Wintersemester 2025/2026

C-Programmierung

Technische Hochschule Mannheim

C Grundlagen

Prof. Thomas Smits

Stand 2025-07-21

Technische Hochschule mannheim

Inhaltsverzeichnis

Kapitel 1

Grundlagen

1.1 Video zum Kapitel [5]

Link zu YouTube

1.2 Warum zum Teufel C? [6]

C ist – nach heutigen Maßstäben – eine uralte Programmiersprache, die für den Benutzer unkomfortabel ist und ihn wenig davor beschützt Fehler zu machen. Viele Dinge, die man in Java ein einigen Zeilen erledigen kann, benötigen in C umfangreiche Konstrukte und bereiten viel Aufwand, bis sie einwandfrei funktionieren. C ist ein Dinosaurier unter den Programmiersprachen. Die Produktivität in C ist meistens deutlich geringer, als in modernen Skriptsprachen wie Ruby oder Python.

Warum sollte man sich heute noch mit C beschäftigen? Oder noch kürzer „warum zum Teufel C?“.

The *good*

- C ist gleichzeitig *Highlevel-* und *Lowlevel-Sprache*
- C bietet bessere *Kontrolle* über die Hardware / das Betriebssystem
- C liefert sehr gute *Performance*
- C ist für *Echtzeitprogrammierung* geeignet
- C erlaubt die *Programmierung von Betriebssystemen* (Linux, Windows, …)
- C hilft die *Hardware zu verstehen* (Embedded Systems)
- C läuft auf *jeder Hardware*, vom Großrechner zum Kühlschrank
- C wird von Tools zum *Reverse-Engineering* verwendet

C ist auf keinen Fall irrelevant und wird heute noch in vielen Bereichen eingesetzt. Durch die Nähe zur Maschine und die **explizite Speicherverwaltung** kann man in C Dinge programmieren, die in anderen Sprachen nicht zu bewältigen wären, z. B. Betriebssysteme oder Echtzeitanwendungen. Eine Echtzeitanwendung, z. B. die Steuerung eines Airbags, wäre in Java undenkbar, da Java gelegentlich Pausen für die Garbage Collection einlegt und damit kein vorhersehbares Zeitverhalten hat. Ebenso benötigt man für die Entwicklung eines Betriebssystems eine Sprache, die direkten Zugriff auf die Hardware erlaubt – eine Java VM wäre hier auf jeden Fall im Weg. Dasselbe gilt beim Vergleich von C mit Skriptsprachen, wie z. B. Python. Während man in Python sehr produktiv programmieren kann, ist die Geschwindigkeit der Programme deutlich geringer. Aus diesem Grund enthalten die Bibliotheken für wissenschaftliches Rechnen in Python auch optimierte, in C programmierte Teile, um die Probleme effizient lösen zu können.

explizite Speicherverwaltung

Wenn man sich mit fortgeschrittenen Themen wie **Reverse Engineering** beschäftigt, also der Frage, wie man ein vorliegendes Programm im Maschinencode analysieren kann, ist C ein wichtiges Hilfsmittel. Wegen seiner Nähe zur Hardware kann man Maschinenprogramme einfach als C-Code darstellen, sodass man beim Reverse Engineering nicht die Assembler-Instruktionen lesen muss, sondern sich mit dem daraus erzeugten C-Code beschäftigen kann.

Reverse Engineering

The *bad*

- C *verzeiht keine Fehler*
- C hat eine *manuelle Speicherverwaltung*
- C hat *keine eingebaute Fehlerbehandlung*
- C benötigt *umfangreicheren Code* als in anderen Sprachen
- C hat *keine goße Standardbibliothek*

Das Alter von C und die Hardwarenähe haben aber auch Auswirkungen. So muss man in C sehr viele Dinge manuell machen, die man in anderen Programmiersprachen geschenkt bekommt. Bei Speicherverwaltung und Fehlerbehandlung bietet C nur sehr wenig Unterstützung und man muss beim Programmieren höllisch aufpassen, um keine Fehler zu machen, die zu einem harten Absturz des Programmes führen.

C-Programme benötigen oft deutlich mehr Code, um dasselbe Ziel zu erreichen als Sprachen wie Python oder Ruby. Das liegt auch daran, dass es in C wenig vorgefertigte Bibliotheken gibt, aus denen man Funktionalitäten nutzen kann. Zwar hat C eine **Standardbibliothek**, wie auch Java, aber diese ist zu einer Zeit standardisiert worden, als es viele Konzepte noch gar nicht gab, z. B. XML, HTTP etc. Diese muss man deswegen heute über externe Bibliotheken nachrüsten.

Standardbibliothek

1.3 Geschichte von C [8]

1960er Jahre: Viele neue Programmiersprachen

- *Assembler* für Betriebssysteme und zeitkritischen Code
- *COBOL* für Geschäftsanwendungen
- *FORTRAN* für wissenschaftliche Berechnungen

- *Lisp*, *Simula* für Informatik-Forschung (AI)
- *PL/I* als Sprache der 2. Generation

Die Geschichte von C beginnt zu einer Zeit, als die Entwicklung der Computer noch in den Kinderschuhen steckte.

Nachdem zu Beginn alle Programme direkt in der **Maschinensprache** des jeweiligen Prozessors geschrieben wurde, entstanden in den 1960 Jahren neue Hochsprachen, welche die Programmierung und die Übertragung von Programmen auf neue Hardware vereinfachen sollten. Die neu entwickelten Sprachen konzentrierten sich jeweils auf spezifische Bereiche: **COBOL** für Geschäftsanwendungen, **FORTRAN** für mathematische Berechnungen und Lisp für die KI-Forschung – Lisp ist tatsächlich eine der ältesten Programmiersprachen überhaupt, mit einer ersten Spezifikation aus dem Jahr 1958.

Maschinensprache

COBOL
FORTRAN

Hello World in FORTRAN

```
program hello
  ! This is a comment line; it is ignored by the compiler
  print *, 'Hello, World!'
end program hello
```

Hello World in COBOL

```
IDENTIFICATION DIVISION.
PROGRAM-ID. IDSAMPLE.
ENVIRONMENT DIVISION.
PROCEDURE DIVISION.
    DISPLAY 'HELLO WORLD'.
    STOP RUN.
```

Eine der wenigen Sprachen der damaligen Zeit, die für einen breiteren Anwendungsbereich vorgesehen wurde, war PL/I, die 1964 das Licht der Welt erblickte.

Hello World in PL/I

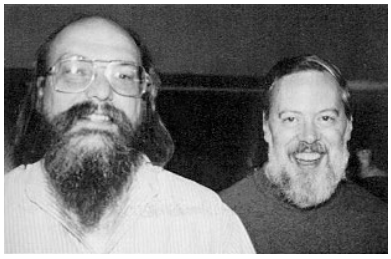
```
Hello2: proc options(main);
     put list ('Hello, world!');
end Hello2;
```

Ebenso rudimentär wie bei den Programmiersprachen war die Lage bei den Betriebssystemen: Diese wurden ebenfalls in Assembler geschrieben und liefen ausschließlich auf der Hardware, für die sie Entwickelt wurden.

- Dominante Betriebssysteme: *OS/360* (Großrechner) und *Multics* (in PL/I geschrieben) (Multiplexed Information and Computer Services)
- Bell Labs wollte eigenes Betriebssystem (als Ersatz für Multics)
  ⇒ *Unics* (Uniplexed Information and Computing Service)

- Erste Version von Unics (dann *Unix*) wurde in Assembler geschrieben
  ⇒ Portierung (Übertragung) auf neue Hardware bei Assembler sehr aufwändig
- **Ken Thompson** entwickelte **B** auf Basis von BCPL auf der PDP-7 (16 kB RAM)
  → einfacher als PL/I und BCPL
- **Dennis Ritchie** entwickelt auf Basis von B die Sprache C (1969–1973)

Ken Thompson
B
Dennis Ritchie



Ken Thompson (links) und Dennis Ritchie (rechts)

Aus dem Bedürfnis heraus, einen Nachfolger für **Multics** zu entwickeln, begannen Ken Thompson und Dennis Ritchie um 1965 mit der Entwicklung von Unix, damals noch „Unics“ genannt. Ein Ziel der beiden war, das Betriebssystem möglichst einfach auf neue Hardware portieren zu können, wobei kleinere Minicomputer wie die PDP/7 oder PDP/11 unterstützt werden sollten. Die verfügbaren Programmiersprachen waren dafür aber entweder zu hardwarenah (Assembler) und damit nicht portierbar oder zu schwergewichtig (PL/I), weswegen Thompson mit B) eine neue Sprache entwarf.

Multics

Beispielprogramm in B

```
main()
{
    auto c;
    auto d;
    d=0;
    while(1)
    {
        c=getchar();
        d=d+c;
        putchar(c);
    }
}
```

Dennis Ritchie entwickelte B dann weiter zur Sprache C, wie wir sie heute kennen. Er und Thompson verwendeten C, um das Betriebssystem **Unix** zu entwickeln. Die ersten Versionen von Unix waren noch in Assembler geschrieben, ab 1973 (Version 4 Unix) dann in C.

Unix

Bei der Entwicklung von C bekam Ritchie Unterstützung von **Brian Kernighan**, mit dem zusammen er auch das Standardwerk „The C Programming Language“ zur C-Programmierung schrieb.

Brian Kernighan

1.4 Von C zu Java [11]

- C: **Dennis Ritchie** (1969–1973) — Dennis Ritchie
  - Sprache zur Systemprogrammierung
  - macht das Betriebssystem hardwareunabhängig
  - ursprünglich nicht für Anwendungen gedacht (→ PL/I oder Fortran)
- C++: **Bjarne Stroustrup** (Bell Labs), 1980er
  → objektorientierte Erweiterung zu C — Bjarne Stroustrup
- Java: **James Gosling** in den 1990ern — James Gosling
  - für eingebettete Systeme gedacht
  - objektorientiert (ähnlich C++ aber deutlich vereinfacht)
  - Syntax von C übernommen

Nachdem C in den 1970er Jahren entwickelt wurde, wurde es in den 1980er Jahren von Bjarne Stroustrup um Objektorientierung erweitert, woraus dann **C++** entstand. — C++

```
#include <iostream>

int main(int argc, const char * argv[])
{
    std::cout << "Hello, world!\n";
}
```

Parallel zu C++ wurde von Tom Love und Brad Cox mit **Objective-C** eine andere objektorientierte Version von C entworfen, die sich bei Apple lange Zeit als Programmiersprache für macOS-Anwendungen gehalten hat, inzwischen aber von Swift abgelöst wurde. — Objective-C

Hello World in Objective-C

```
// FILE: hello.m
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[])
{
    /* my first program in Objective-C */
    NSLog(@"Hello, World! \n");
    return 0;
}
```

Die Syntax der Sprache C diente als Vorbild für viele weitere Sprachen, die heute noch in Gebrauch sind. Neben Java und C# finden sich viele Ideen von C ebenfalls in Go, PHP und weiteren Sprachen wieder.

1.5 C vs. Java [12]

Da Java häufig als erste Programmiersprache gelehrt wird, ist es sinnvoll einen Vergleich zwischen Java und C zu ziehen.

- Java ist eine *objektorientierte* Sprache von Mitte der 1990er Jahre
- C ist eine **prozedurale Sprache** aus den frühen 1970er Jahren

prozedurale Sprache

- *Vorteile* von C
  - direkter Zugriff auf das Betriebssystem (System-Calls)
  - hohe Performance
- *Nachteile* von C
  - Sprache ist portabel, Schnittstellen zum Betriebssystem nicht
  - wenig Unterstützung bei der Programmierung
  - Memory-Leaks und Abstürze drohen
  - Präprozessor ist obskur (und die Fehler noch mehr)

Java ist mehr als 20 Jahre jünger als C und mit einem anderen Ziel entwickelt worden: Während C maschinen- und **hardwarenah** für die Betriebssystementwicklung entworfen wurde, hatte Java das Ziel möglichst maschinenunabhängig zu sein und sollte für die Entwicklung von Anwendungen – damals für Set-Top-Boxen – dienen. Außerdem ist Java konsequent objektorientiert, während C rein prozedural ist.

hardwarenah

Die Entwickler von Java habe sich dafür entschieden, die **Syntax** von Java an C anzulehnen, um Vertrautes zu verwenden. Die **Semantik** von Java orientiert sich aber viel mehr an Smalltalk als an C. Java-Code sieht auf den ersten Blick C-Code ähnlich, funktioniert aber häufig anders.

Syntax
Semantik

Folgende Tabelle gibt einen Überblick über die Unterschiede zwischen den beiden Sprachen.

Java	C
objektorientiert	prozedural
strikt getypt	getypt, kann aber übersteuert werden
Klassen und Pakete für Namespaces	flacher Namensraum
keine Makros	Makros Kern der Sprache
automatische Speicherverwaltung	manuelle Speicherverwaltung
keine Pointer	Pointer
Funktionsaufruf by-value	Funktionsaufruf by-value
Ausnahmebehandlung	Manuelle Fehlerbehandlung
Arrays kennen ihre Länge	Arrays kennen ihre Länge nicht
String als Datentyp	Ausschließlich `char`-Arrays
Viele Bibliotheken	Bibliotheken vom Betriebssystem

Aus der Tabelle sind vier Aspekte besonders herauszuheben, die Umsteigern von Java auf C häufig Schwierigkeiten bereiten:

- **Speicherverwaltung**: In C muss der Entwickler die gesamte Speicherverwaltung übernehmen. Sowohl das Reservieren, als auch das Freigeben von Speicher muss explizit erfolgen. Speicherverwaltung
- **Pointer**: Pointer sind in C ein wichtiger Datentyp und können über die sogenannte Pointerarithmetik manipuliert werden. Ein sicherer Umgang mit Pointern ist unerlässlich, um C-Programme schreiben zu können. Pointer
- **Strings**: C kennt keine Strings. Zeichenketten werden als Arrays von Zeichen (`char[]`) realisiert, deren Ende durch ein Nullbyte (`0x00`) gekennzeichnet ist. Strings
- **Länge von Arrays**: Ein C-Array kennt seine Länge nicht und es ist Aufgabe des Entwicklers, die Länge zu verwalten und an Funktionen, die Arrays verwalten zu übergeben. Länge von Arrays

Java-Programm

- Sammlung von **Klassen** Klassen
- Programm läuft in einer *Java-VM* (Java virtuelle Maschine)
- `main`-Methode einer Klasse ist *Einstiegspunkt*
- Java VM lädt weitere Klassen bei Bedarf aus *JAR-Archiven* nach

C-Programm

- Sammlung von **Funktionen** Funktionen
- Programm läuft direkt auf dem *Prozessor* (physikalischer Maschine)
- `main`-Funktion ist der *Einstiegspunkt*
- alle Funktionen sind zu einem *Executable* (z. B. `.exe`) zusammengefasst
- *Dynamische Libraries* (`.dll`, `.so`) enthalten gemeinsam genutzte Funktionen

Da C nicht objektorientiert ist, bestehen C-Programme nicht aus Klassen, sondern sind eine Sammlung von Funktionen, die sich gegenseitig aufrufen. Alle Funktionen eines C-Programms werden vom Linker (siehe unten) in ein sogenanntes **Executable** zusammengefasst. Gemeinsam genutzte Funktionen werden in dynamischen Libraries abgelegt, die unter Windows die Dateiendung `.dll` und unter Linux/Unix die Dateiendung `.so` bekommen. Executable .dll

Der Namensraum eines C-Programms ist flach, d. h. alle Funktionen teilen sich denselben Namensraum und es dürfen keine zwei Funktionen in einem Programm denselben Namen haben.

1.6 Das erste C-Programm [16]

Im Folgenden soll ein erstes C-Programm Schritt für Schritt analysiert werden. Hierzu dient das berühmte „Hello World“-Beispiel.

HelloWorld.java

```
public class Hello {
    public static void main(String[] args) {
        /* Greet the world */
```

```
        System.out.printf("%s", "Hello, World!\n");
        System.exit(0);
    }
}
```

hello_world.c

```c
#include <stdio.h>

int main(int argc, char** argv) {
    /* Greet the World */
    printf("%s\n", "Hello, World!");
    return 0;
}
```

- `#include <stdio.h>`
  - Zeilen mit # werden vom Präprozessor ausgewertet
  - lädt eine Header-Datei (→ `import` in Java)
- `int main(int argc, char** argv){`
  - deklariert und definiert eine Funktion mit dem Namen `main`
  - Rückgabetyp der Funktion ist `int`
  - zwei Parametern `argc` und `argv`
    `argc` ist die Anzahl der Argumente
    `argv` ist ein Array von Strings mit den Kommandozeilenargumenten

Die *Präprozessordirektiven* werden mit # eingeleitet und sind eine Besonderheit von C-Programmen. Auf den **Präprozessor** wird später noch detailliert eingegangen. Hier sei allerdings bereits erwähnt, dass ein C-Programm in zwei Schritten kompiliert wird: Zuerst verarbeitet der Präprozessor den Quelltext und erzeugt Dateien, die dann vom C-Compiler verarbeitet werden. Dies bedeutet, dass der C-Compiler die Präprozessordirektiven gar nicht versteht, sondern dafür ein eigenes Werkzeug zuständig ist. Dieses Vorgehen kann man nur historisch aus den Beschränkungen der damaligen Hardware nachvollziehen, da der Speicher sehr knapp war und man den Compiler möglichst klein halten wollte.

Präprozessor

Die zwei Sterne bei `char **argv` bedeuten, dass es sich um einen Pointer auf einen Pointer handelt, der auf ein Zeichen zeigt. Das ergibt auf den ersten Blick wenig Sinn, denn an der entsprechenden Stelle wird in Java ein Array von Strings übergeben. In C findet sich hier ebenfalls ein Array von Strings, die aber, da es keine Strings gibt, Arrays von Zeichen sind. Wir haben es somit mit einem Array von Arrays von Zeichen zu tun. In Java könnte man sich das dann als `char[][]` vorstellen. Der letzte Stein zum Verständnis der ** ist, dass ein Array in C als Pointer auf das erste Element verstanden werden kann, z. B. sind `int a[]` und `int* a` identisch. Der Zugriff kann entweder über den Index erfolgen `a[3]` oder aber über den Pointer `*(a+3)`. Damit ist das Geheimnis von `**argv` für das Erste gelüftet. Details zu Arrays und Pointern werden später noch diskutiert.

