;
41     }
42
43     printf("vorher:\n");
44     print_list(list, N);
45
46
47     int * dummy = insert_into_list(list, N, 666, 3);
48     if (dummy) {list = dummy; N++;}
49     else        {printf("Keine Einfuegung hat stattgefunden.\n");}
50
51     printf("nachher:\n");
52     printlist(list, N);
53
54     free(list);
55     return 0;
56 }

```

Ausgabebeispiel: Einfügen in eine Liste: Lösung mit Arrays

```

vorher:
Element #0: 0
Element #1: 1
Element #2: 2
Element #3: 3
Element #4: 4
nachher:
Element #0: 0
Element #1: 1
Element #2: 2
Element #3: 666
Element #4: 3
Element #5: 4

```

Wir legen hier in der `main` ein dynamisches Array mit `N=5` Elementen an, und befüllen Sie mit Beispieldaten, hier aufsteigend die Werte `0...N-1`. Die Funktion `insert_into_list` fügt dann an vierter Stelle

(Arrayindex 3) einen neuen Wert ein.

Dieses Einfügen geschieht, indem eine *Kopie der Original-Liste* angelegt wird. Diese Kopie geht aber nur bis zum Listenelement `pos`, also bis zu der Stelle, an der ein neues Element in die Liste eingefügt werden soll. Ebenso werden die Listenelemente mit den Indizes `pos...N-1` kopiert, jedoch mit einem *Offset* von 1, so dass das neue Element eingefügt werden kann.

Für jede Einfügung wird also die volle Liste kopiert². Außerdem ändert sich nach jeder Einfügung der Wert des Pointers `list`, was eine Fehlerquelle darstellt – es ist leicht, dieses Update zu vergessen.

Betrachten wir im Folgenden eine Struktur, die hier ein günstigeres Verhalten zeigt.

17.2. Verknüpfung mit seinen Nachbarn: Linked Lists

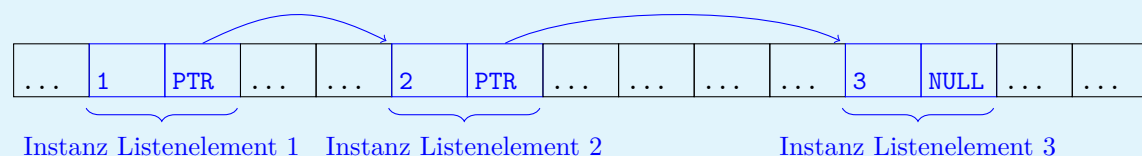
Arrays werden im Speicher dicht an dicht gepackt. Dies ist der Grund, warum in die Listen-Mitte nicht einfach ein Wert eingefügt werden kann, da sonst andere Listenelemente überschrieben werden. Diese dichte Packung ist aber nicht zwingend notwendig, solange nur bekannt ist, *wo im Speicher* ein bestimmter Wert zu finden ist. Insbesondere können wir für jedes Listenelement speichern, wo sein Nachfolger zu finden ist. Dazu bedienen wir uns einer `struct`:

Datentyp für Elemente einer Linked List

```
1 typedef struct listElement_struct {
2     int data;
3     struct listElement_struct * next;
4 } listElement_t;
```

Wir definieren einen Datentyp `listElement_t`³. Dieser Datentyp besteht aus „Nutzdaten“ `data` und der Information, wo das nächste Element der Liste im Speicher zu finden ist (Feld `next`). Diese „Ortsangabe“ verweist wieder auf eine Instanz des Typs `listElement_t`. Da solche Selbstbezüge aber mit `typedef` nicht möglich sind, arbeiten wir zusätzlich mit dem Hilfs-Datentyp `struct listElement_struct`. Im weiteren verwenden wir aber nur noch `listElement_t`. Sie können daher auch das Element `next` als einen Pointer auf eine Instanz von `listElement_t` verstehen.

Visualisierung: Verkettete Liste



Jede Instanz von `listElement_t` besteht aus zwei „Zellen“, die mehr oder minder zufällig im Speicher angeordnet sind. Auch die Reihenfolge ist nicht festgelegt; das dritte Element der Liste kann eine kleinere Adresse haben als das zweite. Dennoch ist die Liste in ihrer Gänze geordnet reproduzierbar, da zu jedem Element bekannt ist, wo der Nachfolger im Speicher ist. Diese Information ist im Feld `next` enthalten.

²Eine Lösung, bei der nur die letzten `N-pos` Elemente kopiert werden ist denkbar. Dies verbessert das Laufzeitverhalten jedoch nur unwesentlich.

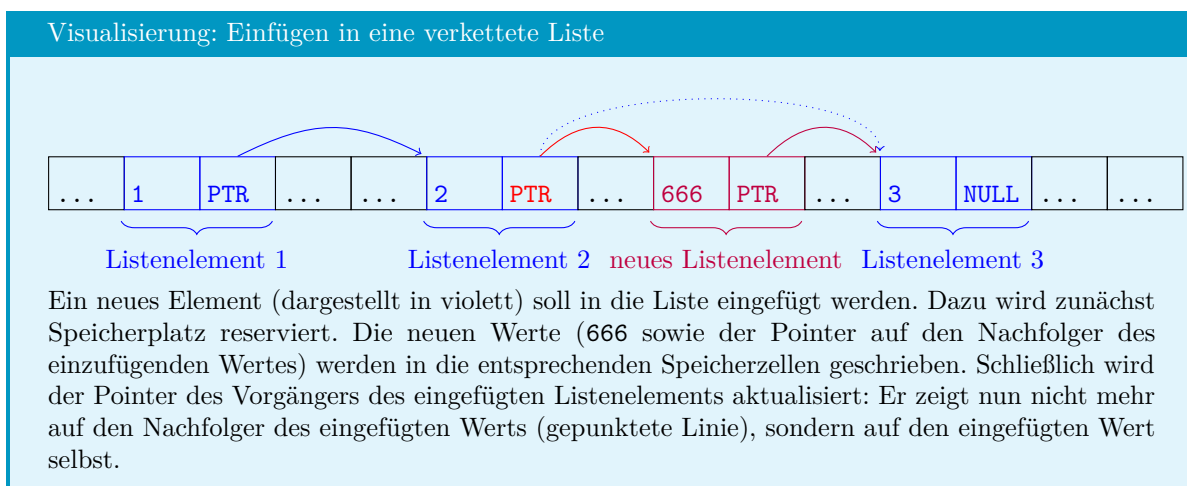
³Es ist eine verbreitete Konvention, selbstdefinierte Datentypen auf den Suffix `_t` enden zu lassen. In der Praxis wird dies aber nicht sehr streng gelebt.

Das letzte Element der Liste – hier das Dritte – hat keinen Nachfolger mehr. Daher wird **next** auf den Wert NULL gesetzt.

Wenn nun ein neues Element in die Liste eingefügt werden soll, so müssen nur drei Schritte ausgeführt werden:

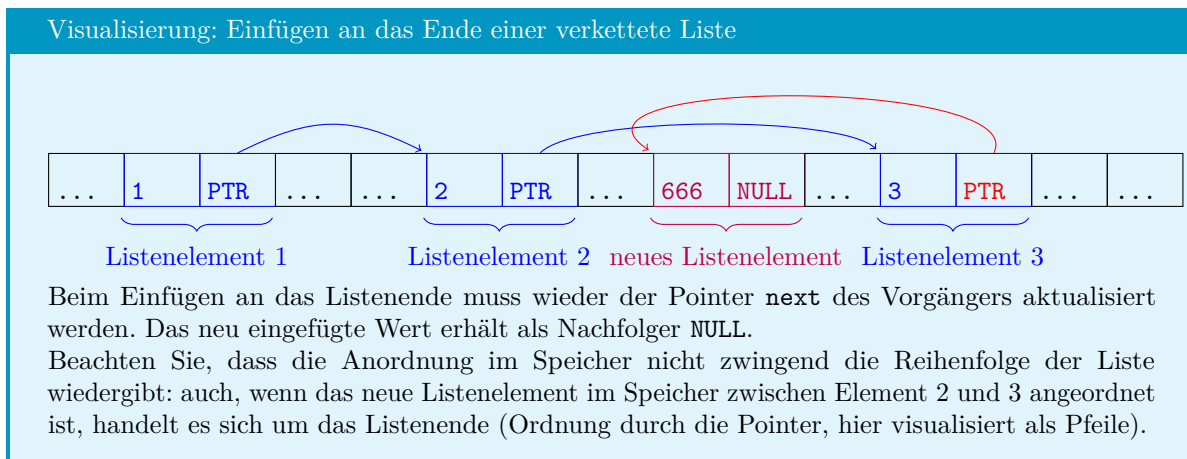
- Speicher für das neue Element bereit stellen
- Den Pointer **next** seines Vorgängers auf das einzufügende Element umleiten
- Den Pointer **next** des einzufügenden Elements auf seinen Nachfolger

Wir können dies so verbildlichen:



Solange wir also das erste Element einer Liste kennen, können wir jedes weitere Element erreichen, indem wir „von Listenelement zu Listenelement springen“⁴. Wenn neue Elemente in die Liste eingefügt werden sollen, so muss nur die Speicherstelle ausfindig gemacht werden, die das Listenelement beschreibt, *hinter* das eingefügt werden soll.

Hierbei gibt es noch zwei Spezialfälle: Wenn ein Element an das *Listenende* angefügt werden soll, gibt es keinen Nachfolger des eingefügten Elements. In diesem Fall wird der Pointer **next** auf NULL gesetzt:

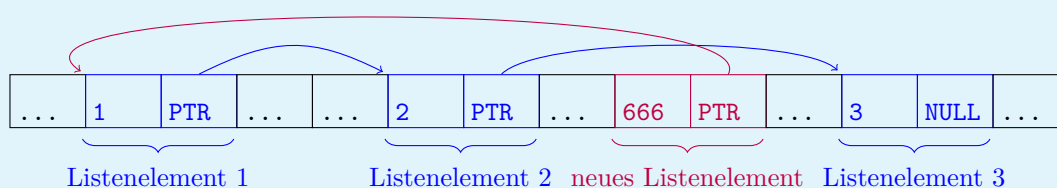


Der andere Spezialfall tritt ein, wenn an den *Anfang* der Liste ein Wert eingefügt werden soll. In diesem Fall müssen wir in der Speicherstruktur unserer Liste keine Pointer aktualisieren. Es ändert sich aber

⁴Natürlich kostet jeder dieser Sprünge Zeit. Wir werden uns hierzu in Abschnitt 17.4 mehr Gedanken machen

der „Einstiegspunkt“: Da wir nur „vorwärts“ in unserer Liste weiter springen können, müssen wir als *handle* die Adresse des eingefügten Elements speichern.

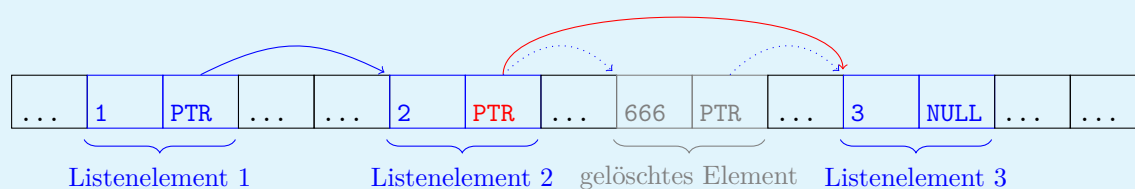
Visualisierung: Einfügen an den Anfang einer verkettete Liste



Beim Einfügen an den Listenanfang muss keines der bestehenden Listenelemente aktualisiert werden. Da jedoch nur Elemente in „Vorwärtsrichtung“ (der Pfeile) erreicht werden können, ist unser Einstiegspunkt für die Listenverwaltung das neu eingefügte Element.

Auch das Löschen von Werten aus der Liste ist jetzt leicht umsetzbar: Wir suchen den *Vorgänger* des zu löschenden Elements, ändern seinen Pointer auf den *Nachfolger* des zu löschenden Elements, und vergessen nicht, den Speicher freizugeben:

Visualisierung: Löschen aus einer verketteten Liste



Beim Löschen eines Elements – hier des Eintrags zwischen Element 2 und Element 3 – muss lediglich der Pointer *next* des *Vorgängers* aktualisiert und der Speicher freigegeben werden.

Auch hier ergeben sich Spezialfälle für Löschen am Listenanfang bzw. am Listende. Sie können sich jetzt leicht vorstellen, wie diese Fälle aussehen. Im folgenden Abschnitt werden wir diese mit Codebeispielen genauer betrachten.

17.3. Aufbau einer Bibliothek zur Verwaltung von Linked Lists

Die im letzten Abschnitt besprochenen Ideen haben bereits einige Komplexität. Wir wollen nicht für jedes Projekt, in dem wir diese Technik brauchen, komplett von Null weg beginnen. Daher schreiben wir uns eine *Bibliothek*, in der wir die Routinen zur Verwaltung einer Linked List in allgemeiner Form ablegen.

17.3.1. Die Datentypen

Da sich der Einstiegspunkt in unsere Liste ändern kann, führen wir einen weiteren Datentyp (eine weitere **struct**) ein, in dem wir diesen Einstiegspunkt und andere Daten speichern. Diesen neuen Datentyp wollen wir `linkedlist_t` nennen.

Während sich durch unsere Operationen (z. B. Einfügen in und Löschen aus der Liste) der Einstiegspunkt ändern kann, bleibt die Speicherstelle der Variable vom Typ `linkedlist_t` konstant. Wenn wir unsere Routinen so gestalten, dass diese diese „Verwaltungs-Variable“ aktuell halten, können wir für

die Verwendung unserer Liste ihre innere Struktur „vergessen“. Wir erledigen einmal die Aufgaben Speicherverwaltung und Listen-Management. Für unser eigentliches Projekt dann können wir einfach die Routinen unserer Bibliothek verwenden und müssen keine Gedanken mehr darauf aufwenden, ob alle Pointer auf die richtigen Speicherstellen zeigen.

Wir wollen mit unserer Bibliothek Listen beliebigen Datentyps anlegen. Dies umfasst sowohl die *primitiven Datentypen* (`int`, `double`, ...) als auch selbst erstellte *structs*. Daher werden wir den Datentyp von `data` in der `listElement_t` auf `void *` abändern. Das heißt, wir speichern „in der linken Zelle“ unserer Listenelemente nicht mehr den Wert selbst, sondern nur, wo im Speicher sich der tatsächliche Wert befindet. Um die Ausgabe der Liste auf dem Bildschirm zu vereinfachen, fügen wir `linkedlist_t` einen Funktionszeiger hinzu. Dieser soll eine Funktion benennen, die den Datentyp der Liste gut auf dem Bildschirm ausgeben kann.

Aus praktischen Gründen speichern wir außerdem noch mit, wie viele Elemente unsere Liste im Moment hat. Diese Information ist zwar durch Nachverfolgen der Pointer zugänglich; bei langen Listen dauert es aber, bis man durch die gesamte Kette gesprungen ist. Schließlich führen wir noch den Wahrheitswert `memoryAutoManaged` ein, dessen Funktion ich weiter unten genauer erläutere.

Aus diesen Überlegungen ergeben sich diese Datentypen:

Datentypen für Listenelemente und Verwaltungsvariable einer Linked List