`char **argv`

**

Es bleibt noch die Frage, warum C eine Variable `int argc` hat, Java aber nicht? Der Grund ist, dass in C die Länge eines Arrays nicht bekannt ist und deshalb der `main`-Funktion mitgegeben werden muss, wie viele Parameter `argv` enthält. Für die Länge der Strings brauchen wir keine Längenangaben, weil diese **Nullterminiert** sind, also am Ende ein `0x00`-Byte haben.

Nullterminiert

- `/* Greet the World */`
  - Kommentar
  - Achtung: `//` ist kein Kommentarzeichen in ANSI-C (erst seit C99 und in C++)
- `printf("%s\n", "Hello, World!");`
  - gibt einen String aus (→ `printf` in Java)
  - `%s` ist der Platzhalter, der durch `Hello, World!` ersetzt wird
  - `\n` fügt einen Zeilenvorschub ein
- `return 0;`
  - beendet das Programm und gibt `0` als *Exit-Code* an das Betriebssystem zurück (→ `System.exit(0)` in Java)

Da fast alle C-Compiler C++-Compiler sind, akzeptieren sie `//` als Kommentarzeichen. Trotzdem ist `//` kein gültiger C-Kommentar. Da es immer passieren kann, dass ein C-Programm von einem Compiler übersetzt wird, der kein C++ beherrscht, sollte man auf `//` als Zeilenkommentar verzichten, insbesondere wenn man ältere Compiler antreffen könnte. In C99-Standard wurde `//` als Kommentarzeichen für Zeilenkommentare endlich auch in C aufgenommen.

//

Die Verwendung von Format-Strings ist bei C und Java nahezu identisch, was kein Wunder ist, da es Java einfach von C übernommen hat.

Am Ende übergibt jedes Programm einen Integer-Wert an das Betriebssystem um Fehler anzeigen zu können. Per Konvention bedeutet `0`, das kein Fehler im Programm aufgetreten ist.

In Java ist es unüblich, das Programm mit `System.exit` zu beenden, insbesondere, wenn man keinen Fehlercode setzen will. Anders als das `return 0;` in C beendet nämlich `System.exit(0);` den Prozess abrupt und bietet der Java-VM keine Möglichkeit mehr, aufzuräumen. Deswegen vermeidet man die Verwendung in Java üblicherweise.

Noch etwas eleganter kann man einen vordefinierten Rückgabewert für die `main`-Funktion aus der C-Standardbibliothek (`#include <stdlib.h>` nutzen. Diese definiert `EXIT_SUCCESS` für den Erfolgsfall und `EXIT_FAILURE` für den Fehlerfall.

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    /* Greet the World */
    printf("%s", "Hello, World!\n");
    return EXIT_SUCCESS;
}
```

Kapitel 2

Compiler

2.1 Video zum Kapitel [20]

Link zu YouTube

2.2 Compilieren und Linken bei Java [21]

- Jede Java-Klasse ist *selbstbeschreibend*
  - Verwender kann alle *Meta-Informationen* aus der .class-Datei beziehen
  - `javap` dient dazu, diese Informationen auszugeben
- Java kennt *keinen Linker*, sondern nur einen Compiler
- Klassen werden
  - von der Java VM *dynamisch* bei Bedarf geladen
  - erst geladen, wenn sie das erste Mal benötigt werden
  - erst bei Bedarf von der VM intern gelinkt
- Java VM sucht die Klassen auf dem **Klassenpfad** (classpath) (VM-Option `-cp` oder `-classpath`)

Klassenpfad

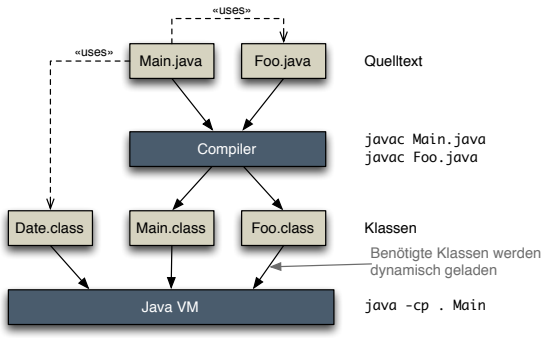
Das Übersetzen von Programmen mit Java folgt anderen Prinzipien als das Übersetzen und Linken in klassischen Programmiersprachen.

In Java existieren keine zusätzlichen Header-Dateien, sondern der Verwender eine Klasse kann alle Informationen, die er benötigt, aus der Klasse selbst beziehen. Java-Klassen sind somit selbstbeschreibend und enthalten sowohl die Definition als auch die Deklaration von Methoden und Variablen. Man kann sich diese Meta-Informationen mit dem Tool `javap` ansehen.

javap

Bei Java existiert kein Linker, d. h. die kompilierten Klassen werden nicht in einem zusätzlichen Schritt zu einer ausführbaren Datei gebunden. An Stelle des Linkers fungiert die Java-VM, die während des Ablaufs eines Programms dynamisch alle Klassen nachlädt, die von diesem benötigt werden **(dynamic classloading)**. Natürlich müssen auch im Falle von Java die Beziehungen zwischen den Klassen aufgelöst werden und symbolische Methodenaufrufe durch echte ersetzt werden. Dieses Binden oder Linken wird aber von der VM intern zur Laufzeit durchgeführt und ist daher für den Verwender nicht sichtbar.

Damit die Java-VM die benötigten Klassen finden kann, kann man mithilfe der Kommandozeilenoption `-classpath` festlegen, wo sie überall nach Klassen suchen soll.



2.3 Klassische Programmiersprachen [23]

In klassischen Programmiersprachen (C, C++, …) besteht die Programmerzeugung aus zwei Schritten

- **Übersetzen (compile)** – die Quelldateien werden einzeln in plattformspezifischen Maschinencode übersetzt — Übersetzen
- **Binden (link)** – alle Teile des Programms werden zu einer einzigen ausführbaren Datei (**Executable**) zusammengebunden — Binden, Executable
  - **statisches Binden** – das erzeugte Executable enthält alle Programmteile und alle Bibliotheken — statisches Binden
  - **dynamisches Binden** – nur Teile des Programms sind im Executable enthalten, andere Teile (meistens Bibliotheken) werden bei Bedarf zur Laufzeit nachgeladen — dynamisches Binden

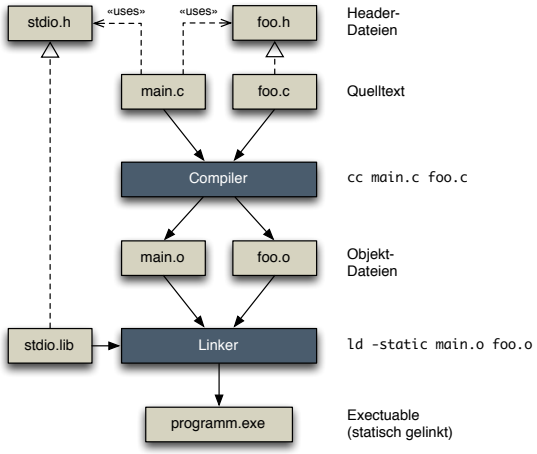
Klassische Programmiersprachen (C, C++, Modula 2, Pascal) erzeugen ausführbare Dateien, die spezifisch für die jeweilige Plattform sind, d. h. sie enthalten Maschinenbefehle für den Prozessor der Maschine. Um die Programmierung modularisieren zu können, wird die Erzeugung einer solchen Binärdatei in zwei Schritte aufgeteilt:

- Im ersten Schritt werden die vorhandenen Quelldateien vom Compiler in Maschinencode für die Plattform übersetzt. Hierbei werden allerdings die Beziehungen zwischen den Quelldateien noch nicht aufgelöst, da sie einzeln übersetzt werden. Beziehungen (meist Funktionsaufrufe), die sich von einer auf die andere Quelldatei beziehen werden nur

vermerkt aber noch nicht miteinander verbunden. Das Ergebnis eines solchen Schrittes bezeichnet man als **Objektdatei (object file)**. Objektdatei

- In einem zweiten Schritt, sammelt ein weiteres Werkzeug, der **Linker**, alle benötigten Objektdateien zusammen und löst die Querbeziehungen auf, indem er die symbolischen Referenzen durch echte Funktionsaufrufe ersetzt (binden). Als Ergebnis erhält man eine **Programmdatei (executable)**. Linker Programmdatei

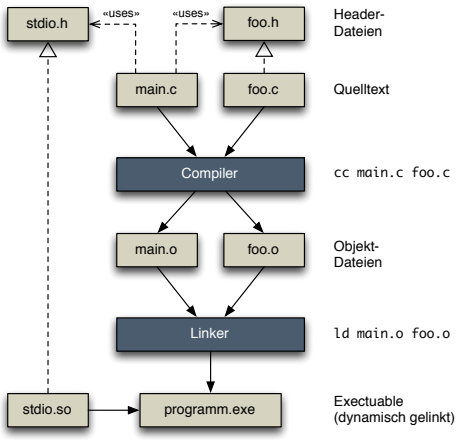
Beim Linken besteht die Möglichkeit entweder alle Objektdateien statisch zu einem einzigen Executable zu binden (statisches Linken) oder aber erst zu Laufzeit einzelne Teile nachzuladen (dynamisches Linken).



Beim **statischen Linken** werden alle notwendigen Objektdateien zu einem großen Executable zusammengebunden. Wenn das Programm Funktionen aus Bibliotheken (z. B. der C-Standard-Bibliothek) verwendet, werden diese ebenfalls in das Executable kopiert. statischen Linken

Der Vorteil eines statisch gelinkten Programms ist, dass es keine Abhängigkeiten zum System hat, auf dem es ausgeführt wird. Es bringt alle seine Bibliotheken und Funktionen mit. Der Nachteil ist, dass das Programm deutlich größer ist und das Betriebssystem keine Möglichkeit hat, die Standardbibliotheken ressourcenschonend nur einmal zu Laden. Somit sind sowohl die Binärdatei als auch der Speicherverbrauch deutlich größer.

Für C-Programme verwendet man normalerweise kein statisches Linken. In Go ist es aber der Normalfall, weil Go das Ziel hat, das gesamte Programm in einer einzigen Datei auszuliefern.



Im Gegensatz zum statischen Linken, werden beim **dynamischen Linken** nur die Objektdateien des Programms selbst zusammengefügt, alle externen Bibliotheken aber nicht. Der Linker setzt an die Stellen, an denen externe Bibliotheken gerufen werden, spezielle Verweise, die erst beim Laden des Programms von einem weiteren Programm, dem **Dynamischen Linker (dynamic linker)**, aufgelöst werden.

dynamischen Linken

Dynamischen Linker

2.4 Beispielprogramm kompilieren [26]

- Quelltext zu einer Objektdatei `hello_world.o` (nicht ausführbar) kompilieren

```
$ gcc -Wall -c hello_world.c
```

- Objektdatei (`hello_world.o`) zu einem Executable (`hello_world`) binden

```
$ gcc -o hello_world hello_world.o
```

- Programm ausführen

```
$ ./hello_world
Hello, World!
$
```

Man kann die Datei auch direkt in einem Schritt zu einem Executable compilieren, und zwar mit dem Aufruf: `gcc -o hello_world hello_world.c`. Hierbei ist die Angabe des Zieldateinamens mit `-o` wichtig, da andernfalls eine Datei mit dem Namen `a.out` für das fertige Programm erstellt wird. Der Name `a.out` hat historische Gründe und steht als Abkürzung für *assembler output*. Ursprünglich bezeichnete es ein Dateiformat in frühen Unix-Systemen, ist

inzwischen aber nur noch ein Relikt aus alten Zeiten, da unter Unix die ausführbaren Dateien im ELF-Format vorliegen.

Die Option `-Wall` bittet den C-Compiler darum, möglichst viele Warnungen (`-W`) auszugeben, damit Programmierfehler schneller erkannt werden. Wie wir noch sehen werden, kann man sich in C schnell selbst ein Bein stellen und `-Wall` sorgt dafür, dass der Compiler, zumindest in vielen Fällen, eine Warnung hiervor ausgibt. `-Wall`

Mit `-c` weist man den Compiler an, die Datei in eine Objektdatei nur zu compilieren und noch nicht zu linken. Obwohl der Linker ein getrenntes Programm ist (`ld` beim gcc) ist der Compiler normalerweise so nett, ihn für uns aufzurufen. Mit `-c` unterbinden wir das. `ld`

Wenn man dem Compiler anstatt einer Quelldatei eine Objektdatei (Endung `.o`) vorlegt, dann ruft er den Linker auf und linkt die Datei für uns. Den Namen der Zieldatei geben wir mit `-o` an.

Wir können die Datei auch linken, ohne den Compiler zu verwenden, indem wir den Linker selbst aufrufen, dafür müssen wir dem Linker aber eine ganze Reihe von Informationen mitgeben, die der Compiler sowieso schon hat, z. B. wo sich die Bibliotheken auf dem Computer befinden.

Linkeraufruf von Hand

```
$ ld -v -plugin /usr/lib/gcc/x86_64-linux-gnu/10/liblto_plugin.so \
  -plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/10/lto-wrapper \
  -plugin-opt=-fresolution=/tmp/ccVVbwld.res \
  -plugin-opt=-pass-through=-lgcc \
  -plugin-opt=-pass-through=-lgcc_s \
  -plugin-opt=-pass-through=-lc \
  -plugin-opt=-pass-through=-lgcc \
  -plugin-opt=-pass-through=-lgcc_s \
  --build-id --eh-frame-hdr -m elf_x86_64 --hash-style=gnu ↩
      --as-needed \
  -dynamic-linker /lib64/ld-linux-x86-64.so.2 -pie -z now -z relro \
  -o hello_world \
  /usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/Scrt1.o ↩
      \
  /usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/crti.o \
  /usr/lib/gcc/x86_64-linux-gnu/10/crtbeginS.o \
  -L/usr/lib/gcc/x86_64-linux-gnu/10 \
  -L/usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu \
  -L/usr/lib/gcc/x86_64-linux-gnu/10/../../../../lib \
  -L/lib/x86_64-linux-gnu \
  -L/lib/../lib -L/usr/lib/x86_64-linux-gnu \
  -L/usr/lib/../lib \
  -L/usr/lib/gcc/x86_64-linux-gnu/10/../../.. hello_world.o -lgcc \
  --push-state \
  --as-needed -lgcc_s --pop-state -lc -lgcc --push-state \
  --as-needed -lgcc_s \
  --pop-state /usr/lib/gcc/x86_64-linux-gnu/10/crtendS.o \
  /usr/lib/gcc/x86_64-linux-gnu/10/../../../x86_64-linux-gnu/crtn.o
```

Deswegen lassen wir den Compiler den Linker aufrufen und überlasen ihm die ganze Komplexität.

2.5 Beispiel: Mehrere Dateien [27]

Ein C-Programm besteht normalerweise aus mehreren Dateien, die man einzeln compilieren kann und die dann vom Linker zu einem ausführbaren Programm gebunden werden. Im folgenden Beispiel haben wir die Begrüßung in eine eigene Datei `greeter.c` ausgelagert, die eine passende Funktion `greeter` anbietet.

greeter.c

```c
#include <stdio.h>
#include "greeter.h"

void greeter(char* name) {
    printf("Hello, %s!\n", name);
}
```

greeter.h

```c
#ifndef GREETER_H
#define GREETER_H
void greeter(char* name);
#endif
```

main.c

```c
#include "greeter.h"

int main(int argc, char** argv) {
    greeter("Thomas");
}
```

Kompilieren und linken

```
$ cc -c main.c
$ cc -c greeter.c
$ cc -o hello_world main.o greeter.o
$ ./hello_world
Hello, Thomas!
$
```

Wenn man den Code ansieht, tauchen sofort vier Fragen auf:

1. Warum gibt es neben `greeter.c` auch noch ein `greeter.h`?
2. Warum inkludiert `main.c` die Datei `greeter.h`?

3. Wieso inkludiert `greeter.c` die Datei `greeter.h`, die doch nur noch einmal die Funktionsdeklaration enthält?
4. Was soll das `#ifndef`-Zeugs?

Zur Beantwortung dieser Fragen muss man verstehen, wie der C-Compiler die Objektdateien erzeugt und was der Linker genau macht.

Jede Quelldatei wird getrennt kompiliert

Jede Quelldatei wird getrennt kompiliert, ohne Betrachtung irgend einer anderen Quelldatei. Verbindungen über Quelldateien hinweg kann erst der Linker erzeugen. Wenn der Compiler die Datei `main.c` compiliert, dann trifft er dort auf den Funktionsaufruf `greeter("Thomas");`. Diese Funktion ist in der Datei aber nicht bekannt, d. h. es käme zu einem Fehler, denn Funktionen, die es nicht gibt, können wir nicht aufrufen.

Wie sagen wir dem Compiler also, dass er chillen soll und die Funktion schon später vom Linker herangeschafft werden wird? Wir sagen es ihm, indem wir eine **Deklaration** der Funktion angeben, also den Kopf der Funktion ohne Rumpf zur Verfügung stellen. Dem Compiler reicht dies, denn er muss nicht wissen, was die Funktion macht, nur dass es sie später geben wird. Den Rest überlässt er dann entspannt dem Linker. Wir könnten also `main.c` einfach so aussehen lassen:

Deklaration

```c
/* Deklaration von Greeter */
void greeter(char* name);

int main(int argc, char** argv) {
    greeter("Thomas"); /* Verwendung von greeter */
}
```

Das wäre aber nicht besonders schlau, weil die Deklaration von `greeter` nicht zu `main.c` gehört, sondern zu `greeter.c`, weswegen wir sie in eine eigene *Header-Datei* (Dateiendung `.h`) auslagern und diese dann von `main.c` aus inkludieren.

Header-Dateien beschreiben die Schnittstelle

Jetzt kann man sich vorstellen, dass viele verschiedene Programme die Funktionalität von `greeter.c` mit der `greeter`-Funktion nutzen wollen. Deswegen ist es in C üblich, dass jede `.c`-Datei, die Funktionen für andere zur Verfügung stellt, diese über eine korrespondierende **Header-Datei** (`.h`) beschreibt. Möchte ich die Funktionen nutzen, inkludiere ich die Header-Datei, nutze die Funktionen und der Linker macht den Rest.

Header-Datei

Nichts anderes macht `greeter.c` in der ersten Zeile mit `#include <stdio.h>`: Hier wird ebenfalls eine Header-Datei inkludiert, nämlich eine der C-Standardbibliothek, welche die `printf`-Funktion zur Verfügung stellt. In Zeile 332 der Datei `stdio.h` findet sich dann `extern int printf (...);` also die Deklaration der `printf`-Funktion. Das `extern` können Sie ignorieren da es bei einer Funktions-Deklaration redundant ist, man kann also sowohl `extern int f();` als auch `int f();` schreiben. Der fehlende Funktionsrumpf impliziert das `extern`.

Warum inkludieren wir `stdio.h` mit spitzen Klammern, `greeter.h` aber mit Anführungszeichen? Die Antwort ist, dass Header-Dateien, die vom System und seinen Bibliotheken zur Verfügung gestellt werden, mit <> inkludiert und vom Compiler automatisch an festgelegten Stellen gesucht werden. Header-Dateien, die vom User geschrieben wurde, werden mit `""` inkludiert und werden im aktuellen Verzeichnis gesucht. Man kann beim Inkludieren mit `""` auch relative Pfade verwenden, z. B. `#include "../welcome/greeter.h"`.

Deklaration und Definition müssen übereinstimmen

Die Datei `greeter.c` enthält die **Definition** der Funktion `greeter`. Beachten Sie bitte den Unterschied:

Definition

- *Deklaration*: Beschreibung der Schnittstelle der Funktion (Rückgabetyp, Name, Parameter)
- *Definition*: Implementierung der Funktion = Deklaration + Funktionsrumpf

Eine Funktion darf *beliebig oft deklariert* aber *nur einmal definiert* werde. Diese spitzfindige Behandlung der beiden Begriffe kennen wir aus Java nicht, weil dort – bedingt durch das Design der Sprache – Definition und Deklaration fast immer gleichzeitig passieren: Die Ausnahme sind abstrakte Methoden.

Warum inkludiert `greeter.c` jetzt die Header-Datei mit der Deklaration einer Funktion, die sofort danach definiert wird? Die Antwort ist: So wird verhindert, dass die Funktionen sich aus Versehen unterscheiden, denn Definition und Deklaration einer Funktion *gleichen Namens* müssen immer übereinstimmten. Wenn ich also aus Versehen einen anderen Rückgabetyp oder andere Parameter in `greeter.c` als in `greeter.h` benutzen würde, fiele das nicht auf, wenn ich nicht `greeter.h` in `greeter.c` inkludieren. Der Linker interessiert sich nämlich nicht für die Parameter und Rückgabetypen und linkt die Funktionen ausschließlich anhand der Namen. Das böse erwachen kommt dann erst zur Laufzeit.

Das folgende Beispiel zeigt das Problem:

greeter.c

```c
#include <stdio.h>
/* Hier fehlt das #include von greeter.h */

void greeter(char* name, int age) {
    printf("Hello, %s! You are %d years old!\n", name, age);
}
```

greeter.h

```c
void greeter(char* name);
```

main.c

```c
#include "greeter.h"

int main(int argc, char** argv) {
```

```
    greeter("Thomas");
}
```

Wenn man das folgende Programm compiliert – was problemlos klappt – und ausführt, dann bekommt man einen sehr seltsamen Output:

```
$ cc -Wall -c greeter.c
$ cc -Wall -c main.c
$ cc -o hello_world greeter.o main.o
$ ./hello_world
Hello, Thomas! You are 2006438408 years old!
```

Woher diese unglaubliche Zahl kommt, können Sie einmal selbst überlegen.

Durch das Inkludieren der eigenen Header-Datei verhindert man, dass es zu einer Abweichung von Deklaration und Definition kommt:

greeter.c

```
#include <stdio.h>
#include "greeter.h"

void greeter(char* name, int age) {
    printf("Hello, %s! You are %d years old!\n", name, age);
}
```

```
cc -Wall -c greeter.c
greeter.c:4:6: error: conflicting types for 'greeter'
    4 | void greeter(char* name, int age) {
      |      ^~~~~~~
In file included from greeter.c:2:
greeter.h:3:6: note: previous declaration of 'greeter' was here
    3 | void greeter(char* name);
      |      ^~~~~~~
```

Doppelte Includes vermeiden

Solange eine Header-Datei nur Deklarationen enthält ist es kein Problem, wenn sie mehrfach eingebunden wird. Meist passiert eine solche mehrfache Einbindung indirekt: A inkludiert B und H, B inkludiert H. Probleme entstehen aber, wenn Header-Dateien Dinge enthalten, die nur einmal vorkommen dürfen (z. B. `#define`). Um hier Konflikte zu verhindern, verwendet man die folgende Konstruktion, um sicherzustellen, dass jede Header-Datei nur einmal vorkommt:

```
#ifndef NAME_DER_DATEI
#define NAME_DER_DATEI
...
```

```
#endif
```

Der Bereich zwischen `#ifndef` (If Not Defined) und `#endif` wird nur dann ausgeführt, wenn die Variable `NAME_DER_DATEI` nicht gesetzt ist. Danach wird die Variable sofort gesetzt, sodass beim nächsten Auftreten desselben Konstruktes, der Bereich zwischen `#ifndef` und `#endif` ignoriert wird.

2.6 Exports und Imports [29]

Wie schafft es der Linker, die verschiedenen Dateien zusammenzubringen und am Ende ein funktionierendes Executable zu erzeugen? Die Antwort liegt darin, dass jede Objektdatei entsprechende Informationen liefert.

Jede Objektdatei liefert dem Linker

- **Exports**: Funktionen, die sie anderen Objektdateien zur Verfügung stellen kann — Exports
- **Imports**: Funktionen, die sie von anderen Objektdateien benötigt — Imports
- Der Überbegriff für Exports und Imports ist **Symbole** — Symbole

Linker sammelt die Informationen und baut das fertige Executable zusammen

Wir werden später noch sehen, dass es sich bei den Exports und Imports nicht nur um Funktionen handel kann, sondern auch Variablen als Symbole exportiert und importiert werden können.

Man kann das Tool `rabin2` aus dem Radare2-Werkzeugset benutzen. Mit `rabin2` kann man sich Informationen zu Objektdateien und Executables ausgeben lassen. Im folgenden Beispiel mit der Option `-Ei` die Exports und Imports bzw. noch korrekter ausgedrückt die exportierten und importierten Symbole.

Imports und Exports von greeter.o

```
$ rabin2 -Ei greeter.o
[Imports]
nth vaddr      bind   type   lib name
5   ---------- GLOBAL NOTYPE     printf

[Exports]
nth paddr      vaddr      bind   type size lib name
4   0x00000040 0x08000040 GLOBAL FUNC 42       greeter
```

- die Funktion `printf` wird importiert
- die Funktion `greeter` wird exportiert

Das Beispiel zeigt, dass die Objektdatei `greeter.o` eine Funktion namens `greeter` exportiert und die Funktion `printf` importiert. Man sieht ebenfalls, dass die Parameter nicht Teil der

Informationen sind, die man über die exportierten und importierten Funktionen bekommt. Tatsächlich interessieren diese den Linker auch nicht, da jede Funktion einen eindeutigen Namen haben muss (es gibt keine überladenen Funktionen) und die Funktionen sich selbst darum kümmern müssen, die richtigen Parameter beim Aufruf zur Verfügung zu stellen.

Import und Exports von main.o

```
$ rabin2 -Ei main.o
[Imports]
nth vaddr      bind   type   lib name
5   ---------- GLOBAL NOTYPE     greeter

[Exports]
nth paddr      vaddr      bind   type size lib name
4   0x00000040 0x08000040 GLOBAL FUNC 26       main
```

- die Funktion `greeter` wird importiert
- die Funktion `main` wird exportiert

`main.o` importiert die Funktion `greeter` und exportiert selbst die Funktion `main`. Letztere wird dann später beim Start des Programms aufgerufen.

Man kann an den Ausgaben nicht erkennen, woher die Funktionen importiert werden. Es ist Aufgabe des Linkers, alle vorhandenen Objektdateien und Bibliotheken nach den passenden Funktionen zu durchsuchen und dann die entsprechenden Verbindungen herzustellen. Die Objektdateien enthalten diese Information nicht.

Schaut man sich als Letztes noch die Exports und Imports vom fertig gelinkten Executable an, dann sieht man (gekürzt) folgendes:

Imports und Exports des fertigen Executables

```
$ rabin2 -Ei hello_world
[Imports]
nth vaddr      bind   type   lib name
3   0x00001030 GLOBAL FUNC       printf

[Exports]
nth paddr      vaddr      bind   type   size lib name
18  0x00001040 0x00001040 GLOBAL FUNC   38       _start
19  0x00001153 0x00001153 GLOBAL FUNC   42       greeter
21  0x00001139 0x00001139 GLOBAL FUNC   26       main
```

Die Funktion `printf` ist weiterhin ein Import des Programms, weil diese Abhängigkeit vom Linker nicht befriedigt wurde, sondern die Funktion zur Laufzeit aus der C-Standard-Bibliothek dynamisch nachgeladen wird. Hierzu wird beim Programmstart nach der Funktion gesucht und sie wird dynamisch dem Programm zur Verfügung gestellt, bevor die `main`-Funktion läuft. Hierfür ist eine spezielle Komponente im Betriebssystem verantwortlich, die man *dynamischen Linker* oder unter Linux als **Program Interpreter** bezeichnet.

Program Interpreter

Ein interessanter Export ist die Funktion `_start`. Diese kommt in den Objektdateien nicht vor, ist aber der echte Einstiegspunkt in das Programm. `main` ist nur für C-Programme die erste Funktion, die gestartet wird. Aber wie verhält es sich mit anderen Programmiersprachen? Tatsächlich ist es so, dass das Betriebssystem beim Start eines Programms nicht die Funktion `main` anspringt, sondern die Funktion `_start`. Es ist Aufgabe des Compilers der jeweiligen Programmiersprache, eine `_start`-Funktion zu generieren, die dann im Fall von C eben die `main`-Funktion aufruft.

2.7 Makefile [32]

- Kompilieren und Linken ist komplex ⇒ man fasst Kommandos in einem **Makefile** zusammen

Makefile

```
# Makefile für das Hello-World-Programm. Achtung: Einrückung muss mit Tabs erfolgen
.PHONY: all clean
all: hello_world

clean:
    rm -f main.o greeter.o hello_world

main.o: main.c greeter.h
    cc -Wall -c main.c

greeter.o: greeter.c greeter.h
    cc -Wall -c greeter.c

hello_world: main.o greeter.o
    cc -o hello_world main.o greeter.o
```

C-Programme von Hand zu kompilieren ist eine Qual. Zwar könnte man dem Compiler immer wieder einfach alle C-Dateien vorwerfen (`cc -o hello_world main.c greeter.c`), dies würde aber unnötig Zeit verschlingen. Der Compiler würde alle Dateien immer wieder neu kompilieren, auch wenn sich nichts geändert hat.

Dieses Problem wird vom Werkzeug `make` adressiert, das fast so alt wie C ist und konzeptionell hervorragend dazu passt. Eine Make-Datei besteht aus:

- **Targets**: Dateien, die erzeugt werden sollen
- **Prerequisites**: Dateien, von denen das Target abhängt
- **Recipes**: Kommandos die ausgeführt werden, um die Targets zu erzeugen

Targets

Prerequisites

Recipes

Betrachtet man die folgenden Zeilen:

```
main.o: main.c greeter.h
    cc -Wall -c main.c
```

Dann ist

- `main.o` das Target
- `main.c` und `greeter.h` die Prerequisites
- `cc -Wall -c main.c` das Recipe

Man gibt beim Aufruf von `make` ein Target an (tut man das nicht, wird das erste Target in der Datei – normalerweise `all` genannt – angesprungen). Jetzt schaut `make`, ob alle Prerequisites existieren. Ist dem nicht so, sucht es nach einem Target, das das Prerequisites erzeugt. Ist ein Prerequisite vorhanden, wird geprüft, ob es neuer als das Target ist. (Falls das Target nicht existiert, wird es einfach als älter als alle Prerequisites betrachtet.) Ist das Target älter als mindestens eines der Prerequisites, wird das Recipe ausgeführt, um das Target auf den neuesten Stand zu bringen.

Durch diesen Mechanismus sorgt `make` dafür, dass immer nur dann die Recipes ausgeführt werden, wenn sich eine Änderung an den Dateien ergeben hat, die für ein Target notwendig sind. Bei großen Projekten führt das zu einer erheblichen Beschleunigung des Entwicklungsprozesses, weil nicht immer alles compiliert werden muss.

Das erste Target (hier `all`) im Makefile wird automatisch angesprungen. Wenn es sich bei einem Target nicht um eine Datei, sondern ein generisches Target wie `all` oder `clean` handelt, wird es als `.PHONY` gekennzeichnet. Dies verhindert, dass das Target nicht mehr angesprungen wird, wenn es eine gleichnamige Datei gibt.

Man kann in Makefiles auch Variablen verwenden, um sich einige Tipparbeit zu sparen. Im Folgenden ein einfaches Beispiel dazu.

```
CC = cc
COMPILER_OPTIONS = -Wall

.PHONY: all clean
all: hello_world

clean:
	@rm -f *.o hello_world

main.o: main.c greeter.h
	$(CC) $(COMPILER_OPTIONS) -c $<

greeter.o: greeter.c greeter.h
	$(CC) $(COMPILER_OPTIONS) -c $<

hello_world: greeter.o main.o
	$(CC) -o $@ $^
```

Die Variablen `$(CC)` und `$(COMPILER_OPTIONS)` sollten selbsterklärend sein. Verwirrender sind die anderen Variablen, die mit `$` beginnen:

- $<: Wird durch das erste Prerequisite ersetzt. Im vorliegenden Beispiel sind das die C-Dateien.
- $^: Wird durch *alle* Prerequisites ersetzt, im Beispiel die Objektdateien.
- $@: Wird durch das Target ersetzt.

automatischen Variablen

Diese Variablen werden als **automatischen Variablen** bezeichnet, weil man sie nicht selbst setzen muss, sondern sie von make automatisch mit Werten belegt werden. Ihre Verwendung hilft dabei, Wiederholungen in den Make-Files zu vermeiden.

Es gibt natürlich noch deutlich mehr Möglichkeiten, Makefiles zu verbessern und zu vereinfachen. Bei großen Projekten arbeitet man meist mit einem Haupt-Makefile und entsprechenden Unter-Makefiles für Subprojekte.

Wenn man den GNU C-Compiler verwendet, kann man sich die Abhängigkeiten (Dependencies) zwischen den einzelnen C-Dateien automatisch vom Compiler erzeugen lassen. Dazu dient die Option –MM. Ein solches Makefile für das aktuelle Projekt sähe dann wie folgt aus:

```
CC = cc
COMPILER_OPTIONS = -Wall
SOURCES := $(wildcard *.c)
OBJECTS := $(patsubst %.c,%.o,$(SOURCES))

.PHONY: all clean
all: hello_world .depend

clean:
	@rm -f *.o hello_world .depend

.depend: $(SOURCES)
	$(CC) $(COMPILER_OPTIONS) -MM $^ > "$@"

include .depend

%.o: %.c
	$(CC) $(COMPILER_OPTIONS) -c $<

hello_world: $(OBJECTS)
	$(CC) $(COMPILER_OPTIONS) -o $@ $^
```

Wenn man das Makefile erstellt hat, kann man sehen, dass nach Änderungen nur die notwendigen Dateien neu kompiliert werden.

```
$ make
cc -Wall -c greeter.c
cc -Wall -c main.c
cc -o hello_world greeter.o main.o

$ touch greeter.c
```

```
$ make
cc -Wall -c greeter.c
cc -o hello_world greeter.o main.o

$ touch main.c
$ make
cc -Wall -c main.c
cc -o hello_world greeter.o main.o
```

Das Kommando `touch` verändert den Zeitstempel einer Datei auf die aktuelle Zeit, lässt sie also für Make als verändert erscheinen. Man könnte die Datei natürlich auch öffnen, ändern und wieder abspeichern. Eine weitere Funktion von `touch` – die hier nicht gebraucht wird – ist, dass es die Datei auch anlegt, wenn sie nicht vorhanden ist.

Kapitel 3

Präprozessor

3.1 C-Präprozessor [36]

Als C entwickelt wurde, waren Hauptspeicher und Festplattenplatz knapp, was zu einem spartanischen Sprachumfang von C geführt hat. Beispielsweise war C nicht in der Lage, Dateien beim Kompilieren zu inkludieren, sodass schnell der Wunsch aufkam die Sprache dahingehend zu erweitern, damit Konstrukte wie wir sie heute von den Header-Dateien kennen überhaupt möglich wurden. Aus heute nicht mehr nachvollziehbaren Gründen wurden diese neuen Aufgaben aber an ein getrenntes Programm ausgelagert, den *Präprozessor*.

- **Präprozessor** (**preprocessor**) cpp löst Makros auf, bevor der Code an den Compiler geht

Präprozessor

```c
#include <stdio.h>
#define PI 3.141592653589793
#define area(r) ((r) * (r) * PI)

int main(int argc, char** argv) {
    printf("%f", area(10));
}
```

```
$ cpp < main.c
...
int main(int argc, char** argv) {
    printf("%f", ((10) * (10) * 3.141592653589793));
}
```

Der C-Präprozessor verarbeitet den Quelltext und die **Makros** *bevor* der Compiler mit der Arbeit beginnt. Korrekterweise muss man allerdings anmerken, dass der Präprozessor heute Bestandteil des Compilers geworden ist, um die Geschwindigkeit zu erhöhen. Er bleibt aber technisch vom eigentlichen Compiler getrennt und hat seine eigene Syntax. Über das Kommando cpp kann man den Präprozessor direkt aufrufen.

Makros

Der Präprozessor weiß auch nicht, welche Sprache er verarbeitet, wie das folgende Beispiel zeigt:

Datei: non_c.txt

```
#define BEWERTUNG super
#define NOTE 1

Die Vorlesung PR3 ist BEWERTUNG. Ich gebe ihr eine NOTE.
```

Ausgabe des Präprozessors