```
1  typedef struct listElement_struct {
2      void *          data;
3      struct listElement_struct * next;
4  } listElement_t;
5
6  typedef struct {
7      listElement_t * first;
8      int            size;
9
10     int            memoryAutoManaged;
11     void (*printElement)(void *);
12 } linkedList_t;
```

Da dies bislang nur *Definitionen* waren, gehört dieser Code in eine Header-Datei. Wir wollen sie `linkedlist.h` nennen. Der folgende Code soll in einer Moduldatei mit dem Namen `linkedlist.c` gespeichert sein.

17.3.2. Aufbau und Bereinigung der Liste – Constructor und Destructor

Um mit der Liste Arbeiten zu können, muss zuerst ein definierter Ausgangszustand geschaffen werden. Hier werden wir bereits Speicherplatz allozieren müssen, daher sollten wir uns im selben Abschnitt der Entwicklung auch bereits Gedanken um die Freigabe des allozierten Speichers machen.

Routinen, die Speicherstrukturen vorbereiten werden *Constructors* genannt. Prozeduren, die Speicher wieder freigeben, bezeichnen wir als *Destructors*. Wir entwickeln hier also den Constructor `make_linkedList` und den Destructor `free_linkedList`.

Da jede Einfügung oder Löschung den Einstiegspunkt (`first`) und die Zahl der Elemente (`size`) ändern können, müssen unsere Routinen mit Pointern auf die Verwaltungs-Variable arbeiten. Damit wir vermeiden können, in jedem Aufruf der Routinen unserer Bibliothek einen Adress-Operator `&` setzen zu müssen, konzipieren wir gleich den Constructor so, dass ein Pointer erzeugt wird:

Constructor einer Linked List

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "linkedlist.h"
5
6  linkedList_t * make_linkedList () {
7      linkedList_t * reVal = malloc(sizeof(*reVal));
8      if (!reVal) {
9          printf("Fehler: Speicher konnte nicht alloziert werden\n");
10         return NULL;
11     }
12
13     reVal->first          = NULL;
14     reVal->size           = 0;
15     reVal->printElement   = NULL;
16     reVal->memoryAutoManaged = 0;
17
18     return reVal;
19 }
```

Alle Elemente der Verwaltungsstruktur werden zunächst auf null gesetzt. Damit sind zufällige Effekte von „Resten im Speicher“ ausgeschlossen.

Der Destructor muss alle Elemente der Liste löschen und schließlich den Speicherplatz für die Verwaltungsstruktur vom Typ `linkedList_t` selbst freigeben. Da wir später ohnehin einzelne Listenelemente löschen wollen werden, können wir für diese Aufgabe auf die (noch nicht existierende) Funktion `delete_from_linkedList` zurückgreifen. Dieser Funktion übergeben wir den Index des zu löschenden Listenelements. Wenn wir oft genug das erste Element der Liste entfernen, geben wir am Ende allen Speicher frei. Der Code lässt sich folgendermaßen schreiben:

Destructor einer Linked List

```
20 void free_linkedList(linkedList_t * list) {
21     while (list->size) {
22         delete_from_linkedList(list, 0);
23     }
24
25     free(list);
26 }
```

17.3.3. Sprünge durch die Liste

Es wäre naheliegend nun die Funktion `delete_from_linkedList` zu schreiben. Da wir für jede Veränderung der Liste jedoch zuerst ein bestimmtes Listenelement „finden“ müssen, bietet es sich an, zuerst diese Sprünge durch die Liste umzusetzen. Wir entwickeln also eine Routine, der wir einen Index mitteilen, und die das zugehörige Listenelement ausfindig macht:

Sprung zu einem bestimmten Element einer Linked List

```
27 listElement_t * get_linkedList_element(linkedList_t * list, int index) {
28     if (index < 0 || index >= list->size) {
29         printf("Fehler: ungültiger Index\n");
30         return NULL;
31     }
32
33     listElement_t * element = list->first;
34     for (int i=0; i<index; i++) {
35         element = element->next;
36     }
37
38     return element;
39 }
```

17.3.4. Löschen und Einfügen in die Liste

Mit diesem Werkzeug können wir nun endlich das Löschen von Elementen aus der Liste angehen; im selben Schritt werden wir uns dann auch mit dem Einfügen in die Liste beschäftigen.

Löschen aus einer Linked List

```
40 int delete_from_linkedList(linkedList_t * list, int index) {
41     if (index < 0 || index >= list->size) {
42         printf("Fehler: ungültiger Index\n");
43         return -1;
44     }
45
46     listElement_t * prev, * self, * next;
47
48     if (index == 0) {
49         prev = NULL;
50         self = list->first;
51         list->first = self->next;
52     } else {
53         prev = get_linkedList_element(list, index - 1);
54         self = prev->next;
55     }
56
57     next = self->next;
58     if (prev) {prev->next = next;}
59
60     if (list->memoryAutoManaged) {free(self->data);}
61     free(self);
62
63     return --list->size;
64 }
65 }
```

Wir beschaffen uns zunächst die Adressen des zu löschenden Elements, sowie die seines Vor- als auch Nachgängers (Variablen `self`, `prev` und `next`.) Wenn das erste Element der Liste gelöscht werden soll

(wenn `index` gleich 0 ist), existiert kein Vorgänger. Wir behandeln diesen Fall gesondert (`if` in Zeile 48). Hier beachten wir auch, dass sich der Einstiegspunkt in unsere Liste ändern wird (Zeile 51). Ansonsten benutzen wir die soeben geschriebene Funktion `get_linkedList_element`, um das zu löschende Element ausfindig zu machen. In jedem Fall ist sein Nachfolger über `self->next` ausfindig zu machen. Da wir mit dem `if` in Zeile 41 prüfen, ob ein „sinnvoller“ `index` übergeben wurde, ist auch sichergestellt, dass `self` von `NULL` verschieden ist. (Sie erinnern sich, dass `self->next` eine Kurzform für `(*self).next` ist, und dass `*pointer` verboten ist, wenn `pointer == NULL`.)

Wird `self` gelöscht, so soll das Listenelement an der Stelle `prev` entweder auf den Nachfolger von `self` verweisen, wenn ein solcher existiert, oder den Wert `NULL` erhalten, wenn es nach der Löschung das letzte Element der Liste ist.

Wir können davon ausgehen, dass unsere Liste zuvor in einem intakten Zustand war. Die Variable `next` erhält damit in Zeile 58 als Wert entweder einen Pointer auf ein Element unserer Liste, oder den Wert `NULL`, falls `self` auf das Ende der Liste zeigt. Wir müssen also keine Fallunterscheidung für Mitte und Ende der Liste machen.

Allerdings kann es sein, dass das erste Element der Liste gelöscht wird, und daher `prev == NULL`. Nur wenn dies nicht der Fall ist, kann ein Pointer aktualisiert werden – wir schreiben in Zeile 59 ein `if`.

Im vorigen Kapitel bei der `stringlist` sind wir auf das Problem gestoßen, dass der Speicherplatz für die Nutzdaten (dort also für die einzelnen Strings) von den Routinen der Stringlist komplett verwaltet werden sollte. Wir wollen hier einen ähnlichen Weg gehen, jedoch ein zusätzliches Szenario bedenken:

Es ist möglich, dass wir in unserem Projekt mehr als eine `LinkedList` benutzen. Die beiden Listen könnten sich Elemente teilen, d. h. dasselbe Element kommt in zwei Listen vor. (Denken Sie beispielsweise an die Verwaltung einer Firma. Eine Liste könnte alle Angestellten sein, eine speichert alle Angestellten, die in der Abteilung IT arbeiten). Wenn nun die erste Liste freigegeben wird (z. B. die Liste *aller* Angestellten), so wird auch der Speicher aller Nutzdaten freigegeben – selbst der Speicher, der in der zweiten Liste (die der Angestellten in der IT) registrierten Nutzdaten.

Um dieses Problem zu umgehen, haben wir den „Schalter“ `memoryAutoManaged` in `linkedList_t` eingebaut. Nutzdaten werden nur freigegeben, wenn die aktuelle Liste die Nutzdaten auch „besitzen“ soll. In Zeile 61 wird daher nur für diesen Fall der Speicher der Nutzdaten freigegeben.

Da nun alle Pointer aktualisiert sind, können wir endlich auch den Speicher des Listenelements freigeben (Zeile 62) und weiter unserer Verwaltungs-Variablen mitteilen, dass sich die Länge unserer Liste geändert hat (Zeile 64).

Eine Liste hat immer eine nicht-negative Länge. Wir nutzen diese Information, und geben als Rückgabewert entweder die neue Länge der Liste zurück, wenn die Löschung erfolgreich war, oder einen negativen Wert, um anzudeuten, dass ein Fehler aufgetreten ist.

Betrachten wir nun Einfügungen in die Liste:

Einfügen in eine Linked List

```
66  int add_to_linkedList(  
67      linkedList_t * list, int index,  
68      void * newData, size_t bytes  
69  ) {  
70      if (index < 0 || index > list->size) {  
71          printf("Fehler: ungültiger Index\n");  
72          return -1;  
73      }
```

```

74     listElement_t * prev, * self, * next;
75
76     self = malloc(sizeof(*self));
77     if (!self) {
78         printf("Fehler: Speicher konnte nicht alloziert werden\n");
79         return -1;
80     }
81
82     if (list->memoryAutoManaged) {
83         self->data = malloc(bytes);
84         if (!self->data) {
85             printf("Fehler: Speicher konnte nicht alloziert werden\n");
86             free(self);
87             return -1;
88         }
89         memcpy(self->data, newData, bytes);
90
91     } else {
92         self->data = newData;
93     }
94
95     if (index == 0) {
96         prev = NULL;
97         next = list->first;
98         list->first = self;
99
100    } else {
101        prev = get_linkedList_element(list, index - 1);
102        next = prev->next;
103    }
104
105    if (prev) {prev->next = self;}
106    self->next = next;
107
108    return ++list->size;
109 }

```

Der Test, ob der übergebene `index` sinnvoll ist, sieht fast genauso aus wie schon in der Routine zum Löschen; jedoch erlauben wir hier Werte bis *einschließlich* der Länge der Liste, da es auch möglich sein soll, Werte an ihr Ende anzuhängen.

Wir reservieren also Speicher für unser Listenelement (Zeile 76). Für die tatsächlichen Nutzdaten betrachten wir wieder den Status des „Schalters“ `memoryAutoManaged`.

Wenn unsere Liste die ihr zugeordneten Elemente *komplett selbst verwalten soll*, legen wir eine Kopie der übergebenen Nutzdaten (`newData`) an, und verhindern so, dass Daten doppelt freigegeben werden (Zeilen 82 bis 89). Da dies bei großen Nutzdaten jedoch viel Zeit beanspruchen kann, erlauben wir auch, dass lediglich die Speicheradresse der Nutzdaten in unsere Liste übernommen wird, ohne eine Kopie anzulegen (Zeile 92).

In Zeile 95 treffen wir wieder auf den Sonderfall, dass an den Listenanfang eingefügt werden soll. Wir notieren, dass es keinen Vorgänger gibt, der aktualisiert werden muss, und merken uns den Nachfolger: das vormals erste Element der Liste. Außerdem aktualisieren wir den Einstiegspunkt.

Für den allgemeinen Fall dagegen machen wir einfach das Listenelement vor dem Einfügapunkt ausfindig (Zeile 101). Sein Nachfolger ist wieder entweder ein Pointer auf ein Listenelement oder der Wert NULL falls `index` auf das Listenelement verweist – also für alle denkbaren Szenarios der richtige Wert. Den so gefundenen Nachfolger `next` weisen wir dann als Nachfolger von `self` zu (Zeile 106), und aktualisieren auch den Vorgänger, falls ein solcher existiert (Zeile 105).