```
$ cpp non_c.txt
Die Vorlesung PR3 ist super. Ich gebe ihr eine 1 .
```

3.2 Syntax des Präprozessors [37]

Losgelöst von der Sprache C entwickelt, hat der Präprozessor seine eigene Syntax, die sich durch die #-Zeichen am Anfang jeder Direktive zeigt.

#

- Der Präprozessor wird mit **Direktiven** gesteuert, die immer mit # am Zeilenanfang beginnen

Direktiven

- Direktiven stehen meistens am Anfang der Datei
- Aufgaben
  - *Dateien* mit `#include` laden
  - *Konstanten* mit `#define` definieren
  - *Macros* mit `#define` definieren
  - *Bedingte Compilierung* mit `#ifdef`, `#ifndef` und `#endif`
- Compiler-Option `-E` zeigt die Ausgaben des Präprozessors an
- Präprozessor muss nicht mit dem Compiler benutzt werden (Programm `cpp`)

Wie wir später sehen werden, sollte man in modernem C-Code versuchen, den Präprozessor weitgehend zu vermeiden. Die Direktive, die man allerdings nicht umgehen kann, ist die `#include`-Direktive, die auch 53 Jahre nach der Erfindung von C die einzige Möglichkeit bleibt, Dateien zu inkludieren.

#include

- `#include <DATEINAME>`: Inkludiert einen **Standardheader**
  z. B. `#include <stdio.h>`

Standardheader

  - normalerweise ausgehend von `/user/include`
  - Suchpfad kann mit der Compileroption `-I` geändert werden
  - enthalten keine relativen Pfade (`..`)
- `#include "DATEINAME"`: Inkludiert einen **Header**

Header

z. B. `#include "gretter.h"`

- relativ zum aktuellen Verzeichnis
- üblicherweise für selbstgeschriebene Header
- enthalten häufig relative Pfade (`..`)

Als Daumenregel gilt:

- Von Ihnen geschriebene Header-Dateien → `#include "DATEINAME"`
- Header-Dateien von Bibliotheken, die Sie benutzen → `#include <DATEINAME>`

Makroersetzungen
`#define`

Der C-Präprozessor erlaubt es **Makroersetzungen** durchzuführen. Hierbei werden mit `#define` definierte Ausdrücke im Text gesucht und ersetzt. Die Makros können einfache Konstanten aber auch Ausdrücke mit Variablen sein.

Konstante

- `#define NAME [KONSTANTE]`: Definiert eine neue **Konstante**
  z. B. `#define PI 3.141592653589793`
  - Name wird *textuell* mit Wert im Programm ersetzt
  - wird benutzt um globale Konstanten zu definieren
  - Wert ist optional (im Zusammenhang mit `#ifdef`)
  - Name ist im Debugger nicht zu sehen!

Macro

- `#define NAME(p1, p2)AUSDRUCK`: Definiert ein **Macro**
  z. B. `#define area(r)((r)* (r)* PI)`
  - kann für *typunabhängigen Code* verwendet werden
  - ergibt **inline Funktionen** (heute oft unnötig)

inline Funktionen

- `#undef NAME`: Entfernt eine Konstantendefinition

Beachten Sie als erstes, dass es kein Gleichheitszeichen oder ähnliches zwischen dem Namen und dem Wert des Macros gibt. Name und Wert werden einfach durch ein Leerzeichen getrennt.

Im Anschluss an den Namen kann man Parameter (in Klammern) angeben, die im Macro verwendet werden. Der Präprozessor setzt dann die Parameter in das Macro ein, wenn er es im Text ersetzt. Wichtig ist allerdings zu verstehen, dass der Präprozessor hier keinerlei Intelligenz walten lässt und einfach stupide Ersetzungen durchführt. Angenommen, dass Macro sei `#define square ((x)*(x))` und Sie schreiben `printf("%d", square("Hallo"));` dann macht der Präprozessor daraus `printf("%d", (("Hallo")*("Hallo")));`.

Wenn man eine Macro verwendet, um globale Konstanten zu definieren, z. B. `#define MAX_VALUE 42`, dann erfolgt die Ersetzung vor dem Compilieren. D. h. in dem generierten Objekt-File sind alle Informationen zu dem Macro-Namen (`MAX_VALUE`) entfernt und dort steht nur noch der eigentliche Wert von 42. Dies kann das Debuggen von C-Programmen erschwerten.

Die – auf den ersten Blick unnötigen – Klammern im Macro `#define area(r)((r)* ↩ (r) * PI)` sind wichtig, damit es bei der Expansion des Makros nicht zu fehlerhaftem C-Code kommt.

Das folgende Beispiel zeig das Problem. Der Code

```
#define PI 3.141592653589793
#define area(r) (r * r * PI)

int main(int argc, char** argv) {
  double r = area(3.0 + 1.0)
}
```

wird expandiert zu

```
int main(int argc, char** argv) {
  double r = (3.0 + 1.0 * 3.0 + 1.0 * 3.141592653589793);
}
```

was auf jeden Fall falsch ist.

Für Macros gibt es zwei Argumente:

- Sie sind schneller als Funktionen: Da das Macro vor dem Compilieren ersetzt wird, finde an der Stelle kein Funktionsaufruf statt, sondern der Ausdruck wird direkt eingesetzt. Das Macro verhält sich somit wie eine Inline-Funktion. Moderne Compiler führen aber ohnehin bei der Optimierung des Codes ein **Inlining** von kurzen Funktionen durch, sodass dieses Argument heute nur begrenzt gültig ist.

  Inlining

- Man kann damit typunabhängigen Code erzeugen: Der Präprozessor weiß nichts von C-Datentypen, sodass man dasselbe Macro unabhängig vom Typ der Variablen einsetzen kann. Der C-Compiler würde dies nicht akzeptieren, da er eine strenge Typprüfung durchführt und für jeden Parameter einer Funktion ein Typ angegeben werden muss. Als Entwickler muss man sich aber darüber im Klaren sein, dass die Typsicherheit von C ein Feature ist, das man nicht leichtfertig unterlaufen sollte.

3.3 Beispiel: Typunabhängiger Code [41]

```
#include <stdio.h>
#define min(x, y) ((x) < (y) ? x : y)
#define max(x, y) ((x) > (y) ? x : y)

int main(int argc, char** argv) {
    printf("%d\n", max(4, 5));
    printf("%f\n", max(4.3, 5.7));
    printf("%ld\n", min(4L, 7L));
}
```

Dieses Beispiel zeigt, wie man ein Macro verwenden kann, um `max` und `min` so zu definieren, dass es für alle numerischen Datentypen verwendet werden kann. Allerdings kann man sich fragen, ob dies wirklich nötig ist: Die Entwicklerin kennt beim Aufruf der `min`- oder `max`-Funktion den Datentyp und könnte auch problemlos eine passende C-Funktion aufrufen.

```c
#include <stdio.h>

int int_min(int x, int y) { return (x < y ? x : y); }
long long_min(long x, long y) { return (x < y ? x : y); }
double double_min(double x, double y) { return (x < y ? x : y); }

int main(int argc, char** argv) {
    printf("%d\n", int_min(4, 5));
    printf("%f\n", double_min(4.3, 5.7));
    printf("%ld\n", long_min(4L, 7L));
}
```

Der Aufwand ist nur geringfügig höher und Macros werden vollständig vermieden.

Ein Grund Macros zu vermeiden liegt darin, dass *Macros keine Funktionen sind*.

3.4 Macros sind keine Funktionen [42]

Einer der gängigen Fehler bei der Verwendung von Macros ist, sie sich als C-Funktionen vorzustellen. Wegen der identischen Syntax bei der Verwendung (`area(20)`) kann dies leicht passieren. Sie sind aber keine Funktionen, sondern rein **textuelle Ersetzungen**.

textuelle Ersetzungen

- Macros sind keine Funktionen, sondern textuelle Ersetzungen!

```c
#include <stdio.h>
#define valid(x) ((x) > 0 && (x) < 20)

int main(int argc, char** argv) {
    int x = 0;
    if (valid(x++)) { /* tu was */ }
    printf("%d\n", x);
    x = 1;
    if (valid(x++)) { /* tu was */ }
    printf("%d\n", x);
}
```

Ausgabe

```
1
3
```

Die Ausgabe versteht man, wenn man das Macro textuell ersetzt. Dann sieht der Code wie folgt aus:

```c
int main(int argc, char** argv) {
    int x = 0;
    if (((x++) > 0 && (x++) < 20)) { }
    printf("%d\n", x);

    x = 1;
    if (((x++) > 0 && (x++) < 20)) { }
    printf("%d\n", x);
}
```

Wenn x > 0 ist, werden beide Bedingungen geprüft und die Variable wird zweimal inkrementiert. Dies liegt daran, dass das Macro kein Funktionsaufruf ist, bei dem die Parameter auf den Stack geschrieben werden, sondern eine textuelle Ersetzung, bei der `x++` einfach in das Macro eingesetzt wird.

3.5 Bedingte Compilierung [43]

Mithilfe des C-Präprozessor kann man eine **Bedingte Compilierung** durchführen, d. h. abhängig von dem Wert eines Ausdrucks werden einzelne Teile des Codes ausgeblendet oder eben nicht.

Bedingte Compilierung

Syntax

```c
#if AUSDRUCK
  /* code 1 */
#elif AUSDRUCK2
  /* code 2 */
#else
  /* code 3 */
#endif
```

- Präprozessor prüft `AUSDRUCK` im `#if` — `#if`
- wenn der Ausdruck wahr ist, wird der erste Code-Bereich ausgegeben
- wenn er falsch ist, werden die nächsten `#elif`-Ausdrücke geprüft — `#elif`
- wenn kein Ausdruck wahr ist, wird der `#else`-Bereich ausgegeben — `#else`

Welche Arten von Ausdrücken sind in einem `#if` erlaubt? Es sind alle C-Ausdrücke erlaubt, die sich als arithmetischer Ausdruck auffassen lassen, also zu `0` oder einem anderen Zahlen-Wert ausgewertet werden:

- Zeichenvergleiche (keine Anführungszeichen!)
  `#if OS == linux`

- Numerische Vergleiche
  `#if VERSION == 5`
  `#if VERSION > 5`
  `#if VERSION + 2 > 5`
- Logische Operationen
  `#if VERSION == 5 && OS == linux`
  `#if !(VERSION == 5)`
- Prüfung auf Existenz eines Macros mit `defined` (besser `#ifdef` und `#ifndef`)
  `#if defined OS`

Die Macros werden vor dem Vergleich einfach wieder textuell ersetzt, d. h. aus

```
#define OS linux
#define VERSION 5
#if OS == linux && VERSION == 5
```

wird

```
#if linux == linux && 5 == 5
```

Da die Macros vom C-Präprozessor und nicht vom C-Compiler ausgewertet werden, können keine Ausdrücke verwendet werden, die sich auf dem C-Typsystem abstützen, z. B. ist kein `sizeof()`-Operator verfügbar.

Wenn ein Macro nicht gesetzt ist, also undefiniert, dann wird in numerischen Vergleichen einfach angenommen, dass es den Wert `0` hat. Somit kann man `#if defined BUFSIZE && BUFSIZE >= 1024` vereinfachen zu `#if BUFSIZE >= 1024`.

Will man Code einfach von der Compilierung ausnehmen aber keinen Kommentar verwenden (z. B. weil Kommentare nicht geschachtelt werden können), kann man einfach `#if 0` verwenden:

```
#if 0
/* wird nicht ausgewertet */
#endif
```

Die bedingte Compilierung wird häufig benutzt, um plattformspezifischen Code zu schreiben.

Das folgende Beispiel zeigt, wie man die bedingte Compilierung verwendet, um Code für verschiedene Betriebssysteme zu schreiben.

```
#define OS linux

#if OS == linux
    /* Linux Code */
```

```
#else
    /* Was anderes als Linux */
#endif /* OS == linux */
```

Das Macro `OS` würde in der Praxis nicht im Quelltext definiert, sondern von außen über den Compiler mit einem entsprechenden Schalter gesetzt. Der C-Compiler setzt bereits eine Reihe von Macros, um plattformabhängigen Code einfach zu ermöglichen, z. B. gibt es ein Macro `__cplusplus`, das anzeigt, ob ein C++ oder „nur“ ein C-Compiler im Einsatz sind.

`__cplusplus`

Da man bei Präprozessor-Makros keine Schachtelung durch Einrückung darstellen kann, hat es sich eingebürgert, beim `#endif` zu notieren, zu welchem `#if` es gehört.

`#endif`

Will man nur auf die Existenz eines Macros prüfen, dann bietet sich mit `#ifdef` und `#ifndef` ein einfacher Mechanismus an.

`#ifdef`
`#ifndef`

Syntax

```
#ifdef NAME
    /* code 1 */
#else
    /* code 2 */
#endif
```

- Präprozessor prüft, ob `NAME` mit `#define NAME` definiert wurde und gibt abhängig den ersten oder zweiten Codeblock aus
- Für den umgekehrten Fall gibt es noch `#ifndef`

Der Code aus dem Beispiel ist identisch mit:

```
#if defined NAME
    /* code 1 */
#else
    /* code 2 */
#endif
```

Man kann anstatt `#if defined NAME` auch etwas eleganter `#if defined(NAME)` schreiben. So fügt sich die Konstruktion etwas besser in den Stil des umgebenden C-Programms ein.

3.6 Schutz vor doppelter Inklusion [46]

- Doppelte Definitionen sind in C problematisch
- Header-Dateien werden oft mehrfach (direkt und indirekt) inkludiert ⇒ Fehler
- Lösung: Header-File mit Präprozessor vor **doppelter Inklusion** schützen

doppelter Inklusion

myheader.h

```c
/* Konvention: DATEINAME */
#ifndef MYHEADER_H
#define MYHEADER_H
/* Deklarationen */
#endif /* MYHEADER_H */
```

Die Gründe für dieses Vorgehen wurden schon weiter oben detailliert besprochen, sodass hier auf die Details verzichtet wird.

3.7 Vordefinierte Variablen [47]

vordefinierte Macros

Der C-Standard erfordert, dass bestimmte, **vordefinierte Macros** vom Compiler automatisch gesetzt werden und vom C-Quelltext verwendet werden können. Einige der wichtigsten sind:

- `__FILE__`: Name der aktuellen Quelltextdatei
- `__LINE__`: laufende Zeile der aktuellen Quelltextdatei
- `__DATE__`: aktuelles Datum
- `__TIME__`: aktuelle Uhrzeit
- `__cplusplus`: gesetzt, wenn der Compiler C++ unterstützt
- …

Verwendung beispielsweise für Testausgaben von Variablen:

```c
printf("file %s, line %d, Wert: %ld\n", __FILE__, __LINE__, wert);
```

3.8 There will be Dragons [48]

Der Präprozessor ist ein Werkzeug, mit dem man viel machen, aber auch viel falsch machen kann.

- Der Präprozessor stammt aus der Computer-Steinzeit
- Der Präprozessor ist ein steter *Quell von Problemen*
- Man sollte Präprozessor-Direktiven nur sehr *sparsam einsetzen*
- Es gibt für fast alle Anwendungszwecke Alternativen direkt in C
  - `#define INT16 ...` → `typedef`
  - `#define MAXLEN 256` → `const`
  - `#define max(a,b)...` → Funktionen
  - Auskommentieren von Code → Versionsverwaltung

Eine per Macro durchgeführte Typdefinition kann man einfach durch ein `typedef` ersetzen: typedef

```
#define INT16 short

void f(INT16 param) {
    /* tu was */
}
```

kann man auch schreiben als:

```
typedef short int16;

void f(int16 param) {
    /* tu was */
}
```

Konstanten lassen sich durch C-Konstanten ersetzen:

```
#include <unistd.h>
#define MAXLEN 256

int main(int argc, char** argv) {
  char buffer[MAXLEN];

  read(0, buffer, MAXLEN);
}
```

wird zu

```
#include <unistd.h>
const int MAXLEN = 256;

int main(int argc, char** argv) {
  char buffer[MAXLEN];

  read(0, buffer, MAXLEN);
}
```

Bei der bedingten Compilierung ist der Präprozessor aber bis heute nicht wegzudenken. Wenn man ein Programm auf verschiedenen Plattformen compilieren möchte, kann man die plattformabhängigen Teile durch entsprechende Präprozessor-Anweisungen ein- bzw. ausblenden. Da man die Präprozessor-Makros beim Compilieren mit `-D` setzen kann, ist es möglich denselben Quelltext passend zu compilieren. -D

Index

C-Programmierung

Technische Hochschule Mannheim

Syntax

Prof. Thomas Smits

Stand 2025-07-21

Technische Hochschule mannheim

Wintersemester 2025/2026

Inhaltsverzeichnis

Kapitel 1

Kontrollstrukturen

1.1 Video zum Kapitel [5]

Link zu YouTube

1.2 Block [6]

Da C so viele moderne Programmiersprachen inspiriert hat, sind die Kontrollstrukturen ebenfalls sehr gängig. Wir kennen sie aus C++, Java oder C#. C bietet hier sehr wenig Überraschungen; kennt man die Kontrollstrukturen aus Java, so kennt man auch die aus C.

Ebenfalls bekannt aus vielen Sprachen ist das Konstrukt des Blocks, also einer Gruppierung von mehreren Statements zu einem gemeinsamen Ganzen.

- Anweisungen, die zwischen zwei geschweiften Klammern { } stehen, bilden einen **Block (compound statement)** — Block
- ein Block kann dort stehen, wo auch ein einzelnes Statement stehen kann (z. B. in Kontrollstrukturen)

```c
int i = 7;

{
  /* Dies ist ein Block */
  i++;
}
```

Der Block kann dort stehen, wo sonst ein einzelnes Statement stehen kann und er verhält sich dann – im Zusammenhang mit den Kontrollstrukturen – wie ein einzelnes Statement. Außerdem werden Blöcke noch benutzt, um den Rumpf von Funktionen anzugeben.

Sichtbarkeitsbereich

- Blocke bildet **Sichtbarkeitsbereich (scope)**
- Variablen, die in einem Block deklariert werden,
  - *sichtbar* nur im Block (und geschachtelten Blöcken)
  - *leben* nur solange, bis der Block verlassen wird