17.3.5. Ausgeben der gesamten Liste

Als Beispiel für die Aufgaben, die mit einer Liste erledigt werden müssen wollen wir alle Elemente auf dem Bildschirm ausgeben. Wir wissen zwar nicht, welche Struktur ein einzelnes Listenelement hat (unsere Nutzdaten sind ein `void *`), aber wir kennen eine Funktion, die mit diesen Daten umzugehen weiß – die `printElement` aus `linkedList_t`! Damit gestaltet sich diese Aufgabe als sehr leicht:

Alle Listenelemente ausgeben

```
110 void print_all_from_linkedlist(linkedList_t * list) {
111     listElement_t * element = list->first;
112     while (element != NULL) {
113         list->printElement(element->data);
114         element = element->next;
115     }
116 }
```

17.3.6. Der Komplette Header

In den vorigen Abschnitten haben wir den Code unserer Bibliothek kennen gelernt. Damit die Routinen in unserem eigentlichen Projekt zur Verfügung stehen, müssen wir den Header um die Deklaration der Routinen ergänzen. Vollständig lautet der Header also:

Kompletter Header `linkedList.h`

```
1  #include <stdlib.h>
2
3  typedef struct listElement_struct {
4      void *          data;
5      struct listElement_struct * next;
6  } listElement_t;
7
8  typedef struct {
9      listElement_t * first;
10     int             size;
11
12     int             memoryAutoManaged;
13     void (*printElement)(void *);
14 } linkedList_t;
```



```

15 linkedList_t * make_linkedList ();
16 void          free_linkedList(linkedList_t * list);
17
18 listElement_t * get_linkedList_element(linkedList_t * list, int index);
19
20 int delete_from_linkedList(linkedList_t * list, int index);
21 int add_to_linkedList(
22     linkedList_t * list, int index,
23     void * newData, size_t bytes
24 );
25
26 void print_all_from_linkedlist(linkedList_t * list);

```

Aller weiterer Code, wie Sie ihn gerade gesehen haben, ist im Modul `linkedlist.c` zu finden.

17.3.7. Anwendungsbeispiel

Wir haben nun die wichtigsten Routinen für die Arbeit mit einer Linked List fertig gestellt. Natürlich kann man den Funktionsumfang noch beliebig erweitern (Durchsuchen, Sortieren, Werte tauschen, ...). Hier wollen wir uns aber darauf beschränken, die gezeigten Routinen in Anwendung zu sehen.

Wir legen eine Liste aus zwei Arbeitnehmern einer Firma an. Wir halten den Schalter `memoryAutoManaged` zunächst auf `false`, um unnötiges Kopieren zu verhindern. Nach dem Befüllen stellen wir den Schalter auf `true`, um eine automatische Freigabe des Speichers für die gesamte Liste zu erreichen.

Anwendungsbeispiel: Angestellte in einer Linked List

```

1  #include <stdio.h>
2  #include <string.h>
3
4  #include "linkedlist.h"
5
6  // ===== //
7
8  typedef struct {
9      char  firstName [20];
10     char  lastName  [20];
11     char  department[20];
12     double salary;
13 } employee_t;
14
15 void printEmployee(void * employee) {
16     printf("%20s | %20s | %20s | %8.2lf\n",
17         (employee_t *) employee->firstName,
18         (employee_t *) employee->lastName,
19         (employee_t *) employee->department,
20         (employee_t *) employee->salary
21     );
22 }
23
24 // ===== //

```

```

25  int main () {
26      linkedList_t * list = make_linkedList();
27
28      list->memoryAutoManaged = 0;
29      list->printElement = printEmployee;
30
31      employee_t * emp = malloc(sizeof(*emp));
32      strcpy(emp->firstName , "Vanessa");
33      strcpy(emp->lastName  , "Schmetterling");
34      strcpy(emp->department, "Betriebsarzt");
35      emp->salary          = 7000.0;
36      add_to_linkedList(list, 0, emp, 0);
37
38      emp = malloc(sizeof(*emp));
39      strcpy(emp->firstName , "Rebecka");
40      strcpy(emp->lastName  , "Rein");
41      strcpy(emp->department, "R&D");
42      emp->salary          = 420.69;
43      add_to_linkedList(list, 0, emp, 0);
44
45      printf(
46          "%20s | %20s | %20s | %8s\n",
47          "Vorname",
48          "Nachname",
49          "Abteilung",
50          "Lohn"
51      );
52      printf(
53          "-----+-----+-----+-----\n"
54          "-----+-----+-----+-----\n"
55          "-----+-----+-----+-----\n"
56          "-----\n"
57      );
58
59      print_all_from_linkedlist(list);
60
61      list->memoryAutoManaged = 1;
62      free_linkedList(list);
63  }

```

Ausgabebeispiel: Angestellte in einer Linked List

| Vorname | Nachname | Abteilung | Lohn |
|-------------------------|---------------|--------------|---------|
| -----+-----+-----+----- | | | |
| Rebecka | Rein | R&D | 420.69 |
| Vanessa | Schmetterling | Betriebsarzt | 7000.00 |

17.4. Einsatzbereiche der Linked List: Vor- und Nachteile

Der Verwaltungsaufwand von LinkedListen ist größer als der von einfachen Arrays. Zugriffe auf einzelne Elemente erfordern bereits Funktionsaufrufe, und die ganze Struktur macht komplexe Überlegungen zum Speicher-Management nötig. Wann also ist dieser Aufwand gerechtfertigt?

Wir hatten gesehen, dass das Einfügen von Werten in die Mitte der Liste und insbesondere an den Listenanfang besonders einfach geht, da kein Kopieren oder Verschieben von Werten notwendig wird. Solche Aufgaben, bei denen also häufig Elemente in Listen eingeschoben und daraus entfernt werden, können sehr effizient mit Linked Lists erledigt werden.

Das *Iterieren* über *alle* Elemente der Liste ist mit vertretbarem Aufwand zu meistern und hat kein bedeutend schlechteres Laufzeitverhalten als das Bearbeiten aller Elemente eines einfachen Arrays. Im einen Fall springt man jeweils von einem Listenelement zu seinem Nachfolger; im anderen Fall erhöht man einen Index.

Wo *wahlfreier Zugriff* gefordert ist, d. h. wo die Reihenfolge der Indizes, auf die man zugreift, nicht der Reihenfolge der Liste entspricht, ist das Verhalten der Linked List klar schlechter als das von Arrays.

Es hängt also vom Anwendungsfall ab: Welche Arten von Aufgaben sollen mit der Liste erledigt werden? Wie wird auf die Listenelemente zugegriffen? Abhängig von der Antwort auf diese Fragen kann die Wahl auf klassische Arrays, Linked Lists oder ganz andere Speicherstrukturen fallen.

Der Vorgeze-Anwendungsfall für Linked Lists sind sogenannte *FIFOs* (englisch: *first in, first out*). Stellen Sie sich einen Stapel vor. Sie können oben auf den Stapel Elemente (Aufgaben) legen; diese werden dann in der umgekehrten Reihenfolge bearbeitet, in der sie auf den Stapel gelegt wurden, d. h. das zuletzt eingefügte Element wird als erstes vom Stapel genommen.

In der Vorlesung *Algorithmen & Datenstrukturen* der Universität Regensburg werden Sie hierzu mehr Details erfahren und auch Strukturen kennen lernen, die auf den Ideen der Linked List aufbauen. Diese Strukturen können zusätzliche Funktionalität bieten (z. B. durchsuchbare Bäume) oder für bestimmte Anwendungsfälle optimiert sein (z. B. *priority queues*, die besonders schnellen Zugriff auf das größte Element einer Liste erlauben.)

Linked Lists und andere Speicherstrukturen gehören zum Standard-Repertoire des Programmierens. Während Sie die zugrunde liegenden Konzepte verstehen und umsetzen können sollten, werden Sie in der praktischen Anwendung i. d. R. keine Linked Lists selbst schreiben, sondern vorgefertigte Lösungen anderer KollegInnen auf Ihre Projekte anwenden. Beispielsweise können Sie in C die SGLIB verwenden. Siehe hierzu <http://sglib.sourceforge.net/>.

Die Containers-Library in C++

Der letzte Absatz gilt umso mehr für C++: Die Aufgabe, Daten effizient im Speicher zu behalten und Schnittstellen für Zugriffe bestimmter Art zu implementieren wurde in der STL (Standard Library) für eine Reihe von Szenarios sehr effizient umgesetzt und kann von jedem/jeder ProgrammiererIn genutzt werden.

Linked Lists stehen dort in der Klasse `std::queue` bzw. als `std::forward_list` zur Verfügung. Verwandt damit ist der `std::stack` (LIFO, also *last in, first out*) und die `std::deque` (double-ended queue).

Siehe dazu <https://en.cppreference.com/w/cpp/container>

18. Ausblicke: C++

C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.

Bjarne Stroustrup

keine grafische Hervorhebung von C++-Themen in diesem Kapitel

Wie der Titel verspricht, behandelt dieses gesamte Kapitel Inhalte zur Sprache C++. Auf eine grafische Hervorhebung wird hier daher verzichtet.

Einer der größten Vorteile der Sprache C ist seine Nähe zur Hardware, die es erlaubt, extrem effiziente Programme zu erstellen. Einer der größte Nachteile der Sprache C ist ebenfalls diese Hardware-Nähe, zwingt sie doch dazu, in abstrakten und kleinschrittigen Konzepten zu denken.

Die Sprache C++ wurde geschaffen, um die Effizienz von C so weit wie möglich zu erhalten, dabei aber menschnähere Denkweise in das Programmieren einzuführen. Vielfach wiederkehrende Aufgaben sind in der STL (C++ Standard Library) in allgemeiner Form gelöst und können ohne weiteren Aufwand in eigene Projekte eingefügt werden¹.

Syntaktisch sind sich C und C++ sehr ähnlich. Je nach gewähltem Compiler lässt sich C-Code mit einem C++-Compiler direkt umsetzen oder bedarf nur kleinerer Änderungen, um ein *funktionierendes* Programm zu erzeugen. Dies soll aber nicht den Eindruck erwecken, C++ sei nur eine Variation der Sprache C. Vielmehr handelt es sich um eine komplette Neuentwicklung. Während C-Code von einem C++-Compiler zu einem *funktionierenden* ausführbaren Programm umgesetzt werden kann, sieht *guter* C++-Code bedeutend anders aus.

C++ ist eine eigenständige Sprache

Einsteiger kommen ob der syntaktischen Nähe oft zu dem Schluss, dass C und C++ quasi dieselbe Sprache seien. Infolge dessen werden Code-Konzepte gemischt, was im besten Fall nur zu Performance-Einbußen oder eingeschränkter Kompatibilität des Codes mit verschiedenen Plattformen führt, im schlimmsten Fall aber zu instabilem Code und schwer auffindbaren Fehlern. Wenn Sie die Sprache C++ erlernen wollen – wozu ich ihnen stark rate – behalten Sie immer im Kopf, dass es sich um eine eigenständige Sprache handelt. Benutzen Sie Nachschlagewerke oder fragen Sie Dozenten und erfahrene ProgrammiererInnen in Ihrem Bekanntenkreis, ob sich eine Ihnen aus C bekannte Technik problemlos nach C++ übersetzen lässt.

Aus Gründen des Umfangs kann hier nur ein „Teaser“ für die Features der Sprache C++ gegeben werden. Eine tiefere Einsicht bietet der Kurs *C++ und Qt* der Universität Regensburg. Für das Selbststudium ist das Tutorial (in englischer Sprache) unter <http://www.cplusplus.com/doc/tutorial/> sehr gut

¹Der Name C++ leitet sich vom Inkrement-Operator ++ der Sprache C ab: C++ ist eine *Aufwertung* der Sprache C. Einem ähnlichen Schema folgt auch die Benennung der Sprache C# (sprich: C-sharp). Diese Weiterentwicklung von C++ sollte ursprünglich mit vier Pluszeichen geschrieben werden, eben als *Aufwertung* von C++. Da dies aber in der Praxis zu umständlich wäre, ordnete man die Pluszeichen so um, dass sie das Raute-Symbol # ergeben.

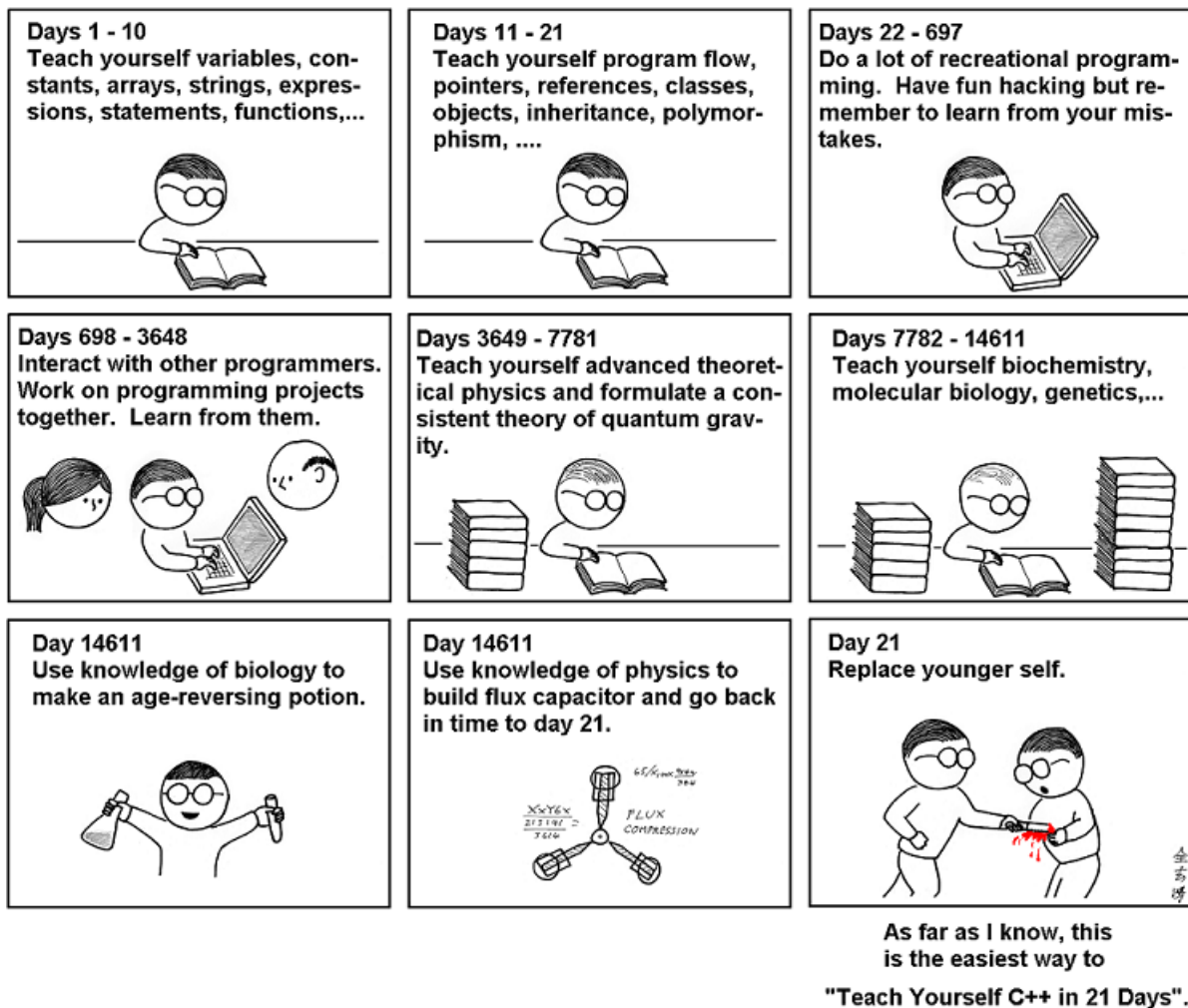


Abbildung 18.1.: Teach Yourself C++ in 21 days – ein Praxisbericht.
Quelle: <https://abstrusegoose.com/249>

geeignet. Selbstverständlich ersetzt dieser Kurz-Kurs nicht die Praxis-Erfahrung (vgl. Abbildung 18.1), die Sie sich hoffentlich bald aneignen – sowohl in C wie auch in C++.

18.1. Default Arguments und Function Overloading

In C++ ist es möglich, Funktionen, die Parameter erwarten, Standardwerte vorzugeben. Beim Aufruf kann man dann diesen Parameter auslassen. Im Prototypen der Funktion wird dazu einfach hinter der Deklaration des Parameters ein = Wert angefügt. In der Kopfzeile der eigentlichen Funktion wird dies dann nicht mehr wiederholt.

Beispiel: Funktion mit Default-Parameter

```
1  #include <iostream>
2
3  void showANumber(int number = 42);
4
5  void showANumber(int number) {
6      std::cout << number << std::endl;
7  }
8
9  int main() {
10     showANumber();
11     showANumber(666);
12 }
```

Ausgabebeispiel: Funktion mit Default-Parameter

```
42
666
```

Fakultative (notwendige) und optionale Parameter (also solche, für die ein Standardwert vorgegeben wurde) können gemischt vorkommen; die optionalen Parameter müssen allerdings am Ende der Deklaration stehen.

Daneben ist es möglich, mehrere Funktionen mit demselben Bezeichner anzulegen, solange sich diese in der Signatur unterscheiden. Beispielsweise kann sich der Rückgabotyp unterscheiden oder die Zahl und Art der Parameter verschieden sein. Wir nennen dies *Function Overloading*

Im Folgenden Beispiel benutzen wir Function Overloading, um mit einem Symbol Matrizen verschiedenen Datentyps (**float** oder **double**) anzulegen. Die Befehle **new** und **delete** sind für Sie noch neu, entsprechen in ihrer Funktion aber den Ihnen schon bekannten Befehlen **malloc** und **free**:

Beispiel: Überladene Funktion

```
1  typedef struct {
2      int    rows;
3      int    cols;
4      float* data;
5  } matrixFloat;
6
7  typedef struct {
8      int    rows;
9      int    cols;
10     double* data;
11 } matrixDouble;
```

```

12 matrixFloat initMatrix(int rows, int cols) {
13     matrixFloat reVal = {-1, -1, nullptr};
14
15     if (rows < 1 || cols < 1) {
16         std::cout << "Unzulässige Matrix-Dimension" << std::endl;
17         return reVal;
18     }
19
20     float * data = new float[rows * cols];
21     if (!data) {
22         std::cout << "Fehler beim Allozieren" << std::endl;
23         return reVal;
24     }
25
26     reVal.rows = rows;
27     reVal.cols = cols;
28     reVal.data = data;
29
30     return reVal;
31 }
32
33 matrixDouble initMatrix(int rows, int cols) {
34     matrixDouble reVal = {-1, -1, nullptr};
35
36     if (rows < 1 || cols < 1) {
37         std::cout << "Unzulässige Matrix-Dimension" << std::endl;
38         return reVal;
39     }
40
41     double * data = new double[rows * cols];
42     if (!data) {
43         std::cout << "Fehler beim Allozieren" << std::endl;
44         return reVal;
45     }
46
47     reVal.rows = rows;
48     reVal.cols = cols;
49     reVal.data = data;
50
51     return reVal;
52 }
53
54 int main() {
55     matrixFloat mf = initMatrix(3, 3);
56     matrixDouble md = initMatrix(3, 3);
57
58     delete mf.data;
59     delete md.data;
60 }

```

18.2. Templates

Im obigen Beispiel gewinnen wir als ProgrammierInnen zwar in der `main` Klarheit des Codes, weil wir das Symbol `initMatrix` benutzen können, ohne uns weiter Gedanken über den Datentyp der erstellten Matrix machen zu müssen. Jedoch zwingt uns diese Struktur, Code doppelt zu schreiben: Bis auf die Allokierung von Speicher sind die Codes für die `float`- und `double`-Variante quasi gleich. Die ist schlecht, da es in der Weiterentwicklung unseres Codes leicht ist zu vergessen, Änderungen in einer Funktion auf ihr Pendant zu übertragen.

C++ erlaubt es daher, Funktionen für einen allgemeinen Datentypen zu schreiben. Der tatsächliche Datentyp wird erst im Aufruf eingesetzt. Wir nennen diese Technik *Templates*. Über das Schlüsselwort `template` wird ein Symbol definiert, das im folgenden Scope gültig ist und einen Typ beschreibt. Aufrufe einer Funktion mit einem Template-Typ „erzeugen“ dann erst den tatsächlichen Code. Ähnlich wie bei Präprozessoren wird also Code erst umgeschrieben, bevor dieser kompiliert wird. Durch intelligentere Mechaniken tauchen aber weit weniger unerwartete Effekte auf.

Das obige Beispiel mit Matrizen lässt sich mit Templates so umschreiben:

Beispiel: Überladene Funktion

```
1  template <typename T>
2  typedef struct {
3      int rows;
4      int cols;
5      T * data;
6  } matrix_t;
7
8  template <typename T>
9  matrix_t initMatrix(int rows, int cols) {
10     matrix_t reVal = {-1, -1, nullptr};
11
12     if (rows < 1 || cols < 1) {
13         std::cout << "Unzulässige Matrix-Dimension" << std::endl;
14         return reVal;
15     }
16
17     T * data = new T[rows * cols];
18     if (!data) {
19         std::cout << "Fehler beim Allokieren" << std::endl;
20         return reVal;
21     }
22
23     reVal.rows = rows;
24     reVal.cols = cols;
25     reVal.data = data;
26
27     return reVal;
28 }
```



```

29  int main() {
30      matrix_t<float> mf = initMatrix<float> (3, 3);
31      matrix_t<double> md = initMatrix<double>(3, 3);
32
33      delete mf.data;
34      delete md.data;
35  }

```

In den Zeilen 1 bis 6 definieren wir also eine Struktur, die Matrizen vom Typ T speichert. An dieser Stelle steht noch nicht fest, für welchen Datentyp T tatsächlich steht.

Die folgende Funktion `initMatrix` bereitet nun eine solche Matrix vor. Auch hier wird nur auf einen *generischen* Datentyp T Bezug genommen; die Einsetzung für einen real existierenden Datentyp geschieht erst später.

In Zeile 30 dann wird eine Variable `mf` vom Typ `matrix_T<float>` angelegt. Die <spitzen Klammern> sind der *Template-Parameter*, also der „Wert“, der für T eingesetzt werden soll. An dieser Stelle wird also aus der Vorlage in den Zeilen 1 bis 6 ein `struct` generiert, in dem T durch `float` ersetzt wird. In der folgenden Zeile 31 passiert dasselbe für den Datentyp `double`. Mit einfacher Tipparbeit sind also zwei Datentypen entstanden, die sich ein gemeinsames Gerüst teilen.

Dasselbe passiert in den Zeilen 30 und 31 für die Funktion `initMatrix`: Durch den Template-Parameter wird eine Variante der Funktion für `floats` und eine Variante für `doubles` erzeugt².

18.3. Klassen und Objektorientierung

In C++ folgt man der Philosophie, gedankliche Einheiten in *Objekten* abzubilden, die bestimmten *Klassen* angehören. Eine Klasse ist eine Gruppe von Variablen zusammen mit *Methoden*, über die mit diesem Objekt gearbeitet werden kann. Eine Klasse ist also eine Erweiterung der Idee einer `struct` um Methoden. Hinzu kommt auch eine Einschränkung der Sichtbarkeit von Elementen der Klasse, ähnlich wie bei Scopes. Dies soll das versehentliche Ändern von Werten verhindern, die Nebeneffekte mit sich bringen und deren Veränderung einen inkonsistenten Programmzustand bewirken würden.

Objekte können zum Beispiel die oben angesprochenen Matrizen sein, oder eine Figur in einem Jump'n'Run-Spiel. Ein Codeauszug aus einem solchen Spiel könnte lauten:

Beispiel: Überladene Funktion

```

1  class character {
2  private:
3      int posX;
4      int posY;
5      int health;
6
7  public:
8      void moveLeft();
9      void moveRight();
10     void jump();
11 }

```

²Bei den Funktionsaufrufen kann der Template-Parameter sogar entfallen. Da der Wert einer entsprechenden Variable zugeordnet werden soll, ist eindeutig vorgegeben, für welchen Zieltyp das Template umgesetzt werden soll.

```

12 // Definition der Funktionen moveLeft, etc.
13
14 int main () {
15     // ... Code ...
16     character x;
17
18     if (keyboard == right) {x.moveRight();}
19 }

```

Das Schlüsselwort **private**: sperrt den Zugriff auf alle nachfolgenden Elemente der Klasse, also hier auf die Felder **posX**, **posY** und **health**. Nur Methoden derselben Klasse – hier also **moveLeft**, **moveRight** und **jump** dürfen diese lesen oder verändern. Das Schlüsselwort **public**: hebt diese Einschränkung wieder auf: Alle nachfolgenden Deklarationen sind von jeder Stelle aus „sichtbar“.

Auf eine ähnliche Art können *Constructors* und *Destructors* definiert werden, d.h. Methoden, die automatisch aufgerufen werden, sobald ein Objekt vom Typ der Klasse erstellt wird, bzw. sobald die Lebensdauer dieses Objekts endet. Damit muss man sich also nur einmal beim Erstellen der Klasse Gedanken um das Speichermanagement machen, und kann weiter in der Anwendung darauf vertrauen, dass diese Aufgabe bereits gelöst ist.

18.4. Überladene Operatoren

In Kapitel 10 hatten wir festgestellt, dass für **structs** nur die Wertzuweisung (=) definiert ist. Alle anderen Operationen mussten dort über explizite Funktionsaufrufe erledigt werden.

In C++ ist es möglich, das Verhalten von Operatoren selbst zu definieren und damit auch neue Datentypen abzudecken. Das Verhalten wird in einer Funktion mit einer speziellen Kopfzeile definiert. Der „Aufruf“ der Funktion geschieht dann über das Operatorsymbol.

Die Addition zweier Vektoren ließe sich in C++ beispielsweise so realisieren:

Beispiel: Überladene Operatoren

```

1  typedef struct {
2      double x;
3      double y;
4  } vector2d_t;
5
6  vector2d_t operator + (vector2d_t LHS, vector2d_t RHS) {
7      vector2d_t reVal = {LHS.x + RHS.x, LHS.y + RHS.y};
8      return reVal;
9  }
10
11 int main {
12     vector2d_t a = {42, 666}, b = {-420, 3.14};
13     vector2d_t c = a + b;
14
15 }

```

18.5. Strings in C++

Die aufwändigen und wiederkehrenden Aufgaben, die beim Umgang mit Texten in C auftreten, sind in C++ gesammelt und in der Klasse `std::string` abgehandelt. Mit dieser sind weiterhin die von C bekannten Techniken möglich.

Als *Klasse* stehen für Strings diverse *Methoden* zur Verfügung, die über die oben gezeigte Syntax `object.method()` aufgerufen werden. Zur Verfügung stehen unter anderem Methoden zur Bestimmung der Länge, zur Verkettung oder zum Finden von Zeichenketten im String.