```
int a = 3;
{
    int b = 4;
    {
        int c = 5;
        a++; b++; c++;
    }
    a++; b++; /* c ist nicht sichtbar */
}
a++; /* c und b sind nicht sichtbar */
```

Im Beispiel ist die Variable a in beiden Blöcken sichtbar und lebt bis zum Ende des Beispiels. Die Variable b ist aber nur innerhalb des äußeren und inneren Blocks vorhanden. c sogar nur im innersten Block.

Die genaue Lebensdauer der Variable können wir nicht bestimmen, da wir außerhalb ihres Sichtbarkeitsbereichs keinen Test machen können, ob sie noch vorhanden ist. Es ist dem Compiler freigestellt, wann er die Variable freigibt. Üblicherweise wird er aus Effizienzgründen alle lokalen Variablen einer Funktion am Ende der Funktion freigeben, er kann es aber auch früher tun, wenn die Variable nicht mehr sichtbar ist.

Block Scope

Diese Eigenschaft, dass die Sichtbarkeit von Variablen auf den umgebenden Block beschränkt sind, bezeichnet man auch als **Block Scope**. Damit grenzen sich die C-artigen Sprachen von anderen Programmiersprachen, wie z. B. Ruby oder Python ab, bei denen die Variablen immer in der ganzen Funktion sichtbar sind.

1.3 Auswahl (if) [8]

Die einfachste Kontrollstruktur wird durch die Auswahlanweisung (`if`) zur Verfügung gestellt. Ist die Bedingung wahr, wird die Anweisung nach der Bedingung ausgeführt.

if-Anweisung

if-Anweisung bietet eine einseitige Auswahl

- Syntax: `if (BEDINGUNG){ TRUE-ZWEIG }`
- wenn die logische Bedingung erfüllt ist, wird der True-Zweig ausgeführt

```
if (antwort == 42) {
    printf("Die Antwort auf alle Fragen");
}
```

Bedingung ist vom Typ `int` mit `0` → false

Da sich Java bei der Syntax und den Kontrollstrukturen stark von C hat beeinflussen lassen, sollte es nicht verwunden, dass die C-Kontrollstrukturen denen von Java exakt gleichen. (Nur ist es natürlich in Wirklichkeit genau andersherum.)

Typ der Bedingung

Ein ganz großer Unterschied zu Java ist der **Typ der Bedingung**: Während in Java die Bedingung vom Datentyp `boolean` sein muss, ist sie in C vom Datentyp `int`. Der Grund ist, dass es in C bis vor kurzem keinen Datentyp für Wahrheitswerte gibt, sondern diese einfach als `int` dargestellt werden. Hierbei gilt, dass

- `0` → false
- `!= 0` → true

_Bool

Dieses Manko wurde im Standard adressiert und ein neuer Datentyp `_Bool` wurde in C99 eingeführt. Er heißt `_Bool`, weil man sich nicht getraut hat, den Datentyp `bool` zu nennen: Existierende Programme, die sich selbst einen `bool` definiert haben, hätten sonst nicht mehr compiliert.

bool

Der C11-Standard hat über die Header-Datei `stdbool.h` einen „echten“ `bool` eingeführt. In der Datei findet man dann aber auch nur:

```
#define bool  _Bool
#define true  1
#define false 0
```

bool

Erst seit dem C23-Standard gibt es einen echten Datentyp `bool` und die Schlüsselworte `true` und `false`.

Die Kontrollstrukturen fassen aber trotzdem weiterhin jeden Wert `!= 0` als `true` und `0` als `false` auf. Deswegen schreiben C-Programmierer völlig unverkrampft Code wie den folgenden – auch heute noch:

```
int i = 7;

while (i--) {
    printf("%d, ", i);
    /* Ausgabe: 6, 5, 4, 3, 2, 1, 0, */
}
```

- Man kann `if` auch ohne Block verwenden

```
if (lottogewinn)
    freuen();
```

- sehr gefährlich, wenn man weitere Statements hinzufügt

```
if (lottogewinn)
    freuen();
    job_kuendigen();
```

Im Beispiel wird die Funktion `job_kuendigen()` immer ausgeführt, da sich das `if` nur auf das unmittelbar nächste Statement bzw. den nächsten Block auswirkt. Da in diesem Fall kein Block verwendet wurde und die Einrückung für den Compiler unerheblich ist, wird `freuen()` nur aufgerufen, wenn der Lottogewinn erfolgt ist, `job_kuendigen()` aber immer.

Es ist deutlich defensiver, immer einen Block zu verwenden. Bis auf zwei zusätzliche Zeichen und eine Zeile mehr macht dies keine zusätzlichen Aufwände, vermeidet aber diesen typischen Fehler. Deswegen schreiben viele Coding-Standards auch vor, dass man Kontrollstrukturen *immer* mit Block verwendet.

```
if (lottogewinn) {
    freuen();
    job_kuendigen();
}
```

Eine zweiseitige Auswahl, also den Entweder/Oder-Fall kann man mit `if` / `else` erreichen.

if-else-Anweisung bietet eine zweiseitige Auswahl

if-else-Anweisung

- Syntax: `if (BEDINGUNG){ TRUE-ZWEIG } else { ELSE-ZWEIG }`
- wenn Bedingung erfüllt ist, wird der True-Zweig ausgeführt, andernfalls der Else-Zweig

```
if (antwort == 42) {
    printf("Die Antwort auf alle Fragen");
} else {
    printf("Keine Antwort");
}
```

1.4 Mehrfachverzweigung mit if-else-if [11]

Wenn man mehrere Fälle hintereinander abprüfen möchte, bietet sich neben dem `switch`/`case` (siehe weiter unten) eine Konstruktion mit `if` und `else if` an, man spricht dann von einer **Mehrfachverzweigung**.

Mehrfachverzweigung

if-else-if-Anweisung bietet eine Mehrfachverzweigung

if-else-if-Anweisung

- Verschachtlung mehrerer `if-else`-Anweisungen
- wird speziell formatiert, um besser lesbar zu sein
- Beliebig viele `else if`, nur ein `else`

Syntax

```
if (BEDINGUNG1) {
    TRUE-ZWEIG1
} else if (BEDINGUNG2) {
    TRUE-ZWEIG2
} else if (BEDINGUNG3) {
    TRUE-ZWEIG3
} else {
    ELSE-ZWEIG
}
```

Es gibt Programmiersprachen, die für diese Art der Mehrfachverzweigung ein eigenes Schlüsselwort (z. B. `elsif` oder `elif`) spendieren. C bleibt sich hier aber treu und stützt die Funktion einfach auf das vorhandene `if` / `else` ab. Die Mehrfachverzweigung ist nämlich einfach eine Kombination aus `if` und `else`, die in besonderer Weise formatiert wird, um übersichtlich zu bleiben. Hier macht man sich zu Nutze, dass sowohl das `if`, als auch das `else` keinen Block benötigen.

Das folgende Beispiel zeigt eine Mehrfachverzweigung für die Abfrage der Temperatur.

```
if (temp < 10) {
    printf("Viel zu kalt");
}
else if (temp < 20) {
    printf("Kalt");
}
else if (temp > 50) {
    printf("Viel zu warm");
}
else if (temp > 30) {
    printf("Warm");
}
else {
    printf("Alles super");
}
```

Man könnte das Beispiel auch anders formatieren, bei exakt gleicher Funktion, die Übersichtlichkeit wäre dann aber dahin.

if-else-if unübersichtlich formatiert

```
if (temp < 10) {
    printf("Viel zu kalt");
}
else {
    if (temp < 20) {
        printf("Kalt");
    }
    else {
        if (temp > 50) {
            printf("Viel zu warm");
        }
        else {
            if (temp > 30) {
                printf("Warm");
            }
            else {
                printf("Alles super");
            }
        }
    }
}
```

Die Reihenfolge der Bedingungen ist natürlich entscheidend, da nach dem ersten Treffer die anderen Zweige nicht mehr betrachtet werden. Es wäre also falsch, die ersten beiden Bedingungen zu tauschen, weil dann sowohl bei 9 als auch bei 19 Grad immer die Ausgabe „Kalt“ gemacht würde.

Falsche Reihenfolge

```
if (temp < 20) {
    printf("Kalt");
}
else if (temp < 10) {
    /* Wird nie erreicht! */
    printf("Viel zu kalt");
}
```

1.5 Mehrfachverzweigung mit switch [13]

In manchen Fällen möchte man eine Variable gegen eine Reihe von Werten testen und dafür eine kompaktere Form verwenden, als sie die Mehrfachauswahl mit `if-else` bietet. Hierfür bietet sich in C das aus anderen Sprachen ebenfalls bekannte `switch` / `case` an.

- **switch-Anweisung** bietet übersichtliche Form der Mehrfachverzweigung

switch-Anweisung

Syntax

```
switch (VARIABLE) {
    case WERT1:
        ANWEISUNGEN;
        break;
    case WERT2:
        ANWEISUNGEN;
        break;
    ...
    default:
        ANWEISUNGEN;
}
```

Besonderheiten

- `VARIABLE`: muss eine Ganzzahl (`int`) sein
- `WERT`: ein möglicher Wert der Variable als Literal oder Konstante
- `default`: Zweig der ausgeführt wird, wenn kein Wert vorher gepasst hat

default

```
switch (monat) {
    case 2: tage = 28; break;
    case 4: tage = 30; break;
    case 6: tage = 30; break;
    case 9: tage = 30; break;
    case 11: tage = 30; break;
    default: tage = 31;
}
```

Die Variable, die im `switch`-Statement verwendet wird (hier `monat`) muss entweder zuweisungskompatibel mit dem Typ `int` sein (`short`, `char`, `int`) oder es muss sich um einen Aufzählungstyp handeln (`enum`). Es ist nicht möglich, im `switch`-Statement Variablen von anderen Typen zu verwenden, d. h. `double`, `float` etc. sind nicht verwendbar.

Die Werte in den `case`-Ästen müssen Konstanten vom Typ `int` sein. Sie brauchen keinen expliziten Block zu schreiben, d. h. anders als beim `if` bezieht sich der `case`-Ast auf alle Statements bis zum nächsten `case` oder `break`. Gleichzeitig entsteht aber kein neuer Scope. Wollen Sie in einem `case` eine Variable definieren und dort verwenden, müssen Sie einen Block einsetzen.

```c
#include <stdio.h>

int main(int argc, char** argv) {
    int selection = 0;

    switch (selection) {
        case 0: {
            /* Block, damit Deklaration von k möglich wird */
            int k = 1;
            k--;
            printf("%d\n", k);
            break;
        }
        case 1:
            printf("%d\n", 1);
    }
}
```

Der `default`-Ast wird ausgeführt, wenn keine der Case-Konstanten gepasst hat.

Bei C fällt bei einem `switch`-Statement der Kontrollfluss durch alle `case`-Statements, die auf das `case` folgen, bei dem die Bedingung zutraf (**Durchfall-Logik**). Daher muss man immer explizit ein `break` hinter die `case`-Statements schreiben, wenn man dieses Verhalten nicht möchte, was fast immer der Fall ist.

Durchfall-Logik

break

Manchmal kann man das Durchfallen durch die `case`-Äste aber auch sinnvoll nutzen, z. B. für das bereits dargestellte Beispiel zu den Wochentagen.

Ausnutzen der Durchfall-Logik

```c
switch (monat) {
    case 2:
        tage = 28;
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        tage = 30;
        break;
    default:
        tage = 31;
}
```

Da das `switch`-Statement in C auf ganzzahlige Typen beschränkt ist, ist es nicht so vielseitig, wie in anderen Programmiersprachen. Ein Vorteil ist aber, dass der Compiler den erzeugten Maschinencode besser optimieren kann, als bei einem `if-else`, da er mit Sprungtabellen

arbeiten kann. Wenn man dem Compiler hier helfen möchte, das `switch` möglichst effizient zu realisieren, sortiert man die `case`-Labels in aufsteigender Reihenfolge.

1.6 Fragezeichen-Operator [16]

Eine syntaktische Beschränkung der bisher vorgestellten Kontrollstrukturen ist, dass sie keine Ausdrücke sind, d. h. sie liefern keinen Wert zurück. Es gibt Programmiersprachen, z. B. Ruby, bei denen das anders ist. In diesen Sprachen kann man z. B. folgendes schreiben:

Kontrollstruktur als Ausdruck in Ruby

```
bewertung = if temp < 10
                "Viel zu kalt"
            elsif temp < 20
                "Kalt"
            elsif temp > 50
                "Viel zu warm"
            else
                "Alles super"
            end
```

Diese Möglichkeit kennt C für die normalen Kontrollstrukturen nicht. Eine Ausnahme bildet hier der Fragezeichen-Operator, der eine Auswahl in Form eines Ausdrucks erlaubt.

- Bedingungen mit `if` und `case` können nicht in Ausdrücken eingesetzt werden
- für Verwendung in Ausdrücken gibt es einen speziellen **Fragezeichen-Operator**
- einziger Operator in C mit *drei* Operanden (**ternärer Operator**)
- Syntax: `BEDINGUNG ? TRUE-WERT : FALSE-WERT`

Fragezeichen-Operator

ternärer Operator

Beispiel: Absolutbetrag

Mit if

```
if (a < 0) {
    b = -a;
}
else {
    b = a;
}
```

Mit Fragezeichen-Operator

```
b = a < 0 ? -a : a;
```

Das Beispiel zeigt, dass es mit dem Operator einfacher ist, direkt aus der Auswahl einen Wert zu erhalten. Der Nachteil ist allerdings, dass es sehr unübersichtlich wird, wenn man mehr als zwei Fälle unterscheiden möchte, wie das folgende Beispiel zeigt:

```
bewertung = (temp < 10) ? "Viel zu kalt"
          : ((temp < 20) ? "Kalt"
          : ((temp > 50) ? "Viel zu warm"
          : "Alles super"));
```

Kapitel 2

Schleifen

2.1 Definition: Schleife [19]

In der Programmierung ist das wiederholte ausführen von Instruktionen eine zentrale Funktion, die man mithilfe von Schleifen realisiert.

> Eine *Schleife* **(loop)** ist ein Block von Anweisungen, der so lange mehrfach nacheinander ausgeführt wird, wie ein *Vergleichsausdruck*, mit dem die Schleife kontrolliert wird, wahr ist.
>
> G. Büchel, Praktische Informatik

2.2 while-Schleife [20]

Bei den Schleifen bietet C wieder die gewohnte Kost, die wir bereits aus anderen Programmiersprachen kennen; mit der `while`-Schleife als einfachste Form.

- **while-Schleife (while loop)** prüft *vor* jedem Durchlauf eine Bedingung (Vergleichsausdruck) — while-Schleife
- *kopfgesteuert* und *abweisend*
- Syntax: `while (BEDINGUNG){ ANWEISUNGEN }`

```c
while (antwort < 42) {
    antwort++;
}
```

Der Rumpf der `while`-Schleife wird nur betreten, wenn die Bedingung des Ausdrucks wahr ist. Wenn man eine Schleife wünscht, die mindestens einmal durchlaufen wird, muss man die `do/while`-Schleife wählen. Die Variable auf die getestet wird sollte richtig initialisiert sein und man sollte nicht vergessen, sie im Schleifenrumpf zu verändern, da man andernfalls in einer Endlosschleife landet.

Wie bei allen Kontrollstrukturen in C ist die Bedingung wieder ein Ausdruck, der als Ganzzahl interpretiert werden kann mit der Regel, dass `0` als `false` und alle anderen Werte als `true` gelten.

Wenn die Anzahl der Schleifendurchläufe schon im Voraus feststeht, sollten Sie die `for`-Schleife verwenden. Das hier gewählte Beispiel ist also nicht optimal, da man es besser mit `for` implementiert hätte. `while` ist gut geeignet für Schleifen, bei denen während des Durchlaufens erst festgestellt werden kann, wann der Abbruch erfolgen soll, z. B. bei linearen Suchen etc.

```c
/* Beispiel: Gib alle Zahlen von 1 bis 10 aus */
int i = 1;
while (i <= 10) {
    printf("%d\n", i);
    i++;
}
```

```c
/* Beispiel: Berechne Summe der Zahlen von 1 bis 10 */
int i = 1;
int summe = 0;
while (i <= 10) {
    summe += i;
    i++;
}
printf("%d\n", summe);
```

2.3 do-while-Schleife [22]

Die `while`-Schleife ist immer dann sinnvoll, wenn man den Rumpf nur betreten will, wenn die Bedingung mindestens ein mal `wahr` ist. In Fällen, in denen der Rumpf aber erst einmal durchlaufen werden soll, um dann am Ende die Bedingung zu prüfen, bietet sich die `do-while`-Schleife an.

do-while-Schleife (do-loop) ist *fußgesteuert* (*nicht abweisend*) do-while-Schleife

- Syntax: `do { ANWEISUNGEN } while (BEDINGUNG);`

```c
do {
    n = readNatuerlicheZahl();
} while (n >= 0);
```

Der Rumpf der `do`/`while`-Schleife wird immer mindestens einmal betreten, da der Test erst am Ende erfolgt. Wenn man eine Schleife wünscht, die die Bedingung vorher testet, muss man

die `while`-Schleife verwenden. Die Variable, auf die getestet wird, sollte im Schleifenrumpf verändert werden, da man andernfalls in einer Endlosschleife landet.

Wenn die Anzahl der Schleifendurchläufe schon im Voraus feststeht, sollten Sie die `for`-Schleife verwenden. Das hier gewählte Beispiel ist also nicht optimal, da man es besser mit `for` implementiert hätte.

```c
/* Beispiel: Gib alle Zahlen von 1 bis 10 aus */
int i = 1;
do {
    printf("%d\n", i);
    i++;
} while (i <= 10);
```

```c
/* Beispiel: Berechne Summe der Zahlen von 1 bis 10 */
int i = 1;
int summe = 0;
do {
    summe += i;
    i++;
} while (i <= 10);

printf("%d\n", summe);
```

2.4 for-Schleife [24]

Die vielseitigste Schleife wird in C durch die `for`-Schleife zur Verfügung gestellt. Sie bietet mehr Kontrolle über den Ablauf der Wiederholung und ist besonders gut geeignet, wenn man die Anzahl der Schleifendurchläufe im Vorfeld kennt.

for-Schleife (**for-loop**) ist *kopfgesteuerte* (*abweisende Schleife*) for-Schleife

- bietet mehr Möglichkeiten als die `while`-Schleife
- Syntax: `for (INIT; BEDINGUNG; UPDATE){ ANWEISUNGEN }`
  - `INIT`: initialisiert Variablen (optional)
  - `BEDINGUNG`: Schleife läuft, solange die Bedingung wahr ist
  - `UPDATE`: Operationen, die nach jedem Durchlauf durchgeführt werden

```c
int i;
for (i = 0; i < 10; i++) {
    /* tu was */
}
```

Wenn man Java gewöhnt ist, wird man sich wundern, warum die Variable `i` nicht in der `for`-Schleife deklariert wird. Der Grund liegt darin, dass bis zum C99-Standard Variablen nur am Anfang eines Blocks deklariert werden durften. Erst mit C99 kam die Möglichkeit, Variablen auch an anderer Stelle, z. B. in dem Initialisierungsteil einer `for`-Schleife, erstmals einzuführen. Der vorhergehende Standard (ANSI-C bzw. C89) kannte diese Erweiterung noch nicht.

Wenn Sie nicht wissen, auf welcher Plattform, mit welchem C-Compiler Ihr Code später einmal compiliert werden wird, sollten Sie auf die Deklaration im `for` verzichten.

Die drei Teile der `for`-Schleife sind alle optional, d. h. `for(;;);` ist gültiger C-Code und eine Endlosschleife.