Beispiel: Die `string`-Klasse

```
1  #include <iostream>
2  #include <string>
3
4  int main () {
5      std::string
6          firstHalf = "Ich bin aus",
7          lastHalf  = " der Hölle",
8          total;
9
10     std::cout << firstHalf + lastHalf << std::endl;
11     std::cout << (firstHalf + lastHalf).length() << std::endl;
12
13     total = firstHalf + lastHalf;
14     total.replace( total.find("aus"), 4, "Frau");
15     total.replace( total.find("der"), 3, "");
16     total[ total.find("ö") ] = 'o';
17     std::cout << total << std::endl;
18
19     firstHalf.insert( firstHalf.find("a"), "r" );
20
21     std::cout << firstHalf << std::endl;
22 }
```

Ausgabebeispiel: Die `string`-Klasse

```
Ich bin aus der Hölle
22
Ich bin Frau Holle
Ich bin raus
```

Alle Aufgaben des Speicher-Managements werden von der internen Mechanik der Klasse `string` automatisch übernommen.

Siehe auch https://en.cppreference.com/w/cpp/string/basic_string für weitere Details zur Klasse.

18.6. Die Container-Library

Neben Strings sind in der STL auch andere Klassen vorgefertigt, die die Arbeit mit größeren Datenmengen erleichtern. Listen beliebigen Datentyps können mit Hilfe dieser Klassen bequem angelegt, vergrößert und

verkleinert werden. Wie schon bei Strings geschieht die Speicherverwaltung automatisch. Für verschiedene Anwendungsfälle stehen verschiedene Klassen zur Verfügung. Sie unterscheiden sich im Wesentlichen darin, welche Aufgaben die Methoden optimiert wurden. Für die meisten Anwendungsfälle ist die Klasse `std::vector` gut geeignet.

Beispiel: Die `vector`-Klasse

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4
5  int main () {
6      std::vector<int> list;
7
8      // make a list of numbers
9      for (int i=0; i<20; i++) {list.push_back((i - 3) * (i - 1) * (i - 6));}
10
11     // output on screen
12     for (int i=0; i<20; i++) {std::cout << list[i] << " ";}
13     std::cout << std::endl;
14
15     // sort in ascending order
16     std::sort(list.begin(), list.end());
17
18     // output on screen
19     for (int i=0; i<20; i++) {std::cout << list[i] << " ";}
20     std::cout << std::endl;
21 }
```

Ausgabebeispiel: Die `vector`-Klasse