```c
/* Beispiel: Gib alle Zahlen von 1 bis 10 aus */
int i = 0;
for (i = 1; i <= 10; i++) {
    printf("%d\n", i);
}
```

```c
/* Beispiel: Berechne Summe aller Zahlen von 1 bis 10 */
int summe = 0;
int i;

for (i = 1; i <= 10; i++) {
    summe += i;
}

printf("%d\n", summe);
```

Man kann über den Kommaoperator sogar mehr als eine Variable in der `for`-Schleife nutzen und Verändern. Die Details werden beim Operator weiter hinten erläutert.

2.5 Schleifenabbruch [26]

In manchen Fällen möchte man während der Iteration auf den Ablauf dieser Einfluss nehmen. Hierzu gibt es mit `break` und `continue` zwei passende Schlüsselwörter.

Aktueller Schleifendurchlauf kann abgebrochen werden

- `break` – Schleife wird ganz *verlassen* (Sprung an die Anweisung nach der Schleife) — `break`
- `continue` – beginnt sofort einen *neuen Durchlauf* (Sprung in die Prüfung der Bedingung) — `continue`

Wird häufig als schlechter Programmierstil angesehen. Alternativen

- Kontrollvariable als Ersatz für `break`

- zusätzliches `if` als Ersatz für `continue`

Die Verwendung von `break` und `continue` wird teilweise als schlechter Programmierstil angesehen, weil der Ablauf der Iteration hierdurch unübersichtlich werden kann. Andererseits gibt es Fälle, in denen ein gut platziertes `break` oder auch `continue` das Programm erheblich besser lesbar macht. Es kommt also darauf an, wann und wie man die Schlüsselworte benutzt.

```c
/* Suche kleinste Zahl, die durch 8 und 14 teilbar ist (mit break) */
for (int i = 1; i <= 8*14; i++) {
    if ((i % 8 == 0) && (i % 14 == 0)) {
        printf("%d\n", i);
        break;
    }
}
```

Das erste Beispiel zeigt, wie man durch ein `break` nach dem Finden der gesuchten Lösung die Schleife verlässt. Will man dies nicht, muss man – wie im zweiten Beispiel gezeigt – eine zweite Kontrollvariable einführen (`gefunden`) und diese dann im Erfolgsfall umsetzen.

```c
/* Suche kleinste1 Zahl, die durch 8 und 14 teilbar ist (ohne break) */
int gefunden = 0;
for (int i = 1; i <= 8*14 && !gefunden; i++) {
    if ((i % 8 == 0) && (i % 14 == 0)) {
        printf("%d\n", i);
        gefunden = 1;
    }
}
```

Die Variante mit dem `break` ist im vorliegenden Fall übersichtlicher als die zusätzliche Variable und hier deswegen vorzuziehen, weil klarer.

2.6 Das goto [28]

C erlaubt es mithilfe von `goto` unbedingte Sprünge durchzuführen. In Java ist `goto` zwar als Schlüsselwort reserviert aber nicht in der Sprache implementiert. In C hingegen hat es eine Funktion.

goto

- Anders als Java besitzt C noch einen `goto` Befehl
- `goto LABEL` springt zum Label `LABEL`
- nur innerhalb *derselben Funktion*

```c
int i = 0;
```

```
jump_here:

/* Schleife mit goto. Pfui! */
if (i < 5) {
    printf("i=%d\n", i);
    i++;
    goto jump_here;
}
```

Das Beispiel zeigt eine *falsche Verwendung* von `goto`, weil hier dasselbe Ziel auch mit einer normalen Schleife erreicht werden könnte und daher die unübersichtliche Konstruktion vollkommen unnötig ist.

Man verwendet `goto` in C hauptsächlich für Fehlerbehandlungsroutinen und Aufräumarbeiten im Falle eines Fehlers. So kann man den Code für die Fehlerbehandlung an einer zentralen Stelle in der Funktion halten und über `goto` anspringen.

Über Funktionsgrenzen hinweg kann in C mit der Funktion `_longjmp` gesprungen werden. Wegen der Auswirkungen auf den Stack ist die Anwendung aber nicht trivial und wird deswegen nicht weiter diskutiert.

Kapitel 3

Operatoren

3.1 Video zum Kapitel [30]

Link zu YouTube

3.2 Ausdruck [31]

Ein Ausdruck ist eine Kombination von Variablen, Literalen, Operatoren und Funktionen, die zusammen einen Wert ergeben. Ein Ausdruck kann eine Berechnung, einen Vergleich oder eine andere Art der Wertzuweisung darstellen.

Ein **Ausdruck (expression)** verknüpft Operanden mithilfe eines Operators

Ausdruck

- **Operand** ist der Wert (Variablen oder Literale) der verknüpft werden soll — Operand
- **Operator** legt die Art der Verknüpfung fest (z. B. Addition mit +) — Operator

```
19 + 7
vermoegenVonBillGates + 1000000
```

Variablen an sich sind nutzlos, solange man mit den Werten, die darin gespeichert sind, nicht rechnen oder arbeiten kann. Daher benötigt man, wie in der Mathematik *Operatoren*, die es erlauben aus bekannten Werten, neue Werte zu berechnen. Die Berechnung eines neuen Wertes erfolgt mithilfe eines *Ausdrucks*, z. B. `2 + 7`. Ein Ausdruck besteht aus

- *Operanden* die Werte miteinander verknüpfen (im Beispiel 2 und 7)
- *Operatoren* welche die Art der Verknüpfung festlegen (im Beispiel +)

Bei typsicheren Sprachen, wie C, ist eine der wichtigen Fragen, welchen *Typ* das Ergebnis der Operation hat. Bei vielen Operatoren hängt der Typ des Ergebnisses vom Typ der Operanden ab. So ist z. B. der Typ von `5 + 2` `int`, weil beide Operanden vom Typ `int` sind. Der Typ von `5 + 2.0` ist allerdings vom Typ `double`, weil ein Operant vom Typ `double` ist.

Das Spannende an den Ausdrücken sind also die Operatoren, die man in ihnen verwenden kann.

3.3 Operatoren [32]

Operatoren in Programmiersprachen werden verwendet, um bestimmte Operationen auf Variablen oder Werten auszuführen. Sie ermöglichen es, Berechnungen anzustellen, Vergleiche zu ziehen oder logische Ausdrücke zu evaluieren. Es gibt verschiedene Arten von Operatoren.

- *Zuweisungsoperatoren* (z. B. =, +=, -=)
  Weisen Variablen Wert zu oder verändern sie
- *Arithmetische Operatoren* (z. B. +, -, *, /)
  Führen mathematische Berechnungen durch
- *Vergleichsoperatoren* (z. B. ==, !=, <, >)
  Vergleichen zwei Werte miteinander und geben Wahrheitswert zurück
- *Logische Operatoren* (z. B. &&, ||)
  Verknüpfen logische Bedingungen miteinander

In C gibt es auch kombinierte Operatoren, wie +=, die sowohl arithmetische als auch Zuweisungsoperatoren sind.

3.4 Zuweisung [33]

Der einfachste Operator ist die **Zuweisung**, welche einer Variable einen Wert zuweist. Zuweisung

```
a = 5
```

- *Linke* Seite der Zuweisung
  → Variable (**lvalue**) lvalue
- *Rechte* Seite der Zuweisung
  → Ausdruck dessen Ergebnis zugewiesen wird (**rvalue**) rvalue
- unterschiedliche Typen rechts und links ⇒ Typkonvertierung
- Ergebnis der Zuweisung ist wieder ein Wert

```
a = b = c = d = e = 0;
a = (b = (c = (d = (e = 0))));
```

Die Unterscheidung in *lvalue* und *rvalue* ist bei C eine recht wichtige, weil der C-Standard sich häufig auf diese Begriffe bezieht. Außerdem neigt der Compiler dazu, Fehlermeldungen ebenfalls mit einem Hinweis auf diese zu garnieren.

Folgendes (fehlerhafte) C-Programm-Fragment

```
int k = 0;
k++ = 3;
```

führt beim gcc zu der Fehlermeldung

```
<source>:4:6: error: lvalue required as left operand of assignment
    4 |     k++ = 3;
      |     ~^~
```

Interessanterweise ist `++k = 3;` in C ebenfalls kein lvalue, in C++ aber korrekt.

Folgende Ausdrücke sind in C ein lvalue:

- Name einer Variable (`k`)
- Zuweisung, wenn rechts wieder ein lvalue steht (`k = j`)
- dereferenzierter Pointer (`*p`)
- Array-Elementzugriff (`a[3]`)
- Zugriff auf ein `struct`-Member (`s.m`)
- Zugriff auf ein `struct`-Member über einen Pointer (`s->m` oder `(*s).m`)

Die Regel ist, dass nur diese lvalues auf der linken Seite einer Zuweisung vorkommen dürfen. Auf der rechten Seite der Zuweisung dürfen rvalues *und* lvalues stehen.

Wenn Sie einen C++-Compiler verwenden, erweitert dieser die möglichen lvalues noch um:

- Pre-Increment oder Pre-Dekrement (`++k`, `--k`)
- Ternärer Ausdruck, wenn alle Rückgaben lvalues sind (`(x < y ? y : x)`)

Ein reiner C-Compiler würde diese beiden Formen aber abweisen. Deswegen sollte man sie vermeiden, auch weil sie ohnehin recht obskur und normalerweise nicht sinnvoll sind.

Dadurch, dass die Zuweisung selbst wieder einen Wert liefert, nämlich den Wert des rechten Ausdrucks, kann man sie in C verketten. Allerdings ist die hier gezeigte Verkettung nicht der Hauptanwendungsfall für diesen Effekt, sondern man nutzt ihn, wenn man Zuweisungen in Kontrollstrukturen verwendet.

```
int fd;
char buffer[1024];

if ((fd = open("file", O_RDONLY)) < 0) {
    /* Fehlerbehandlung */
}
```

```
read(fd, buffer, 1024);
```

Hier signalisiert die Funktion open einen Fehler mit einem Rückgabewert kleiner als 0 (Fehlerbehandlung in C wird später noch besprochen). Wenn kein Fehler auftritt, braucht man aber den Rückgabewert für die weiteren Operationen. Durch die Kombination von Zuweisung und Vergleich kann man dies elegant lösen.

3.5 Arten von Operatoren [34]

Wenn man etwas Licht in den Dschungel der Operatoren bringen will, kann man sie in einem ersten Schritt nach der Anzahl der möglichen Operanden aufteilen. Manche Operatoren wirken nur auf einen Operanden, die meisten auf zwei und ein einziger Operator kann drei Operanden haben.

Drei Arten von Operatoren

- **Unäre Operatoren** haben nur einen Operanden — Unäre Operatoren
  - Syntax: `op Operand`
  - Beispiel: Negation einer Zahl (`a = -b`)
- **Binäre Operatoren** verknüpfen zwei Operanden — Binäre Operatoren
  - Syntax: `Operand1 op Operand2`
  - Beispiel: Multiplikation zweier Zahlen (`a = b * c`)
- **Ternäre Operatoren** verknüpfen drei Operanden (selten) — Ternäre Operatoren
  - Syntax: `Operand1 op Operand2 op Operande3`
  - Beispiel: Bedingter Ausdruck (`x = a > b ? a : b`)

Man kann Operatoren danach klassifizieren, wie viele Operanden sie haben. Am häufigsten sind Operatoren mit zwei Operanden (*binäre Operatoren*). Deutlich seltener sind solche mit nur einem Operanden (*unäre Operatoren*). Mit drei Operanden (ternärer Operator) existiert in C nur ein einziger Operator, der sogenannte *Fragezeichen-Operator*, der später noch behandelt werden wird.

In C gibt es folgende unären Operatoren:

- **Negation** (-) — Negation
  `a = -b`
- **Increment**(++) — Increment
  `a++`
- **Decrement** (--) — Decrement
  `a--`
- **NOT** (!) — NOT
  `b = !a`
- **Complement** (~) — Complement

```
b = ~a
```

- **Adress-Operator** (&) — Adress-Operator
  ```
  p = &a
  ```
- **Indirektion** (*) — Indirektion
  ```
  *p = 7
  ```
- **sizeof** — sizeof
  ```
  x = sizeof(int)
  ```
- **Cast** ((TYPE)) — Cast
  ```
  a = (int)2.0
  ```

3.6 Rangfolge der Operatoren [35]

- **Rangfolge (priority)** legt fest, welche Operatoren zuerst ausgewertet werden (vgl. „Punkt vor Strichrechnung“ in der Mathematik) — Rangfolge
- **Assoziativität** legt fest, in welcher Richtung Operatoren mit gleicher Rangfolge ausgewertet werden — Assoziativität
  - **links-assoziativ**: Auswertung erfolgt von Links nach Rechts — links-assoziativ
    `3 + 5 + 6` ⇔ `(3 + 5)+ 6`
  - **rechts assoziativ**: Auswertung erfolgt von Rechts nach Links — rechts assoziativ
    `a = b = c` ⇔ `a = (b = c)`
- Durch *Klammern* kann die Rangfolge geändert werden
  `(2 + 2)* 2 = 8`

Da häufig mehrere Operatoren aufeinandertreffen, muss man festlegen, welche Operatoren zuerst ausgewertet werden. Diese Reihenfolge wird durch die *Rangfolge* der Operatoren eindeutig bestimmt, d. h. für jeden Operator ist festgelegt, vor welchen anderen Operatoren er ausgewertet wird.

Wenn ein Ausdruck mehrere Operatoren mit identischer Rangfolge enthält, wird die Richtung der Ausführung durch die *Assoziativität* bestimmt.

Zum Beispiel sind Multiplikations- und Modulo-Operator linksassoziativ. Daher sind die beiden folgenden Anweisungen austauschbar:

```c
int result = 5 * 70 % 6; /* ergibt 2 */
int result = (5 * 70) % 6; /* ergibt 2 */
```

Nicht aber

```c
int result = 5 * (70 % 6); /* ergibt 20 */
```

Rechtsassoziativ sind zum Beispiel die Vorzeichenoperatoren (+, –) oder die Zuweisung. Hier wird der Ausdruck von rechts nach links ausgewertet.

```c
int summe;
int result = summe = 12; /* summe ist 12, result ist 12 */
```

Im Zweifelsfall sollte man lieber Klammern setzen, als sich auf die Rangfolge zu verlassen: `2 + (2 * 2)`.

3.7 Bitweise Operatoren [36]

In C wird sehr viel mit maschinennahen Daten operiert, weswegen die *bitweisen Operatoren* eine besondere Bedeutung haben.

bitweisen Operatoren: arbeiten auf den Bits eines Wertes

bitweisen Operatoren

- *Eingabe*: zwei Zahlen &, ^, <<, >> und | bzw. eine Zahl bei ~
- *Ergebnis*: eine Zahl

Operator	Bedeutung	Beispiel	Beispiel	Ergebnis
&	Bitweise UND	x & y	13 & 1	1
|	Bitweise ODER	x | y	13 | 1	13
^	Bitweise XOR	x ^ y	13 ^ 11	6
~	Bitweise NOT	~ x	~13	-14
<<	Shift left	x << y	13 << 1	26
>>	Shift right	x >> y	13 >> 1	6

Die Operatoren &, ^ und |arbeiten auf den einzelnen Bits einer Zahl, d. h. man kann es sich so vorstellen, als ob jede einzelne Stelle einer Zahl in ihrer Binärdarstellung des einen Operanden mit dem korrespondierenden Bit des anderen Operanden verknüpft wird.

```
   1101 (13)         1101 (13)         1101 (13)
&  0001 ( 1)      |  0001 ( 1)      ^  1011 (11)
------------      ------------      ------------
 = 0001 ( 1)       = 1101 (13)       = 0110 ( 6)
```

Der Not-Operator (oder auch **Complement Operator**) bildet das **1er-Komplement** seines Operanden. Für das 2er-Komplement muss noch 1 addiert werden.

Complement Operator

1er-Komplement

16bit-Integer

```
       0000000000001101 (13)
~      1111111111110010 (-14)
```

Die Shift-Operatoren verschieben die Bits nach rechts bzw. links.