```
-18 0 4 0 -6 -8 0 24 70 144 252 400 594 840 1144 1512 1950 2464 3060 3744
-18 -8 -6 0 0 0 4 24 70 144 252 400 594 840 1144 1512 1950 2464 3060 3744
```

Ähnlich funktionieren die folgenden Klassen:

`std::array` Wird auf dem Stack angelegt und kann nicht mehr in der Größe verändert werden. Viele Operationen laufen mit dieser Klasse schneller. (<https://en.cppreference.com/w/cpp/container/array>)

`std::list` Intern als Linked List umgesetzt. Einfügen an den Anfang und entfernen vom Anfang der Liste ist besonders schnell, ebenso ist Sortieren effizient programmiert. Diese Optimierung geht auf Kosten der Lesezeit für Elemente in der Mitte der Liste (<https://en.cppreference.com/w/cpp/container/list>)

`std::map` Implementiert ein „dictionary“, also eine Zuordnung von Wertepaaren zueinander. Anstelle eines Index wird also beispielsweise ein Wort als Schlüssel verwendet. (<https://en.cppreference.com/w/cpp/container/map>)

Die Mechaniken hinter diesen Klassen werden z. T. in der Vorlesung *Algorithmen und Datenstrukturen* an der Universität Regensburg besprochen.

Appendices

A. Begriffe

A...

Adresse „Nummer des Bytes im Speicher“, ab dem eine Information zu finden ist.
Siehe Abschnitt 3.1.2

Adress-Operator Das Zeichen `&`. Wird vor Variablen gestellt, um den Speicherort (d. h. die *Adresse*) der Variable zu ermitteln.
Siehe Abschnitt 3.1.2

Allozieren Anfrage an das Betriebssystem, einen Speicherbereich für andere Programme zu sperren, also den Speicherbereich für eine bestimmte Aufgabe zu *reservieren*. Üblicherweise mit den Befehlen `malloc` oder `calloc` aus dem Header `<stdlib.h>` umgesetzt.
Siehe Abschnitt 8.2.1

Alphanumerisch Schriftzeichen, die für menschliche Worte (im Englischen) üblich sind, sowie Ziffern: A-Z, a-z, 0-9.

Argument Wert, der einer *Funktion* in runden Klammern `()` *übergeben* wird. Die Funktion kann nun mit diesem Wert arbeiten. Argumente können *Konstanten*, *Variablen* oder komplexe *Ausdrücke* sein. Beispiel: Die Funktion `sin` erwartet ein *Argument* vom Datentyp `double`. Damit können als Argumente z. B. `0`, `x` oder `3.14 * x + acos(0)` möglich.
Siehe Abschnitt 9.2

Array Eine Liste von Werten, die im Speicher „hintereinander“ abgelegt wird. Zugriff auf die Elemente des Arrays geschieht über einen *Pointer* auf den Listenanfang und einen *Index*.
Siehe Abschnitt 8.1

Assembler Eine Programmiersprache und auch der Name einer Art von Programmen.
Die Befehle der Sprache Assembler entsprechen dem (sehr kleinen) Befehlssatz eines Prozessors. Daher ist in dieser Sprache eine sehr hardwarenahe, kleinschrittige Denkweise notwendig. *Hochsprachen* erlauben größere Abstraktion und dadurch eine für Menschen natürlichere Denkweise. Oft werden Hochsprachen-Programme erst in die Sprache Assembler übersetzt und dann von einem Assembler-Programm in *Maschinensprache* weiter übersetzt.
Siehe Abschnitt 1.2

Ausdruck Eine Folge von *Konstanten*, *Variablen* und *Funktionsaufrufen*, die durch *Operatoren* so miteinander verknüpft sind, dass daraus ein Wert berechnet werden kann.
Beispiele (jeweils durch Semikolons getrennt): `x + 9`; `42`; `-sqrt(5.3) * 7 - y`; `x > y`.
Siehe Abschnitt 2.2

Ausschlussset-Schreibweise Teil eines *Format-Strings* bei `scanf`, der bei der Eingabe von *Strings* benutzt wird. Form `[^...]`, wobei `...` für die Zeichen steht, die *nicht* beachtet werden sollen.
Siehe Abschnitt 8.3.2

Automatische Arrays *Arrays*, die vom Compiler verwaltet werden, d. h. für die kein Speicher mit `malloc` *alloziert* werden muss, und die nicht mit `free` freigegeben werden dürfen. Bei der *Deklaration* wird in [eckige Klammern] die Arraygröße in Elementen angegeben (Ausnahme: *Initializer-List*). Einmal angelegt kann die Größe von automatischen Arrays nicht mehr geändert werden. Siehe

daher auch *Dynamische Arrays*, die zur *Laufzeit* die Größe ändern können
Siehe Abschnitt 8.1

B...

Bibliothek Eine Sammlung von *Routinen*, die für sich genommen kein alleinstehendes Programm ergeben, die aber Probleme lösen, welche in anderen Programmen (mehrfach) auftreten können. Üblicherweise stehen Bibliotheken als *vorkompilierte* Datenobjekte zur Verfügung. Sie werden über einen *Header* in den eigenen Code eingebunden. Beim *kompilieren* muss gegen die Bibliothek *gelinkt* werden.

Siehe Abschnitt 1.3 und Kapitel 11

Binärformat Darstellung von Werten in einem Format, das von Computern leicht und effizient verarbeitet werden kann, das für Menschen dagegen i. d. R. nicht entzifferbar ist. Dateien in diesem Format werden mit **fwrite** geschrieben und mit **fread** gelesen. Alle Werte im Speicher werden im Binärformat gehalten.

Siehe Abschnitt 13.3

Binärsystem Zahlensystem, das lediglich aus den Ziffern 0 und 1 besteht und die Grundlage der Rechner-Elektronik bildet.

Siehe Abschnitt 1.1

Bitweise Operatoren *Operatoren*, die nicht mit Zahlen als Einheit arbeiten, sondern die die einzelnen Bits der Zahlen getrennt voneinander verarbeiten. Die Bitweisen Operatoren werden mit den Zeichen `^`, `&`, `|` und `~` gesetzt und sind von den *logischen Operatoren* abzugrenzen.

Siehe Abschnitt 3.2

C...

Compiler Ein Programm, das die Übersetzung von (menschenslesbarer) Programmiersprache in *Maschinensprache* übernimmt. Häufig wird dabei zunächst als Zwischenschritt in *Assembler transpiliert*. Umgangssprachlich wird das *Linken* auch als Teil des Kompilier-Vorgangs aufgefasst.

Siehe Abschnitt 1.3.

D...

Dateimodi Lese- oder Schreibzugriff.

Siehe Abschnitt 13.1

Datentyp Die Art der Information, auf die über eine *Variable* oder einen *Ausdruck* zugegriffen werden kann, bzw. die einer *Konstanten* zugeordnet ist. Ein Datentyp kann eine von mehreren Arten von *Ganzzahlen*, *Fließkommazahlen* oder *Pointern* sein.

Siehe Abschnitt 2.2 sowie die Tabellen B.8 und B.9

Definition Eine Anweisung, die zwar die Bedeutung eines Symbols festlegt, aber keine *Speicherstelle* reserviert oder ändert.

Siehe *Forward declaration*, *Header*, *Präprozessor*.

Deklaration Abschnitt im Code, bei dem einem *Symbol* innerhalb seines Scopes eine *Datentyp* und eine *Speicherstelle* zugewiesen werden (letzteres geschieht dabei automatisch).

Siehe Abschnitte 2.2.2 und 9.2

Dereferenzierung Anweisung an den Compiler, nicht mit einem *Pointer* zu arbeiten, sondern mit dem Wert im Speicher, der durch den Pointer referenziert wird. Dies geschieht entweder über den Dereferenzierungsoperator (`*pointer`) oder über den Array-Index-Zugriff (`pointer[index]`)
Siehe Abschnitte 3.1.2 und 8.1.

Dualsystem Synonym für *Binärsystem*

Dynamische Arrays *Arrays*, die vom/von der ProgrammiererIn verwaltet werden, d. h. für die Speicher mit `malloc` *alloziert* werden muss, und die nicht mit `free` freigegeben werden müssen. Bei der *Deklaration* wird das Symbol `*` benutzt, um das zugeordnete Symbol als *Pointer* zu markieren. *Dynamische Arrays* können zur *Laufzeit* die Größe ändern, sind aber aufwändiger zu verwalten als *Automatische Arrays*
Siehe Abschnitt 8.1

E...

Errorcode Eine *Ganzzahl* zwischen 0 und 255, die eine Information darüber enthält, welche Fehler bei der Ausführung eines Programms aufgetreten sind.
Siehe Abschnitt 12.1

Escape-Char, Escape-Sequenz Ein Zeichen, das angibt, dass das oder die folgenden Zeichen eines Textes nicht eins zu eins umgesetzt werden sollen, sondern durch andere, schwer einzugebende Zeichen ersetzt werden sollen (z. B. Zeilenumbruch). In C ist das Escape-Char der Backslash `\`. Escape-Char und *Escape-Sequenz* werden zusammen durch eine Entsprechung ersetzt.
Siehe Tabelle B.1

Expansion Der Code, der vom *Präprozessor* erzeugt wird, nachdem ein *Macro* durch seinen Macrotext ersetzt wurde.
Siehe Kapitel 14

F...

Fließkommazahl Eine Information, die eine Zahl mit Nachkomma-Anteil beschreibt, also z. B. `3.14` oder auch `42.0`.
Siehe Abschnitt 2.2.3

Format-String Eine Zeichenkette, die mit `printf`, `scanf` und den damit verwandeten Funktionen (`sprintf`, `fprintf`, ...) verwendet wird.
Ein Format-String enthält Steuerzeichenketten, die über Datentyp und Format von Werten informieren. Diese Zeichenketten beginnen in der Regel mit einem Prozentsymbol `%` und können aus einem oder mehreren weiteren Zeichen bestehen.
Siehe Tabellen B.2, B.3 und B.4

Forward Declaration Eine „unvollständige *Deklaration*“, die notwendig ist, um Zirkelbezüge aufzulösen. Diese Technik kann mit *Funktionen* und `structs` angewandt werden.
Siehe Abschnitt 9.2.2 und 10.1.2

Function-Body Die eigentlichen Anweisungen, wie sie bei *Funktionsaufruf* ausgeführt werden, eingefasst von {geschweiften Klammern}.
Siehe Abschnitt 9.2

Funktion, Function Ein Codeabschnitt, der einen eigentständigen *Scope* definiert und eine allgemeine Lösung für ein Problem bereitstellt. Funktionen haben einen Namen, einen *Rückgabety*p und eine *Parameterliste* (d. h. null bis mehrere *Argumente*).
Siehe Abschnitt 9.2

Funktionsaufruf Der Befehl, mit der Code-Ausführung in eine *Funktion* zu springen.
Siehe Abschnitt 9.2

Funktionssignatur Die Gesamtheit der Informationen *Rückgabety*p plus *Parameterliste*, ohne den Funktionsnamen. Man kann die Signatur einer Funktion als ihren *Datenty*p auffassen.
Siehe Abschnitt 9.2

G...

Ganzzahl Eine Information, die eine Zahl ohne Nachkomma-Anteil beschreibt, also z. B. 3 oder auch -42.
Siehe Abschnitt 2.2.3

H...

Header Eine Code-Datei, die lediglich *Definitionen* enthält: *Funktions-Signaturen* ohne *Function-Body*, **structs**, **enums** und **typedefs** sowie *Präprozessor-Direktiven*.
Siehe Abschnitt 9.2, und Kapitel 10, sowie 14

Heap (Großer) Speicherbereich, der dynamisch verwaltet werden kann, d. h. in dem u. a. *dynamische Arrays* angelegt werden.
Siehe Abschnitt 8.2.3

Hexadezimalsystem Zahlensystem, das aus 16 Ziffern besteht (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F). Im Programmier- und Elektronik-Umfeld erlaubt dies eine platzsparende und übersichtliche Darstellung mancher Zusammenhänge.
Siehe Abschnitt 3.5

Hochsprache Sprachen, die (gemessen am Befehlssatz eines Prozessors) komplexere Aufgaben in einem gedanklichen Schritt erledigen lassen. C ist noch sehr systemnah, zählt aber bereits als Hochsprache. Das Gegenstück bildet *Assembler* mit einer (beinahe) eins-zu-eins-Entsprechung von Befehlen der Sprache und erzeugtem *Maschinencode*.
Siehe Abschnitt 1.2

I...

Index Nummer eines Elements in einer Liste. Indices beginnen in C bei 0!
Siehe Abschnitt 8.1

Initializer-List Eine durch Kommata getrennte Liste von Werten, eingefasst in {geschweiften Klammern}. Wird zur gleichzeitigen *Deklaration* und Wertzuweisung bei *automatischen Arrays* und **structs** gebraucht.
Siehe Abschnitt 8.1.2 und 10.1.6

Instanz Eine Speicherstelle, der ein Datentyp zugeordnet ist. Dieser Datentyp kann ein *primitiver Datentyp* sein oder ein **struct**. Die Instanz ist also eine *Variable* oder ein *Array-Element*.
Siehe Kapitel 10

Iteration, Iterationsschritt Wiederholung von ähnlichen Vorgängen in direkter Folge, häufig bei Veränderung von nur einer Variablen. Synonym für *Schleifen*. Ein *Iterationsschritt* ist ein einzelner Durchlauf des *Schleifenkörpers*.
Siehe Kapitel 7

K...

Kommandozeile Text-basiertes User-Interface, in das Kommandos eingegeben werden können, die vom Betriebssystem direkt umgesetzt werden. Im Kurs benutzt zum Aufruf des *Compilers* und zum Starten unserer Programme.
Siehe Abschnitt 1.3

Kommandozeilen-Option Teil des Textes, der in die *Kommandozeile* eingegeben wird. Diese Optionen werden durch Leerzeichen vom Programmnamen und voneinander abgegrenzt, und beeinflussen den Programmablauf.
Siehe Abschnitt 12.2

Kompilieren Der Vorgang, bei dem ein menschenlesbarer Programmcode in Maschinensprache übersetzt wird. Umgangssprachlich wird das *Linken* als Teil des Kompilierens aufgefasst.
Siehe *Compiler*, *Linker*

Kompilierzeit Der Zeitpunkt, zu dem ein Programmcode in Maschinensprache übersetzt wird. Hier stehen noch nicht alle Informationen zur Verfügung, die zur *Laufzeit* gegeben sind, z. B. Usereingaben und damit Speicherbedarf. Dies macht dynamische Programmierung notwendig, also etwa *dynamische Arrays*.
Siehe Abschnitt 8.1

Konstante Ein Wert im Programm, der – auch versehentlich – nicht mehr geändert werden kann, sobald das Programm *kompiliert* wurde. Beispiele: 3, "Dune"

L...

Laufzeit Der Zeitpunkt, an dem ein Programm ausgeführt wird. Hier stehen alle Informationen zur Verfügung (Usereingaben, Speicherbedarf, ...), aber es können keine strukturellen Änderungen am Code mehr durchgeführt werden, da das Programm bereits *kompiliert* ist.
Siehe Abschnitt 8.1

Laufzeitverhalten Die Beziehung von Zeitbedarf zu Datenmenge, die ein Algorithmus verarbeitet.

Library Englisch für *Bibliothek*

Linker Ein Programm, das mehrere Dateien mit Code in *Maschinensprache* zu einer einzigen, ausführbaren Datei zusammen fügt.

Logische Operatoren Operatoren, die mit *Wahrheitswerten* operieren. In C existieren die logischen Operatoren **&&**, **||** und **!**.
Siehe Abschnitt 5.3

M...

Macro, Makro Textblöcke, die vom *Präprozessor* durch anderen Text (Code) ersetzt werden.

Siehe *Expansion*, Kapitel 14

Maschinensprache Folge von Zahlenwerten, die Anweisungen für den Prozessor enthalten, und die vom Menschen nicht mehr (mit vertretbarem Aufwand) verstanden werden können. Wird beim *kompilieren* erzeugt.

Siehe Abschnitt 1.3.

Memory Leakage Fehler in der Programmierung, bei der *allozierter Speicher* nach Programmende nicht mehr freigegeben wird.

Siehe Abschnitt 8.2.1

Menschenlesbares Format Eine Darstellung von Information in für Menschen gedachte Schriftzeichen, Insbesondere durch die ASCII-Zeichen 32 bis 127. Daten im Menschenlesbaren Format sind i. d. R. weniger effizient von Computern erfassbar als solche im *Binärformat*.

Modul Eine Code-Einheit, die üblicherweise in einer einzelnen Datei abgelegt wird, und zunächst zu einem eigenständigen Object-File umgesetzt wird. Code eines Moduls „lebt“ in einem eigenen Scope.