```
1101 >> 1 = 0110 (6)          1101 << 1 = 11010 (26)
```

Der Shift-Operator muss sich bei vorzeichenbehafteten Zahlen speziell verhalten, damit er weiterhin einer Multiplikation oder Division mit 2 entspricht.

Der C-Standard legt nicht fest, wie vorzeichenbehaftete Zahlen bei bitweisen Shift-Operatorn behandelt werden müssen. Die meisten C-Compiler behandeln vorzeichenbehaftete Zahlen allerdings speziell: Bei *vorzeichenbehafteten Zahlen* (z. B. `int`) wird bei einem Shift nach rechts (>>) oder links (<<) ein *arithmetischer Shift* verwendet. Das bedeutet, dass das Vorzeichenbit (das höchste Bit) während der Verschiebung beibehalten wird, um das Vorzeichen der Zahl zu bewahren: `-8 >> 1` ergibt -4, `-8 << 1` ergibt -16.

Handelt es sich um eine vorzeichenlose Zahl (z. B. `unsigned int`), wird ein *logischer Shift* durchgeführt und alle Bits werden verschoben: `8 >> 1` ergibt 4, `8 << 1` ergibt 16.

Man kann sich aber auf dieses Verhalten nicht verlassen.

3.8 Logische Operatoren [37]

Logische Operatoren verknüpfen Wahrheitswerte miteinander

Logische Operatoren

&&
||
!

- *Eingabe*: zwei Wahrheitswerte (`true` oder `false`) && und || bzw. ein Wahrheitswert bei !
- *Ergebnis*: eine Wahrheitswert (`true` oder `false`)

Operator	Bedeutung	Beispiel	Ergebnis
&&	und	`true && false`	`false`
||	oder	`true || false`	`true`
!	nicht	`!false`	`true`

Wie bereits erläutert, kennt C keine wirklichen logischen Datentypen, sondern bildet – selbst im neusten Standard – wahr und falsch auf `1` und `0` ab. Deswegen kann man anstelle der vordefinierten Werte `true` und `false` (aus `stdbool.h`) auch einfach mit den Zahlen `0` und `1` hantieren:

```c
printf("1 || 0 = %d\n", 1 || 0);
printf("1 && 0 = %d\n", 1 && 0);
printf("1 && 1 = %d\n", 1 && 1);
printf("!1 = %d\n", !1);
printf("!0 = %d\n", !0);
```

Ausgabe

```
1 || 0 = 1
1 && 0 = 0
1 && 1 = 1
```

```
!1 = 0
!0 = 1
```

Sie können sogar beliebige Zahlen einsetzen, d. h. auch Ausdrücke wie `33 && 99` schreiben. Allerdings geben moderne C-Compiler hier eine Warnung aus, dass man möglicherweise nicht weiß, was man tut.

Man kann die bitweisen Operatoren auch zur Auswertung von logischen Ausdrücken verwenden, weil bei der Verwendung von `0` und `1` für `false` und `true`, dasselbe Ergebnis herauskommt: `1 & 0` → `0`, `1 && 0` → `0` etc. Trotzdem sollte man das nicht tun, da die logischen Operatoren eine spezielle Eigenschaft haben: Sie sind als *Kurzschluss-Operatoren* implementiert.

Kurzschluss-Operatoren

`&&` und `||` sind sogenannte **Kurzschluss-Operatoren**

- `&&` und `||` brechen die Auswertung ab, sobald das Ergebnis feststeht
  - bei `&&` ist ein Ausdruck `false` ⇒ gesamter Ausdruck ist `false`
  - bei `||` ist ein Ausdruck `true` ⇒ gesamter Ausdruck ist `true`
- `&` und `|` werten den gesamten Ausdruck aus und berechnen dann erst das Ergebnis
- Normalerweise braucht man `&` und `|` nicht und sollte immer `&&` und `||` verwenden

Es gibt in C zwei Arten von Operatoren für UND und ODER, die bitweisen `|` und `&` und die sogenannten Kurzschlussoperatoren `||` und `&&`. Bei den bitweisen Operatoren werden alle (Teil-)Ausdrücke ausgewertet und erst dann wird der Operator angewandt. Bei den Kurzschlussoperatoren wird die Auswertung streng von links nach rechts durchgeführt und sie wird abgebrochen, sobald der logische Wert des Gesamtausdrucks feststeht.

Da ein Abbruch der Auswertung in nahezu allen Fällen im Interesse des Programmierers liegt, sollten *immer die Kurzschlussoperatoren* zum Einsatz kommen.

Das folgende Beispiel zeigt, warum die bitweisen Operatoren problematisch für logische Tests sind:

```
int *p = NULL;

if (p != NULL & *p > 7) {
     /* Absturz, weil p dereferenziert wird */
     printf("%s\n", "Wert ist größer als 7");
}
```

Das Programm stürzt ab, weil im Ausdruck `p != NULL & *p > 7` erst beide Operanden ausgewertet werden, bevor die bitweise Operation durchgeführt wird. Da `p` aber `NULL` ist, stürzt das Programm bei dem Versuch, den Pointer über `*p` zu dereferenzieren ab. Korrekt müsste die Bedingung `p != NULL && *p > 7` heißen. Durch den Kurzschlussoperator wird der zweite Teil des Ausdrucks nie ausgewertet, wenn `p` den Wert `NULL` hat.

Wie fügen sich die logischen Operatoren in die Rangfolge der anderen Operatoren ein? Alle stehen vor der Zuweisung und Negation und Vergleich haben einen höheren Rang als die UND- bzw. ODER-Verknüpfung.

Rang	Operator	Assoziativität
1.	!	R
2.	==	L
2.	!=	L
3.	&&	L
4.	||	L
5.	=	R

Diese Rangfolge hat den Vorteil, dass man in den gängigsten Fällen auf Klammern verzichten kann. Insbesondere kann man das Ergebnis einer logischen Verknüpfung einer Variable zuweisen, ohne klammern zu müssen: `int a = !b == c;`

3.9 Arithmetische Operatoren [40]

Arithmetische Operatoren für *ganze Zahlen*

Arithmetische Operatoren

- *Eingabe*: zwei ganze Zahlen (z. B. `char`, `short`, `unsigned int`, `long`)
- *Ergebnis*: eine ganze Zahl (z. B. `int`, `long`, `unsigned int`)
- Division durch 0 führt zu einem Laufzeitfehler (`5 / 0`)

Operator	Bedeutung	Beispiel	Ergebnis
+	Addition	`39 + 3`	42
-	Subtraktion	`26 - 3`	23
*	Multiplikation	`19 * 7`	133
/	Division	`19 / 7`	2
%	Modulo	`19 % 7`	5

Die arithmetischen Operatoren für ganze Zahlen sind alle bereits aus der Mathematik bekannt, werden nur teilweise durch andere Zeichen repräsentiert.

Der Modulo-Operator kommt in der Schulmathematik selten zum Einsatz, hat in der Informatik aber eine überragende Rolle, da viele Problemlösungen auf Module-Arithmetik beruhen, z. B. die *Hashtabellen*.

Integer Promotion

Ganzzahlige Typen, die kleiner als `int` sind (`char`, `short`), werden bei der Durchführung einer Operation erweitert (**Integer Promotion**). Wenn alle Werte des ursprünglichen Typs als `int` dargestellt werden können, wird der Wert des kleineren Typs in ein `int` konvertiert; andernfalls wird er in ein `unsigned int` konvertiert.

Integer Promotion

```
char c1, c2;
c1 = c1 + c2;
```

Hier werden `c1` und `c2` zu einem `int` erweitert, addiert und das Ergebnis wird wieder zu einem `char` verkürzt. Hierdurch werden Fehler durch einen Überlauf bei der Addition vermieden, sodass der folgende Ausdruck korrekt ausgewertet wird:

```
signed char cresult, c1, c2, c3;
c1 = 100;
c2 = 3;
c3 = 4;
cresult = c1 * c2 / c3; /* -> 75 */
```

Integer Conversion Rank

Für die Umwanlung zwischen den Integer-Typen, muss man das Konzept des **Integer Conversion Rank** heranziehen.

Integer Conversion Rank

Jeder ganzzahlige Typ hat einen sogenannten ganzzahligen Umwandlungsrang. Dieser basiert auf dem Konzept, dass jeder ganzzahlige Typ mindestens so viele Bits enthält wie die Typen, die darunter eingestuft sind. (C-Standard 2011, Unterabschnitt 6.3.1.1).

- Keine zwei unterschiedlichen *signed* Integer Typen dürfen denselben Rang haben, auch wenn sie dieselbe binäre Darstellung haben.
- Bei zwei *signed* Typen, bedeutet höhere Präzision (mehr Bit) einen höheren Rang.
- Es muss für den Rang der *signed* Typen gelten
  - `signed long long int` vor
  - `signed long int` vor
  - `signed int` vor
  - `signed short int` vor
  - `signed char`.
- Der Rang der *unsigned* Typen muss dem Rang des entsprechenden *signed* Typen entsprechen.
- Der Rang von `char` muss dem Rang von `signed char` und `unsigned char` entsprechen.
- Der Rang von `_Bool` muss kleiner sein als der Rang aller anderen Ganzzahltypen.

Arithmetische Umwandlung

Der ganzzahlige Umwandlungsrang wird bei den üblichen arithmetischen Umwandlungen verwendet, um zu bestimmen, welche Umwandlungen durchgeführt werden müssen, um eine Operation auf gemischte Ganzzahltypen zu unterstützen.

Nach der Erweiterung auf den größeren Typ werden folgende Regeln auf den arithmetischen Ausdruck angewendet (Notation $t_1 == typ(op_1)$:

1. $t_1 = t_2$
   $\Rightarrow$ keine Konvertierung.
2. $sign(t_1) = sign(t_2) \wedge rang(t_1) > rang(t_2)$
   $\Rightarrow t_2 \rightarrow t_1$
3. $t_1 = unsigned \wedge t_2 = signed \wedge rang(t_1) \geq rang(t_2)$
   $\Rightarrow t_2 \rightarrow t_1$
4. $|t_1| = |t_2| \wedge t_1 = unsigned \wedge t_2 = signed \wedge t_1 \subseteq t_2$
   $\Rightarrow t_1 \rightarrow t_2$
5. $t_1 = unsigned \wedge t_2 = signed \wedge t_1 \not\subseteq t_2$
   $\Rightarrow t_1 \rightarrow |t_2|_{unsigned}$ und $t_2 \rightarrow |t_2|_{unsigned}$

Das folgende Beispiel passiert, wenn man diese Konvertierungsregeln nicht beachtet:

Typumwandlung führt zu unerwartetem Ergebnis

```c
unsigned int a = 1;
signed int b = -1;
printf("%s\n", (b < a) ? "Smaller" : "Not smaller"); /* Not smaller */
```

Wie kann das Ergebnis passieren? Beide Datentypen sind gleich groß, unterscheiden sich aber im Vorzeichen. `signed int` kann nicht den gesamten Bereich von `unsigned int` erfassen, also greift Regel 4 und `b` wird in einen `unsigned int` konvertiert. `-1` wird dann zu `UINT_MAX` und das ist größer als `1`, sodass die Bedingung falsch ist.

Das Verhalten ändert sich, wenn man die Datentypen ändert:

Integer Promotion führt zu erwartetem Ergebnis

```c
unsigned short a = 1;
signed short b = -1;
printf("%s\n", (b < a) ? "Smaller" : "Not smaller"); /* Smaller */
```

Hier wird zuerst eine Integer-Promotion durchgeführt und der Datentyp `short` wird zu `int`. Da sowohl ein `unsigned short`, als auch ein `signed short` in einen `signed int` passen, werden sie entsprechend umgewandelt. Da die Typen dann gleich sind, greift Regel 1 und es wird keine weitere Typkonvertierung mehr durchgeführt. Damit ist das Ergebnis wie erwartet.

Arithmetische Operatoren für *Fließkommazahlen*

- *Eingabe*: zwei Zahlen (davon *mindestens* eine Fließkommazahl)
- *Ergebnis*: eine Fließkommazahl

Operator	Bedeutung	Beispiel	Ergebnis
+	Addition	39.1 + 3.3	42.4
-	Subtraktion	26.0 - 3	23.0
*	Multiplikation	19.3 * 7.8	150.54
/	Division	37.41 / 4.3	8.7
%	Modulo	19.0 % 7	5.0

3.10 Zusammengefasste Operatoren [43]

Zuweisung und Rechnung können zusammengefasst werden (häufig unübersichtlich)

Operator	Bedeutung
a += b	a = a + b
a -= b	a = a - b
a *= b	a = a * b
a /= b	a = a / b
a %= b	a = a % b

Man kann den Zuweisungsoperator mit den arithmetischen Operatoren kombinieren. Die so entstehenden Konstrukte sind manchmal unübersichtlich und sollten mit Vorsicht eingesetzt werden.

Da es sich um eine Zuweisung handelt, hat sie selbst wieder einen Wert, den man weiterverwenden kann.

```c
int a = 3;
int b;
b = (a *= 3);
printf("%d\n", b); /* -> 9 */
```

3.11 Vergleichsoperatoren [44]

Numerische Vergleichsoperatoren verknüpfen Zahlen miteinander

Numerische Vergleichsoperatoren

- Entsprechen den bekannten Operatoren aus er Mathematik
- *Eingabe*: zwei ganze Zahlen oder Fließkommazahlen
- *Ergebnis*: eine Wahrheitswert (true oder false)

Operator	Bedeutung	Beispiel	Ergebnis
<	kleiner	8 < 9	true
>	größer	9 > 4	true
<=	kleiner gleich	7 <= 3	false
>=	größer gleich	7 >= 3	true
==	gleich	6 == 5	false
!=	ungleich	6 != 5	true

Die Tabelle für die Fließkommazahlen ist identisch – abgesehen von den Beispielen.

Operator	Bedeutung	Beispiel	Ergebnis
<	kleiner	`8.0 < 9.0`	true
>	größer	`9.0 > 4.0`	true
<=	kleiner gleich	`7.0 <= 3.0`	false
>=	größer gleich	`7.0 >= 3.0`	true
==	gleich	`6.0 == 5.0`	false
!=	ungleich	`6.0 != 5.0`	true

Wie bereits oben beschrieben, muss man bei den Vergleichsoperatoren die Umwandlung der Integer-Werte berücksichtigen, um nicht auf ungewöhnliche Ergebnisse hereinzufallen.

Eine weitere Falle lauert bei den Fließkommazahlen, wenn man einen Vergleich durchführt.

Vergleich zweier Fließkommazahlen mit ==

- Fließkommazahlen sind mit *Ungenauigkeit* behaftet
- Vergleich mit == schlägt häufig wegen *Rundungsfehlern* fehl
- daher besser mit Epsilon-Umgebung vergleichen
  statt `a == b` besser `(a + epsilon > b)&& (a - epsilon < b)`

```c
double a = 3.0;
double b = 0.1;
double c = a * b; /* -> 0.30000000000000004 */

/* schlägt wegen Rundungsfehler fehl */
printf("%d\n", c == 0.3); /* false (0) */
/* besser */
printf("%d\n", (c - 0.00001 < 0.3) && (c + 0.00001 > 0.3)); /* true (1) */
```

3.12 Inkrement und Dekrement-Operator [46]

Ein berühmter Operator in C, der sogar namensgebend für C++ geworden ist, dient dem Erhöhen oder Erniedrigen einer Zahl um den Wert 1.

Inkrement-Operator und **Dekrement-Operator** → Nur auf *lvalue* anwendbar

Inkrement-Operator
Dekrement-Operator
Präfix-Operator

- **Präfix-Operator** (`++a` und `--a`) verändern den Wert der Variable und geben diesen zurück
- **Postfix-Operator** (`a++` und `a--`) geben den Wert der Variable zurück und verändern sie

Postfix-Operator

Operator	Bedeutung
a++	a = a + 1
++a	a = a + 1
a--	a = a - 1
--a	a = a - 1

Eine der häufigsten arithmetischen Operationen ist das Erhöhen oder Erniedrigen einer ganzen Zahl um den Wert Eins. Daher gibt es für diesen Fall zwei (genaugenommen vier) Operatoren, nämlich ++ und --, die auf eine Variable (nicht auf einen Wert) angewendet werden können.

Eine für Einsteiger schwer verständliche Eigenschaft der Operatoren ist, welchen Wert der Ausdruck selber zurückliefert. In jedem Fall wird die Variable erhöht oder erniedrigt. Von der Position des Operators (vor oder hinter der Variable) hängt aber ab, welchen Wert der Ausdruck zurückgibt.

- Steht der Operator *vor* der Variable, wird zuerst die Variable verändert und dann wird das Ergebnis zurückgeben.
- Steht der Operator *hinter* der Variable, wird erst der Wert der Variable zurückgegeben und dann der Wert erhöht.

Außerhalb einer Zuweisung oder einer Bedingung, ist es egal, welche Form man verwendet. Es hat sich aber bei den meisten Programmierern eingebürgert, dann die Postfix-Schreibweise (a++) zu verwenden.

```
int a = 7;
int b = a++;
printf("a=%d, b=%d\n", a, b); /* a=8, b=7 */
```

```
int a = 7;
int b = a--;
printf("a=%d, b=%d\n", a, b); /* a=6, b=7 */
```

```
int a = 7;
int b = ++a;
printf("a=%d, b=%d\n", a, b); /* a=8, b=8 */
```

```
int a = 7;
int b = --a;
printf("a=%d, b=%d\n", a, b); /* a=6, b=6 */
```

3.13 Komma-Operator [48]

Ein verwirrender Operator in C ist der **Komma-Operator**. Er erlaubt es an Stellen, die nur einen Ausdruck erlauben, mehrere Ausdrücke zu verwenden, die alle ausgewertet werden.

Komma-Operator

- Syntax: `Ausdruck1 , Ausdruck2`
  - `Ausdruck1` wird ausgewertet, das Ergebnis wird weggeworfen
  - `Ausdruck2` ausgewertet und dessen Ergebnis wird verwendet
- nur sinnvoll, wenn `Ausdruck1` einen Seiteneffekt hat

```
int x = 42;
int y = 23;

int z;
z = (x++, y++);
printf("x=%d, y=%d, z=%d\n", x, y, z); /* x=43, y=24, z=23 */
```

Man braucht den Komma-Operator nicht sehr häufig, er ist aber ausgesprochen nützlich, wenn man in einer `for`-Schleife mit mehreren Laufvariablen arbeiten möchte, wie das folgende Beispiel zeigt.