Siehe Kapitel 11

Modulo-Operator Berechnet den Rest einer Division. Ausgedrückt durch das Prozent-Symbol %.

Siehe Abschnitt 2.2.4

N...

NAN „not a number“. Ein Bitmuster, das bei *Fließkommazahlen* als „Fehlerwert“ gedeutet wird. Rechnungen, an denen eine NAN Anteil hat, haben zum Ergebnis wiederum eine NAN.

Null-Char Der Wert `0`, als `char` im Speicher abgelegt. Dieser Wert markiert das Ende eines *Strings*.

Siehe Abschnitt 8.3

O...

Object-File Datei bestehend aus *Maschinencode*, der aus einem einzigen *Modul* stammt.

Siehe Abschnitt 1.3

Offset Englisch: „Versatz“: Der Abstand von einem bestimmten Anfangspunkt. Meist der Abstand vom Beginn eines *Arrays* zu einer Speicherstelle, gemessen in Bytes; je nach Kontext aber auch in der Größeneinheit der Array-Elemente.

Oktalsystem Zahlensystem, das aus 8 Ziffern besteht (0 ...). Im Programmier- und Elektronik- Umfeld früher weit verbreitet. Heute nur noch im Kontext von Festplatten-Hardware üblich und weitgehend durch das *Hexadezimalsystem* abgelöst

Siehe Abschnitt 3.5

Operator Eine Rechenanweisung. Wir unterscheiden unäre Operatoren, die nur einen einzigen Wert zur Berechnung des Ergebnisses brauchen (z. B.. logisches und bitweises NOT, sowie das Vorzeichen Minus), binäre Operatoren (die zwei Zahlen brauchen, wie z. B. +, *, <<)

Siehe Abschnitte 5.3, 2.2.4

Overhead Zusätzlicher Verwaltungsaufwand (für den Prozessor), der durch die Nutzung mancher Techniken notwendig wird und häufig nicht offensichtlich ist. Beispielsweise muss bei *Funktionsaufrufen* der Codepunkt der Fortsetzung nach Funktionsende gespeichert werden, *Parameter* kopiert werden und ein Programmsprung ausgeführt werden.

P...

Parameter, Parameterliste Synonym für *Argument*. Eine Parameterliste wird in runden Klammern () gesetzt und überhält die Werte, die an eine *Funktion* übergeben werden.
Siehe Abschnitt 9.2

Parität Eigenschaft einer ganzen Zahl, gerade oder ungerade zu sein; man sagt also 1 hat ungerade Parität, während 2 gerade Parität hat.

Parsen Sammelbegriff für das Deuten eines *Strings* in einem bestimmten Kontext, also beispielsweise die Zerlegung in einzelne Worte oder die Verarbeitung eines *Format-Strings*.

Pointer Typ von Variablen, die nicht eine Information selbst, sondern ihre *Adresse* im Speicher enthalten.
Siehe Abschnitt 3.1.2

Präprozessor Ein Programm, das (C-)Code „vorverarbeitet“, indem bestimmte Text-Elemente durch andere ersetzt werden.
Siehe Kapitel 14

Präprozessor-Direktive Eine Anweisung an den *Präprozessor*, die z. B. festlegt, welche Ersetzungen vorgenommen werden sollen.

Primitiver Datentyp Datentypen, wie sie in der Sprache C ohne Zutun des/der ProgrammiererIn zur Verfügung stehen.
Siehe Tabellen B.8 und B.9

Prototyp Die erste Zeile einer *Funktion*, in der *Rückgabetyt*, Funktionsname und *Parameterliste* zu finden sind.
Siehe Abschnitt 9.2

Prozedur Synonym für *Funktion*.
Siehe Abschnitt 9.2

R...

Registerbreite Die Zahl der Bits, die für eine *Instanz* eines *Datentyps* zur Verfügung stehen muss.

Rekursion Technik, bei der eine Funktion sich selbst aufruft.
Siehe Kapitel 16

Rekursionstiefe Die „Anzahl der Verschachtelungen“ oder der Aufrufe der *rekursiven Funktion*
Siehe Kapitel 16

Routine Synonym für *Funktion*.

Rückgabetyt Der *Datentyp* des Wertes, der von einer *Funktion* berechnet wird.
Siehe Abschnitt 9.2

Rückgabewert Der Wert, der von einer *Funktion* berechnet wird.
Siehe Abschnitt 9.2

S...

Scope Codeabschnitt, innerhalb dem die Bedeutung eines *Symbols* definiert ist.

Siehe Abschnitt 9.1

Schleife Code-Struktur, die mehrere ähnliche Anweisungen in direkter Folge wiederholt, und dabei häufig nur den Wert von einer oder wenigen Variablen verändert.

Siehe Kapitel 7

Schleifenkörper Der Code in {geschweiften Klammern}, der durch die *Schleife* wiederholt werden soll.

Siehe Kapitel 7

Segfault Schwerer Fehler, der durch den falschen Umgang mit *Pointern* entsteht. Beim unbeabsichtigten Zugriff auf eine Speicherstelle werden Werte überschrieben und führen zum Programmabsturz.

Set-Schreibweise Teil eines *Format-Strings* bei `scanf`, der bei der Eingabe von *Strings* benutzt wird. Form [...], wobei ... für die Zeichen steht, die für die Eingabe erlaubt sollen.

Siehe Abschnitt 8.3.2

Speicher Freigeben Anweisung an das Betriebssystem, dass ein vorher *allozierter* Speicherbereich nicht mehr benötigt wird und wieder für andere Programme zur Verfügung steht. Wird durch den Befehl `free` umgesetzt.

Siehe Abschnitt 8.2.1

Speicher Reservieren Anweisung an das Betriebssystem, dass Speicherplatz für eine bestimmte Aufgabe benötigt wird. Das Betriebssystem findet einen zusammenhängenden, genügend großen Block im Arbeitsspeicher und sperrt diesen für andere Prozesse. Wird durch die Befehle `malloc` und `calloc` umgesetzt.

Siehe Abschnitt 8.2.1

Speicherstelle Die Bytes im Arbeitsspeicher, an denen eine Information abgelegt ist.

Signatur Die Gesamtheit der Informationen *Rückgabotyp* plus *Parameterliste*, aber ohne den Namen der Funktion.

Siehe Abschnitt 9.2

Stack (Kleiner) Speicherbereich, der automatisch verwaltet wird, d. h. in dem u. a. *automatische Arrays* sowie alle „einfachen Variablen“ angelegt werden. Auf dem Stack wird außerdem die Rücksprungsadresse bei *Funktionsaufrufen* gespeichert.

Siehe Abschnitt 8.2.3

Startadresse Die *Adresse* des ersten Werts eines *Arrays*.

Siehe Abschnitt 8.1

String Ein *Array* aus `char`-Werten, das insbesondere Text enthält. Strings in C enden mit einem *Null-char*.

Siehe Abschnitt 8.3

String-Literal Eine *String-Konstante*, also Text, eingefasst durch "doppelte Anführungszeichen"

Siehe Abschnitt 8.3

Strong-Typed Eigenschaft vieler Programmiersprachen, dass der *Datentyp* einer *Variablen* einmalig bei ihrer *Deklaration* festgelegt wird und sich danach nicht mehr ändern kann.

structs Benutzerdefinierte Datentypen, die aus mehreren *primitiven Datentypen* zusammengesetzt sind.

Siehe Kapitel 10

Symbol Eine Folge von *alphanumerischen* Zeichen sowie Unterstrichen (`_`) die ein Konzept im Code bezeichnen. Ein solches Konzept kann z. B. eine *Variable*, ein Funktionsname, ein `struct`, `enum` oder eine *Prüprozessor-Direktive* sein.

Syntaxfehler Eine Art von Programmierfehler, bei der der *Compiler* die Arbeit abbricht. In der Regel handelt es sich um Tippfehler.

T...

Transpiler Ein Programm, das Code von einer *Hochsprache* in eine andere übersetzt.

U...

Übergeben von Werten Teil eines *Funktionsaufrufs*: Parameter von der aufrufenden Stelle werden an eine *Speicherstelle* kopiert, mit der in der *Funktion* gearbeitet werden kann
Siehe Abschnitt 9.2

V...

Variable Ein *Codesymbol*, das auf eine *Speicherstelle* verweist, deren Wert sich im Ablauf des Programms ändern kann.
Siehe Abschnitt 2.2

W...

Wahrheitswert Die Information „wahr“ oder „falsch“ („true“ oder „false“). In C wird dies durch einen *Ganzzahl*-Wert ausgedrückt, der entweder gleich null (false) oder ungleich null (true) ist.
Siehe Abschnitt 5.1

Z...

Zahlensystem Die Gesamtheit an Symbolen, die als Ziffern verwendet werden. Gängig sind das Dezimalsystem und das *Hexadezimalsystem*.

B. Tabellen

B.1. Escape-Sequenzen

| Sequenz | Funktion |
|-------------------------|---|
| <code>\a</code> | <i>alert</i> – Ausgabe eines Tons über den Beeper, sofern vorhanden |
| <code>\b</code> | <i>backspace</i> – Textcursor um eine Spalte nach links verschieben |
| <code>\f</code> | <i>form feed</i> – Textcursor um eine Zeile nach unten verschieben |
| <code>\n</code> | <i>newline</i> – Textcursor an den Anfang der nächsten Zeile verschieben |
| <code>\r</code> | <i>carriage return</i> – Textcursor an den Anfang der aktuellen Zeile verschieben |
| <code>\t</code> | <i>horizontal tab</i> – Textcursor auf die nächste Tabulator-Position nach rechts verschieben |
| <code>\v</code> | <i>vertical tab</i> – Textcursor auf die nächste vertikale Tabulator-Position nach unten verschieben |
| <code>\\</code> | <i>backslash</i> – das Zeichen <code>\</code> selbst |
| <code>\"</code> | <i>double quote</i> – das Zeichen <code>"</code> selbst |
| <code>\xhh</code> | <i>ASCII character</i> – das Zeichen mit dem ASCII-Code <code>0xhh</code> einfügen, wobei <code>hh</code> ein Hexadezimalwert zwischen 00 und FF ist |
| <code>\uhhhh</code> | <i>unicode character</i> – das Unicode-Zeichen <code>0xhhhh</code> einfügen, wobei <code>hhhh</code> ein Hexadezimalwert zwischen 00 und FFFF ist |
| <code>\Uhhhhhhhh</code> | <i>unicode character</i> – das Unicode-Zeichen <code>0xhhhhhhhh</code> einfügen, wobei <code>hhhhhhhh</code> ein Hexadezimalwert zwischen 00 und FFFFFFFF ist |
| <code>%%</code> | <i>percentage</i> – das Prozentzeichen. Achtung: Hier werden tatsächlich zwei <code>%</code> hintereinander gesetzt, nicht der Backslash <code>\</code> . |

Tabelle B.1.: Escape-Sequenzen nach ISO/IEC 9899:TC3

Siehe Seite 19 auf <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf>

B.2. Formatierte Textausgabe

Die Format-Codes in Tabellen B.2 und B.3 funktionieren sowohl mit `printf`, `sprintf`, `snprintf`, als auch mit `fprintf`.

Siehe auch <http://en.cppreference.com/w/c/io/fprintf>

| Datentyp | format string | Beispiele | Anmerkung |
|-------------------------|-----------------------|--|---|
| float oder double | %f %lf | 0.700000 1234567890123456768.000000 | Dezimalpunkt |
| | %e | 7.000000e-1 1.234568e+18 | Exponentialschreibweise mit „e“ |
| | %E | 7.000000e-1 1.234568E+18 | Exponentialschreibweise mit „E“ |
| | %g | 0.7 1.234568e+18 | Dezimal oder E-Schreibweise mit „e“ |
| | %G | 0.7 1.234568E+18 | Dezimal oder E-Schreibweise mit „E“ |
| | %a | 0x1.666666666666p-1 0x1.12210f47de981p+60 | Hexadezimal mit Kleinbuchstaben |
| | %A | 0X1.666666666666P-1 0X1.12210F47DE981P+60 | Hexadezimal mit Großbuchstaben |
| long double | %Lf | wie bei float und double | Variationen (%Le, ...) analog zu float und double |
| char | %c | A | als ASCII-Zeichen, Werte von 0 bis 255 |
| | %hhi | 65 | als Dezimalzahl, Werte von 0 bis 255 |
| int | %d %i | -1 2147483647 | Dezimal |
| | %u | 4294967295 2147483647 | Dezimal, ohne Vorzeichen |
| | %o | 3777777777 1777777777 | Oktal, ohne Vorzeichen |
| | %x | ffffff 7ffffff | Hexadezimal, ohne Vorzeichen, Kleinbuchstaben |
| | %X | FFFFFF 7FFFFFF | Hexadezimal, ohne Vorzeichen, Grpßbuchstaben |
| long long int | %lli, %llx, ... | -1 9223372036854775807 | alle Formen wie bei int, jeweils mit Präfix ll |
| short int | %hi | -1 32767 | Variationen analog zu int |

Tabelle B.2.: Codes für formatierte Textausgabe (Zahlen)

| Datentyp | format string | Beispiele | Anmerkung |
|-------------------------------------|------------------------|---------------------------------|--|
| alle Zahlen-Typen | <code>%+code</code> | -1 +2147483647 | Ausgabe wie oben, immer mit Vorzeichen |
| | <code>%N.Mcode</code> | -0.70 1234567890123456768.00 | Vorgabe: Mindestens Platz für N Zeichen; davon M für Nachkommastellen, rechtsbündig. Negative Zahlen für N und M machen die Ausgabe linksbündig. |
| | <code>%0N.Mcode</code> | -00.70 1234567890123456768.00 | Vorgabe: wie oben, aber leere Stellen werden mit 0 aufgefüllt |
| char*, char[], unsigned char* | <code>%s</code> | abcdefg | Als Zeichenkette; alle Zeichen von der adressierten Speicherstelle bis zum ersten NULL-Zeichen. |
| <i>Datentyp</i> * (Pointer) | <code>%p</code> | 0x7fffc7a9dfb8 (nil) | Pointer auf beliebigen Datentyp, Ausgabe als Hexadezimalzahl. NULL wird als (nil) ausgegeben. |

Tabelle B.3.: Codes für formatierte Textausgabe (Spacing und Strings)

B.3. Formatierte Texteingabe

Die Format-Codes in Tabelle B.4 funktionieren sowohl mit `scanf`, `sscanf`, `sscanf`, als auch mit `fscanf`. Siehe auch <http://en.cppreference.com/w/c/io/fscanf>

| Pointer für Datentyp | format string | Anmerkung |
|-------------------------------------|---------------|---|
| float | %f | Alle Schreibweisen (Dezimalpunkt, exponential, hexadezimal) werden automatisch erkannt) |
| double | %lf | |
| long double | %Lf | |
| int | %i | automatische Erkennung der Basis |
| | %d | dezimal |
| unsigned int | %u | vorzeichenlos, dezimal |
| | %o | vorzeichenlos, oktal |
| | %x | vorzeichenlos, hexadezimal |
| char | %hi | Werte zwischen -128 und 127 |
| | %c | ein Zeichen |
| short int | %hi, %hu, ... | Variationen analog zu <code>int</code> und <code>unsigned int</code> |
| long int | %li, %lu, ... | Variationen analog zu <code>int</code> und <code>unsigned int</code> |
| long long int | %lli, ... | Variationen analog zu <code>int</code> und <code>unsigned int</code> |
| char*, char[], unsigned char* | %s | Zeichenkette beliebiger Länge. Leerzeichen und Tabulatoren werden als Text-Ende interpretiert. |
| | %Ns | Zeichenkette mit maximaler Länge N. Leerzeichen und Tabulatoren werden als Text-Ende interpretiert. |
| | %[list]s | Zeichenkette beliebiger Länge. Nur Zeichen, die in <i>list</i> aufgeführt sind, werden akzeptiert. Der Ausdruck <i>list</i> kann entweder eine Aneinanderreihung aller erlaubten Zeichen sein, oder ein Ausdruck der Art <i>von-bis</i> . Alles hinter dem ersten Zeichen, das nicht in <i>list</i> genannt wurde, wird ignoriert. Beispiel 1: %[0-9a-fA-F] erlaubt Eingabe von Hexadezimalen Ganzzahlen Beispiel 2: %[\ta-z] erlaubt Eingabe von Kleinbuchstaben, dem Tabulator und dem Leerzeichen. |

Tabelle B.4.: Codes für formattierte Texteingabe

B.4. UNIX/bash-Kommandos zum Formatieren

Die Ausgabe der Zeichenketten (z. B. mit `printf`) in den Tabellen B.5 und B.6 wird von den auf *unixoiden Systemen* (z. B. Linux, Mac OS) üblichen Konsolen als Änderung des Ausgabeformats interpretiert. Die Details sind abhängig von der Implementierung der Konsole. Beschrieben hier ist die auf verbreitete Konsole *bash*. Die beschriebenen Farben stellen die Standard-Einstellungen dar und können bei jedem Anwender individuell angepasst werden.

| Farbe | Schriftfarbe | Hintergrund |
|------------------|--------------|-------------|
| Schwarz | \x1b[30m | \x1b[40m |
| Rot | \x1b[31m | \x1b[41m |
| Grün | \x1b[32m | \x1b[42m |
| Braun | \x1b[33m | \x1b[43m |
| Blau | \x1b[34m | \x1b[44m |
| Magenta | \x1b[35m | \x1b[45m |
| Zyan | \x1b[36m | \x1b[46m |
| Grau | \x1b[37m | \x1b[47m |
| Dunkelgrau | \x1b[90m | \x1b[100m |
| Hellrot | \x1b[91m | \x1b[101m |
| Hellgrün | \x1b[92m | \x1b[102m |
| Gelb | \x1b[93m | \x1b[103m |
| Hellblau | \x1b[94m | \x1b[104m |
| Pink | \x1b[95m | \x1b[105m |
| Hellzyan | \x1b[96m | \x1b[106m |
| Hellgrau | \x1b[97m | \x1b[107m |
| (Standard-Farbe) | \x1b[39m | \x1b[49m |

Tabelle B.5.: UNIX/bash-Farbkommandos

| Effekt | Aktivieren | Deaktivieren |
|--|------------|--------------|
| Normalen Text einstellen | \x1b[0m | – |
| Fettdruck | \x1b[1m | \x1b[21m |
| Kursivtext | \x1b[3m | \x1b[23m |
| Unterstreichung | \x1b[4m | \x1b[24m |
| Blinkend | \x1b[5m | \x1b[25m |
| Standardeinstellungen wiederherstellen | \033[m | – |

Tabelle B.6.: UNIX/bash-Formatkommandos

B.5. Operator-Hierarchie

Siehe auch https://en.cppreference.com/w/c/language/operator_precedence für weitere Informationen.

C++: Erweiterte Liste von Operatoren

Die Sprache C++ kennt einige weitere Operatoren, die hier nicht angesprochen werden können. Der Vollständigkeit halber sei dennoch auf den C++-spezifischen Artikel unter https://en.cppreference.com/w/cpp/language/operator_precedence verwiesen.

| Priorität | Operator | Beschreibung |
|-----------|----------|--|
| 1 | ++, -- | Inkrement und Dekrement (Postfix) |
| | () | Funktionsaufruf |
| | [] | Array-Indizierung |
| | ., -> | Strukturzugriff |
| 2 | ++, -- | Inkrement und Dekrement (Prefix) |
| | !, ~ | Logisches und bitweises NOT |
| | (type) | Typecast |
| | * | Dereferenzierung |
| | & | Adresse finden |
| | sizeof | Strukturgröße ermitteln |
| 3 | *, /, % | Multiplikation, Division, Modulo |
| 4 | +, - | Addition, Subtraktion |
| 5 | <<, >> | Bitshift nach links und rechts |
| 6 | <, > | Vergleichsoperatoren kleiner/größer als |
| | <=, >= | und kleiner gleich/größer gleich |
| 7 | ==, != | Vergleichsoperatoren Gleichheit, Ungleichheit |
| 8 | & | Bitweises AND |
| 9 | ^ | Bitweises XOR |
| 10 | | Bitweises OR |
| 11 | && | Logisches AND |
| 12 | | Logisches OR |
| 13 | ? : | Bedingte Auswertung (<i>ternärer Operator</i>) |
| 14 | = | |

Tabelle B.7.: Präzedenz der Operatoren in C

B.6. Übersicht Datentypen

Tabelle B.8 beschreibt Datentypen, die immer zur Verfügung stehen. Grau gesetzte Werte sind plattformabhängig; hier ist jeweils die übliche Umsetzung auf modernen Rechnern angegeben.

| Datentyp | Wertebereich | Speicherbedarf |
|------------------------|---|----------------|
| char | -128 ... +127 | 1 Byte |
| short int | -32 768 ... +32 767 | 2 Byte |
| int | -2 147 483 648 ... +2 147 483 647 | 4 Byte |
| long int | -9 223 372 036 854 775 808 ... +9 223 372 036 854 775 807 | 8 Byte |
| long long int | -9 223 372 036 854 775 808 ... +9 223 372 036 854 775 807 | 8 Byte |
| unsigned char | 0 ... 255 | 1 Byte |
| unsigned short int | 0 ... 65 535 | 2 Bytes |
| unsigned int | 0 ... 4 294 967 295 | 4 Bytes |
| unsigned long int | 0 ... 1.844 674 4e+19 | 8 Bytes |
| unsigned long long int | 0 ... 1.844 674 4e+19 | 8 Bytes |
| float | $\pm 3.4\text{E}+38$, ca. 7 signifikante Ziffern | 4 Byte |
| double | $\pm 1.7\text{E}+308$, ca. 16 signifikante Ziffern | 8 Byte |
| long double | sehr viel | 10 Byte |

Tabelle B.8.: Standard-Datentypen der Sprache C

Tabelle B.9 beschreibt Datentypen, die zur Verfügung stehen, sobald der Header `stdint.h` eingebunden wird:

| Datentyp | Wertebereich | Speicherbedarf |
|-----------------------|---|----------------|
| <code>int8_t</code> | -128 ... +127 | 1 Byte |
| <code>int16_t</code> | -32 768 ... +32 767 | 2 Byte |
| <code>int32_t</code> | -2 147 483 648 ... +2 147 483 647 | 4 Byte |
| <code>int64_t</code> | -9 223 372 036 854 775 808 ... +9 223 372 036 854 775 807 | 8 Byte |
| <code>uint8_t</code> | 0 ... 255 | 1 Byte |
| <code>uint16_t</code> | 0 ... 65 535 | 2 Byte |
| <code>uint32_t</code> | 0 ... 4 294 967 295 | 4 Byte |
| <code>uint64_t</code> | 0 ... 1.8446744e+19 | 8 Byte |

Tabelle B.9.: Standard-Datentypen der Sprache C

Weitere Details können von der CPP-Referenz entnommen werden:

- Standard-Typen: <https://en.cppreference.com/w/c/language/types>.
- Erweiterte Ganzzahl-Typen: <https://en.cppreference.com/w/c/types/integer> (wie definiert in `stdint.h`)

C++: Erweiterte Liste von Datentypen

Die Sprache C++ wurde gegenüber C um einige Standard-Typen erweitert, die auf dem Artikel unter <https://en.cppreference.com/w/cpp/language/types> beschrieben werden.

B.7. Vordefinierte Präprozessor-Symbole

Unter <https://gcc.gnu.org/onlinedocs/cpp/Predefined-Macros.html> finden Sie eine Liste von vordefinierten Macros. Für Betriebssystem-spezifische Macros, siehe http://nadeausoftware.com/articles/2012/01/c_c_tip_how_use_compiler_predefined_macros_detect_operating_system.

Hier seien einige der nützlichsten Macros aufgeführt¹:

¹Ich beziehe mich hier auf den `gcc`; für andere Compiler haben die Macros ggf. andere Namen oder sind gar nicht definiert.

| Macro | Umsetzung |
|--|---|
| <code>__FILE__</code> | String, der den Dateinamen der aktuell verarbeiteten Datei enthält |
| <code>__LINE__</code> | String, der die Nummer der Code-Zeile in der aktuell verarbeiteten Datei enthält |
| <code>__func__</code> | String, der die aktuell kompilierte Funktion enthält |
| <code>__DATE__</code> | String, der das Datum zur Kompilierzeit im Format MMM DD YYYY enthält. |
| <code>__TIME__</code> | String, der die Uhrzeit zur Kompilierzeit im Format HH:MM:SS enthält. |
| <code>__VERSION__</code> und <code>__STDC_VERSION__</code> | String, der die Versionsnummer des Compilers enthält. |
| <code>__unix__</code> | Symbol ohne Inhalt, nur definiert, wenn für unixoide Systeme kompiliert wird |
| <code>__linux__</code> | Symbol ohne Inhalt, nur definiert, wenn für das Betriebssystem Linux kompiliert wird |
| <code>__APPLE__</code> | Symbol ohne Inhalt, nur definiert, wenn für das Betriebssysteme von Apple kompiliert wird |
| <code>__WIN32__</code> | Symbol ohne Inhalt, nur definiert, wenn für 32bit-Windows-Systeme kompiliert wird |
| <code>__WIN64__</code> | Symbol ohne Inhalt, nur definiert, wenn für 64bit-Windows-Systeme kompiliert wird |

Tabelle B.10.: Vordefinierte Präprozessor-Symbole des `gcc`

C. Geschichte

Der folgende Abschnitt ist eine freie Übersetzung von:

<https://www.techopedia.com/the-history-of-the-c-programming-language/2/32996>

Bis heute zählt die Sprache C zu einer der bedeutendsten Programmiersprachen. Von der Vielzahl an Programmiersprachen, die für verschiedenste Einsatzbereiche zur Verfügung stehen, ist ein Großteil von C beeinflusst. Ungeachtet der Existenz dieser Spezialisierungen bleibt C als Allzweck-Sprache von Bedeutung.

Es ist nicht bekannt, ob die Entwickler *Dennis Ritchie* und *Ken Thompson* bereits die Vision der großen Erfolge hatten, die die Sprache C einmal erreichen würde. Wie die meisten Innovationen sah auch C viele Veränderungen über die Zeit. Vermutlich der größte Erfolg ist die Tatsache, dass C bis in die jüngste Zeit hinein relevant blieb. Für die Entwickler ist es sicherlich sehr erfüllend zu sehen, dass C nicht als überholt oder als nur in Nischenanwendungen nützlich gilt. Stattdessen ist C als starke Allzweck-Sprache anerkannt.

Das Ursprüngliche Ziel der Entwickler war nicht die Entwicklung einer neuen Sprache. Tatsächlich war es erst die Entwicklung der Sprache anstoßen. In den 1960ern arbeitete

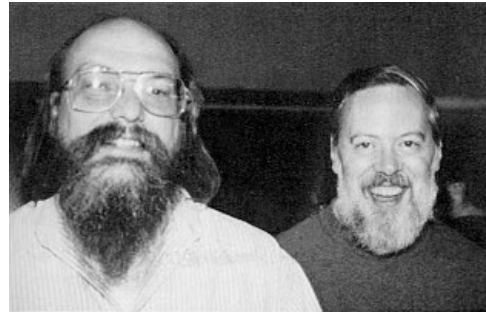


Abbildung C.1.: Ken Thompson und Dennis Ritchie (1973) – die Entwickler der Sprache C

https://en.wikipedia.org/wiki/Dennis_Ritchie

Zusammenspiel mehrerer Zufälle, die die Entwicklung der Sprache anstoßen. In den 1960ern arbeitete Dennis Ritchie bei Bell Labs (AT&T) an einem Betriebssystem das von vielen Anwendern gleichzeitig benutzt werden konnte. Dieses Projekt *MULTICS* (Multiplexed Information and Computer Services) sollte die gemeinsame Verwendung von Rechenressourcen erlauben. Von dem Projekt versprach man sich viele Verbesserungen – aber bald stellte sich heraus, dass die Kosten für die Umsetzung die Vorteile überwiegen würden. Bell Labs zog sich graduell aus dem Projekt zurück.

Vor diesem Hintergrund schloss sich Ritchie dem Team von Ken Thompson und Brian Kernighan an. Thompson entwickelte in der Sprache Assembler ein Dateisystem für den DEC PDP-7 Supercomputer. Weitere Verbesserungen daran führten schließlich zur Konzeption des Betriebssystems UNIX, in dem einige der Ideen von MULTICS wieder auflebten. Tatsächlich weist schon der Originalname des neuen Projekts – UNICS (Uniplexed Information and Computing Service) – auf seine Verwandtschaft mit dem früheren Projekt hin.

UNIX wurde in Assembler geschrieben, was leicht für Computer umzusetzen ist, für Menschen aber eine herausfordernde Arbeitsumgebung darstellt. Um die Arbeit an UNIX zu erleichtern kamen die Sprachen Fortran und B zum Einsatz. Aus den Einschränkungen, die diese Sprachen brachten, erwuchs die Idee für die Sprache C.

Die Sprache B – eine Weiterentwicklung der Sprache BCPL von Martin Richards – stellte sich als nützliches Werkzeug in der Entwicklung von UNIX heraus. Jedoch mussten auch hier, wie auch in Assembler, Daten in Maschinensprache zur Verfügung gestellt werden. Datenstrukturen wurden in B ebenfalls nicht unterstützt.

Für die effiziente Fortsetzung der Arbeiten musste sich etwas ändern. Aus diesem Grunde machten sich Ritchie und seine Kollegen in den Jahren 1971 bis 73 daran, die Einschränkungen, die B ihnen auferlegte,

abzuschalten. Es sollte nicht vergessen werden, dass die Sprache C im Geiste von B entwickelt wurde – trotz oder gerade wegen dessen Beschränktheit. Viele Features und Konzepte wurden beibehalten, während neue Ideen – darunter Datentypen und Strukturen – hinzugefügt wurden. Der Name C weist direkt darauf hin, dass es sich hierbei um einen Nachfolger der Sprache B handelt. Anfangs wurde C im Hinblick auf die Arbeit an UNIX entwickelt. Um die Performance des Kernels und die Arbeit am Betriebssystem zu verbessern wurden viele UNIX-Komponenten neu in C umgeschrieben.

Ritchie und Kernighan dokumentierten ihr Werk in dem Buch *The C Programming Language*. Obwohl Kernighan behauptete, keinen Einfluss auf das Design der Sprache C genommen zu haben, ist er doch der Autor vieler berühmter UNIX-Programme.

Bald begann C, in PCs für die Software-Entwicklung eingesetzt zu werden. Die erste (kleinere) Änderung am Sprachkonzept fand im Jahr 1983 statt, als das American National Standards Institute (ANSI) ein Komitee zur Standardisierung der Sprache bildete. Einige Modifikationen wurden vorgenommen, die Code in der nun normierten Sprache kompatibler zu Vorgängerversionen machte. Im Jahre 1989 wurde der neue Standard verabschiedet und ist heute als ANSI C oder C89 bekannt. Die International Organization for Standardization (ISO) trug auch zur Standardisierung von C bei.

Im Laufe der Zeit hat sich C stark weiterentwickelt und Features wie Speichermanagement, Funktionen, Klassen und Bibliotheken in ihren Umfang aufgenommen und wird heute in einigen der größten und bekanntesten Projekte der Welt verwendet. Die Sprache beeinflusste die Entwicklung vieler anderer Sprachen wie AMPL, AWK, csh, C++, C-, C#, Objective-C, Bit C, D, Go, Java, JavaScript, Julia, Limbo, LPC, Perl, PHP, Pike, Processing, Python, Rust, Seed7, Vala und Verilog.

Sind Sie ein Windows-User? Dann benutzen Sie ein Produkt, das vorwiegend in C geschrieben ist. Dasselbe gilt für MacOS, Linux, Android, iOS als auch für das Windows Phone – kurz für alle modernen Betriebssysteme. Weiter findet C auch verbreitete Anwendung in Embedded Systems wie sie in Fahrzeugen, Smart TVs und den vielen Elementen des Internet of Things (IoT) verbaut sind.

Die vielfältigen Anwendungsfelder von C sind zu zahlreich, um sie hier aufzulisten. Einige jedoch seien als Beispiel genannt:

- Entwicklung von Compilern
- Datenbanken
- Computer- und Handy-Spiele
- Der UNIX-Kernel in seiner fortlaufenden Entwicklung
- Auswertung Mathematischer Gleichungen
- Design von Netzwerk-Geräten

Wie alle der großen Erfindungen entstand die Sprache C aus einer Notwendigkeit heraus. Die Probleme und Umstände der Zeit ihrer Entwicklung waren die Inspiration für die Konzeption der Sprache. Anders als viele Sprachen, die heute als (fast) ausgestorben gelten, „gedeiht“ C auch weiterhin. Einige Sprachen finden heute nur noch in Nischenbereichen Anwendung – Fortran etwa kommt nur noch im Engineering-Bereich zum Einsatz, und COBOL hat kaum mehr Relevanz. C dagegen ist nicht nur weiterhin eine Sprache, die es zu kennen lohnt. Sie ist und bleibt Einfluss für die Weiterentwicklung bestehender und Konzeption neuer Sprachen. aufwändige Selbst Technologien wie das IoT, AI und Automations-Konzepte konnten sich nicht von der Sprache C lösen. Es bleibt zu erwarten, dass diese Sprache auch lange in die Zukunft ein Teil der Programmier-Welt bleiben wird.

Abbildungsverzeichnis

| | |
|--|-----|
| 1.1. ASCII-Codes | 2 |
| 3.1. Pointer im echten Leben | 24 |
| 3.2. Dezimale und binäre runde Zahlen | 28 |
| 5.1. Ein realistisches Szenario | 40 |
| 6.1. Die Startseite der C-Referenz https://en.cppreference.com/w/c | 53 |
| 6.2. Suchergebnisse in der CPP-Referenz | 53 |
| 6.3. Anfang des Artikels zu <code>printf</code> auf der CPP-Referenz | 54 |
| 6.4. Liste der Header auf der CPP-Referenz | 55 |
| 6.5. Funktionen der math-library in der CPP-Referenz | 55 |
| 6.6. Anfang des Artikels zu <code>sqrt</code> in der CPP-Referenz | 56 |
| 7.1. Programmflussdiagramm mit Schleife | 59 |
| 8.1. Langlebigkeit von Standards in der IT | 76 |
| 9.1. Code Quality | 112 |
| 15.1. Das <i>Jahr 2038-Problem</i> | 180 |
| 16.1. Auflistung einer Ordnerstruktur | 192 |
| 18.1. Teach Yourself C++ in 21 days – ein Praxisbericht. | 221 |
| C.1. Ken Thompson und Dennis Ritchie (1973) – die Entwickler der Sprache C | 247 |