```
int x, y;

for (x = 0, y = 1; x < 10; x++, y *= 2) {
    printf("x=%d, y=%d\n", x, y);
}
```

Ausgabe

```
x=0, y=1
x=1, y=2
x=2, y=4
x=3, y=8
x=4, y=16
x=5, y=32
x=6, y=64
x=7, y=128
x=8, y=256
x=9, y=512
```

Index

C-Programmierung

Technische Hochschule Mannheim

Datentypen

Prof. Thomas Smits

Stand 2025-07-21

Technische Hochschule mannheim

Wintersemester 2025/2026

Inhaltsverzeichnis

Kapitel 1

C-Datentypen

1.1 Video zum Kapitel [5]

Link zu YouTube

1.2 Deklaration und Definition von Variablen [6]

Um in Programmen Daten verarbeiten zu können, verwaltet man sie in Variablen. Damit die Programmiersprache weiß, welche Art von Daten sich in einer Variable verstecken und insbesondere wie viel Speicher für diese Variable beschafft werden muss, haben Variablen in C einen **Datentyp**. Der Datentyp bleibt für die gesamte Lebensdauer der Variable gleich und kann sich in C nicht mehr ändern.

Datentyp

Variablen werden durch eine **Variablendeklaration** bekannt gemacht und der entsprechende Datentyp wird zugeordnet. Die **Variablendefinition** beschafft den notwendigen Speicher für die Ablage der Daten.

Variablendeklaration

Variablendefinition

Variablendeklaration und *Variablendefinition*

- ordnet einer Variablen einen Datentyp zu
- beschafft Speicher für die Variable
- Syntax: `TYP VARIABLENNAME;`
- Syntax: `TYP VARIABLENNAME, VARIABLENNAME, ...;`

```c
/* Deklaration und Definition */
int a; /* einzelne Variable vom Typ int */
int b, c; /* mehrere Variablen vom Typ int */
```

```
a = 3; /* Zuweisung */
a = 4; /* erneute Zuweisung */
a = 3.4; /* Fehler: Falscher Datentyp */
```

Anders als in Skriptsprachen, wie z. B. Python, können einer Variablen keine Daten eines anderen Typs zugewiesen werden. Der Datentyp der Variablen kann sich später auch nicht ändern, sie hat immer denselben Typ. Dies erlaubt dem Compiler den Speicher einmal anzufordern und dann sicher zu sein, dass jede Zuweisung an die Variable in den Speicher passen wird.

Wenn man eine Variable nur deklarieren will, die Definition aber an anderer Stelle erfolgt, dann muss man das Schlüsselwort `extern` verwenden.

extern

Variablendeklaration ohne *Variablendefinition*

- macht eine Variable und deren Datentyp bekannt
- Definition erfolgt an anderer Stelle
- Syntax: `extern TYP VARIABLENNAME;`
- Syntax: `extern TYP VARIABLENNAME, VARIABLENNAME, ...;`

```
/* Deklaration ohne Definition */
extern int a; /* einzelne Variable vom Typ int */
extern int b, c; /* mehrere Variablen vom Typ int */
```

Der Compiler kann nicht überprüfen, ob die Variable wirklich definiert wurde, da er immer nur eine einzige Quelldatei betrachtet. Der Linker wird aber nach dem Symbol der Variable in den Objektdateien suchen, die er zusammen bindet. Wenn die Variable nicht in einer definiert (und exportiert wurde), gibt er eine Fehlermeldung aus.

Linker Fehlermeldung bei fehlender Definition

```
/usr/bin/ld: /tmp/cc3N5eqp.o: in function `main':
extern.c:(.text+0x6): undefined reference to `b'
/usr/bin/ld: extern.c:(.text+0xc): undefined reference to `a'
collect2: error: ld returned 1 exit status
```

1.3 Standardtypen [8]

Jede Programmiersprache braucht eine gewisse Anzahl von Datentypen, die direkt in der Sprache verstanden werden. Diese können dann vom Entwickler genutzt werden, um weitere (komplexe) Datentypen zu konstruieren.

C hat eine Reihe von **Standarddatenytpen**, die ähnlich zu denen von Java sind

Standarddatenytpen

- `char`

char

- `int` — int
- `short` — short
- `long` — long
- `long long` — long long
- `float` — float
- `double` — double
- `long double` — long double

Es fehlt ganz offensichtlich der Datentyp `boolean` und auch die Bitbreite der Datentypen ist nicht identisch zu Java (siehe unten).

Besonders beachtenswert sind die zusammengesetzten Bezeichnungen: Ein `long` ist etwas anderes als ein `long long` und `long long` bezeichnet – obwohl es *zwei* Worte sind – *einen* Datentyp.

C wurde mit dem Ziel entwickelt, dass sich in C geschriebene Programme auf unterschiedlichen Plattformen (Prozessoren, Betriebssystemen) kompilieren und ausführen lassen. Aufgrund der Maschinennähe von C sollten die Datentypen besonders gut zu dem verwendeten Prozessor auf der Plattform passen, sodass man sich dazu entschieden hat, die Bitbreiten der Datentypen *nicht* zu spezifizieren. Stattdessen fordert der C-Standard einige Beziehungen zwischen den Datentypen.

- *Breite* (in Bits) der Standardtypen ist *nicht* festgelegt und hängt von der Plattform ab
- *kleinste Datentyp* ist immer `char`
- es gibt nur *Relationen* zwischen den Typen und Mindestgrößen
  - `long long` (mind. 64 Bit) >= `long`
  - `long` (mind. 32 Bit) >= `int`
  - `int` (mind. 16 Bit) >= `short`
  - `short` (mind. 16 Bit) >= `char`
  - `char` (mind. 8 Bit)

Auf den ersten Blick könnte man denken: „OK, welche Plattform unterstützt denn nicht wenigstens 64 Bit nativ?“ Die Antwort ist, dass C auch heute noch auf Plattformen eingesetzt wird, die eine Standardbitbreite 16 Bit haben, z. B. diverse Mikrocontroller. In Steuerungssystemen sind bis heute sogar 8-Bit-Prozessoren im Einsatz.

Laut C-Standard, sollte der Datentyp `int` derjenige sein, der von der Plattform am besten unterstützt wird – der **native Datentyp**. Im Allgemeinen heißt das, dass es der Datentyp sein sollte, welcher der Breite der Prozessorregister entspricht. native Datentyp

Es gibt noch eine Reihe von (historisch entstandenen) alternativen Namen für die Datentypen:

Type	Aliases

`short`	`short int`
	`signed short`
	`signed short int`

`unsigned short`	`unsigned short` `unsigned short int`
`int`	`signed` `signed int`
`unsigned int`	`unsigned`

`long`	`long int`
	`signed long`
	`signed long int`

`unsigned long`	`unsigned long int`

`long long`	`long long int`
	`signed long long`
	`signed long long int`

`unsigned long long`	`unsigned long long int`

Im Folgenden ein paar Beispiele für die Bitbreiten der C-Datentypen auf unterschiedlichen Plattformen.

Datentyp	Arduino	macOS	Win32	Win64
`char`	8 Bit	8 Bit	8 Bit	8 Bit
`short`	16 Bit	16 Bit	16 Bit	16 Bit
`int`	16 Bit	32 Bit	32 Bit	32 Bit
`long`	32 Bit	64 Bit	32 Bit	32 Bit
`long long`	—	64 Bit	64 Bit	64 Bit
`float`	32 Bit	32 Bit	32 Bit	32 Bit
`double`	32 Bit	64 Bit	64 Bit	64 Bit

Wie man sieht, definieren nur Arduino und Win32 den Datentyp `int` entsprechend der Breite der Prozessorregister. Die anderen Plattformen haben sich entschieden, `int` bei 32 Bit zu belassen, damit die Kompatibilität mit älteren Programmen gewahrt bleibt.

Ob man mit unterschiedlichen Breiten der Datentypen leben kann, hängt vom Programm ab. Geht es z. B. darum, Netzwerkpakete zu verarbeiten, dann benötigt man Datentypen mit bekannter Bitbreite, um die Daten darin abzulegen. Will man nur ein Zahlenratespiel implementieren oder über ein Array mit einigen Elementen laufen, ist es egal, ob ein `int` 16 oder 32 Bit hat.

Datentypen mit fester Breite

Das Problem wird gelöst durch die Definition von **Datentypen mit fester Breite**. Jetzt hat man die Wahl.

In `stdint.h` werden Typen mit *fester Breite* deklariert, die plattformübergreifend gelten

- `int8_t` – 8 Bit signed integer
- `int16_t` – 16 Bit signed integer
- `int32_t` – 32 Bit signed integer
- `int64_t` – 64 Bit signed integer
- `uint8_t` – 8 Bit unsigned integer
- `uint16_t` – 16 Bit unsigned integer
- `uint32_t` – 32 Bit unsigned integer
- `uint64_t` – 64 Bit unsigned integer

`int8_t` `int16_t` `int32_t` `int64_t` `uint8_t` `uint16_t` `uint32_t` `uint64_t`

In modernen C-Programmen sollte man die Datentypen aus `stdint.h` verwenden und sich so die Kopfschmerzen bezüglich der nicht genormten Bitbreiten ersparen.

1.4 Wertebereich der Zahlentypen [12]

Anders als in Java, werden die Ganzzahl-Datentypen in C noch einmal darin unterschieden, ob sie mit oder ohne Vorzeichen benutzt werden. Hierfür gibt es weitere Schlüsselworte (`signed` und `unsigned`), die anzeigen, ob ein Datentyp mit oder ohne Vorzeichen gewünscht wird. Daraus ergibt sich, dass die Datentypen unterschiedliche **Wertebereiche** haben, abhängig davon, ob sie `signed` oder `unsigned` sind.

Wertebereiche

- C-Datentypen gibt es mit und ohne Vorzeichen
  - `unsigned` → *Ohne Vorzeichen*
    z. B. `unsigned int`
  - `signed` → *Mit Vorzeichen*
    z. B. `signed int`
  - Ohne Angabe → `signed`
    z. B. `int` ist `signed int`
- *Wertebereich* ändert sich durch Vorzeichen

Vorzeichen	Breite	Min	Max
`signed`	8 Bit	-2^7	$2^7 - 1$
`signed`	16 Bit	-2^{15}	$2^{15} - 1$
`signed`	32 Bit	-2^{31}	$2^{31} - 1$
`signed`	64 Bit	-2^{63}	$2^{63} - 1$
`unsigned`	8 Bit	0	$2^8 - 1$
`unsigned`	16 Bit	0	$2^{16} - 1$
`unsigned`	32 Bit	0	$2^{32} - 1$
`unsigned`	64 Bit	0	$2^{64} - 1$

Datentyp	Breite	Wertebereich
Fließkomma	32 Bit	+/- $3.402,823,4 * 10^{38}$
Fließkomma	64 Bit	+/- $1.797,693,134,862,315,7 * 10^{308}$

1.5 Literale [14]

Literale erlauben die Angabe von Werten direkt im Quelltext. Hier gibt es wegen der Gemeinsamkeiten von C und Java keine Überraschungen.

- Die Schreibweise von Integer- und Float-**Literalen** in C entspricht der in Java

  Literalen

  - `xxxx` – Integer-Literal in Dezimalschreibweise, z. B. `1299`
  - `0xHHHH` – Integer-Literal in hexadezimaler Schreibweise, z. B. `0xCAFEBABE`
  - `0xxxx` – Integer-Literal in oktaler Schreibweise, z. B. `0777`
  - `xxx.xxx` – Double-Literal
- Durch einen Suffix kann der Datentyp angezeigt werden
  - `L` – `long`, z. B. `182721L` — L
  - `LL` – `long long`, z. B. `18272222LL` — LL
  - `u` – `unsigned`, z. B. `0x8a632ffeuLL` — u

1.6 Typumwandlung [15]

C ist eine typsichere Sprache (wie auch Java), d. h. der Compiler überprüft die Typen von Variablen und Literalen bei Zuweisungen und stellt sicher, dass bei der Zuweisung keine inkompatiblen Typen zum Einsatz kommen.

Anders als der Java-Compiler meckert der C-Compiler aber nicht, wenn man einen größeren an einen kleineren Typ (z. B. `long` und `short`) zuweist (**narrowing**), sondern führt die Zuweisung einfach durch. Wenn der zugewiesene Wert außerhalb des Wertebereichs der Variable liegt, kommt es zu einem Überlauf und Daten gehen verloren. Variablen, deren Typ vorzeichenbehaftet ist, springen dann auch gerne in den negativen Bereich. Hier lauert aber dann eine der größten Problemfelder von C, das sogenannte *undefined behavior*, das weiter unten noch genauer erläutert wird.

narrowing

- **Implizite Typumwandung**, d. h. ohne *Cast*

  Implizite Typumwandung
  - `char` ↔ `short` ↔ `int` ↔ `long`
  - wenn ein Operand `double` ist, wird der andere zu `double` konvertiert
  - wenn ein Operand `float` ist, wird der anderer zu `float`
- **Explizite Typumwandung** mit *Cast*

  Explizite Typumwandung
  - Syntax: `(typ) wert`
  - Analog zu Java
- C-Umwandlungen machen nicht immer das, was man erwartet

Will man eine Typ explizit umwandeln, kann man dazu einen **Cast** verwenden, der mithilfe des **Cast-Operators** `()` eine Umwandlung durchführt.

Cast

Cast-Operators

```c
unsigned char c;
short s;
int i;
long l;
double d;

i = 100000;
s = i; /* narrowing, impliziter Cast */
printf("%d %d\n", i, s); /* 100000 -31072 */

s = (short) i; /* narrowing, expliziter Cast */
printf("%d %d\n", i, s); /* 100000 -31072 */

l = i; /* widening, impliziter Cast */
printf("%d %ld\n", i, l); /* 100000 100000 */
```

Das Beispiel zeigt, dass C einfach den zu großen `int`-Wert `i` in die `short`-Variable `s` presst. Da `s` vorzeichenbehaftet ist, springt der Wert ins Negative um. Warum? Bei der Zuweisung werden einfach die unteren 16 Bit von `i` (`1100001101010000`) in `s` geschrieben `1000011010100000`. Das oberste Bit ist gesetzt, also handelt es sich um eine negative Zahl. Da diese in Zweierkomplementdarstellung vorliegt, müssen wir 1 abziehen und die Bits flippen: `0111100101100000`, um den Wert zu erhalten. Diese ist 31072, unter Beachtung des Vorzeichens also tatsächlich `-31072`.

Die Typumwandlung kann man auch durch einen Cast `s = (short)i` herbeiführen. Am Ergebnis ändert sich allerdings nichts.

```c
c = i; /* narrowing, impliziter Cast */
printf("%d %d\n", i, c); /* 100000 160 */

d = 3 / 2; /* Ausdruck ist int */
printf("%f\n", d); /* 1.000000 */

d = 3 / 2.0; /* Ausdruck ist double */
printf("%f\n", d); /* 1.500000 */

d = (double) i;
printf("%f\n", d); /* 100000.000000 */
```

In der letzten Zeile des Beispiels sieht man eine explizite Umwandlung eines `int` in einen `double`-Wert. Hier ist der Compiler immerhin so nett und schreibt nicht das Bitmuster in die Variable, sondern konvertiert den Wert in das korrekte Format für Fließkommazahlen.

1.7 Boolean [18]

In vielen Programmiersprachen gibt es einen expliziten Datentyp für Wahrheitswerte, der in Java z. B. als `boolean` bezeichnet wird. C verzichtet auf einen solchen und bildet dies einfach auf `int` ab.

boolean

- C hat bis C23 *keinen* Datentyp für **boolean**
- wird über `int` oder `char` simuliert
  - `== 0` → false
  - `!= 0` → true

```c
int i = 7;
int bigger = i > 8;
int smaller = i < 8;

printf("%d\n", bigger); /* -> 0 */
printf("%d\n", smaller); /* -> 1 */
```

Details zu der Frage, wie man in C mit Wahrheitswerten umgeht, wurden bereits im Abschnitt zu den Kontrollstrukturen diskutiert. Die Abbildung auf `int` führt dazu, dass man in C einige Abkürzungen nehmen kann, die in anderen Sprachen nicht möglich sind.

bool

Mit dem C23-Standard wurde der Datentyp (und das Schlüsselwort) `bool` mit den Konstanten `true` und `false` eingeführt. Es gilt, dass `true` dem Wert `1` und `false` dem Wert `0` entspricht.

1.8 Eigene Typen [19]

typedef

In C wird `typedef` verwendet, um Aliasnamen für bestehende Datentypen zu erstellen. Dies kann die Lesbarkeit des Codes verbessern, besonders bei komplexen Datentypen wie Strukturen, Unions oder Pointern. Mit `typedef` kann man kürzere oder aussagekräftigere Namen für Datentypen definieren, was den Code verständlicher und wartbarer macht.

- Mit `typedef TYP NAME` kann man eigene Typen definieren
- besonders wichtig für `struct` und `enum` (siehe unten)

```c
typedef short int smallNumber;
typedef unsigned char byte;

smallNumber x;
byte b;
```

Hierbei bezeichnet `TYP` einen vorhandenen Datentyp und `NAME` gibt den Namen für den neu zu definierenden Typ an.

Auf den ersten Blick wirkt es etwas unnötig, dass man für einen vorhandenen Datentyp wie `short int` mit einem neuen Namen `smallNumber` zu versehen. Es gibt aber drei wichtige Einsatzzwecke für `typedef`:

- Datentypen plattformunabhängig definieren
- Enumerationen deklarieren
- Strukturen deklarieren

1.9 Enumerationen [20]

Häufig benötigt man einen Aufzählungsdatentyp, mit dem man eine Auswahl aus einer Reihe von Elementen darstellen kann. Hierzu bietet C *Enumerationen* an.

- Aufzählungstypen (**Enumerationen**) können mit `enum` definiert werden
- Verhalten sich wie `int`

Enumerationen

```
typedef enum { Red, Green, Blue } Color;
typedef enum { Rain=1, Snow=2, Wind=4 } Weather;

Color c = Green;
Weather w = Snow;

printf("%d\n", c); /* -> 1 */
printf("%d\n", w); /* -> 2 */
printf("%d\n", w + c + Wind); /* -> 7 */

w = 9; /* Außerhalb des Wertebereiches */
```

Aus Effizienzgründen werden Enumerationen in C intern als Ganzzahl dargestellt. Gibt man bei der Deklaration einer Enumeration keine Zahlenwerte an, werden die Elemente einfach bei 0 beginnend nummeriert. Wie das Beispiel zeigt, kann man aber auch explizit die Werte für die Elemente vorgeben.

C macht allerdings – anders als Java – wenig Anstalten, den wahren Charakter von Enumerationen als Zahlenwerte zu verbergen. Man kann einfach mit ihnen rechnen und auch Werte zuweisen, die außerhalb des Wertebereiches liegen. C-Enumerationen sind daher *nicht* typsicher, anders als in Java. Dafür sind sie sehr viel effizienter implementiert und haben wenig Overhead.

Beginnend mit dem C23-Standard kann man dem Compiler auch mitteilen, welchen Ganzzahlentyp er für die Enumeration verwenden soll.

```
typedef enum : unsigned long long {
    a0 = 0x00FFFFFFFFFFFFFFULL,
    a1 = 0xFFFFFFFFFFFFFFFFULL
} BigEnum;
```

Andernfalls kann der Compiler selbst einen Datentyp wählen und verschiedene Compiler treffen hier durchaus unterschiedlichen Entscheidungen.

1.10 Objekte [21]

C ist eine prozedurale Programmiersprache und kennt deswegen keine Objekte. Ein C-Programm besteht aus Funktionen und globalen Daten.

- C hat keine Objekte
- Variablen in einem Block { } sind nur in dem Block gültig (→ Java)
- Variablen außerhalb eines Blocks
  - sind global
  - leben so lange, wie das Programm lebt
  - werden mit `static` auf eine Datei beschränkt
- eine lokalen Variable ist nach der Definition *nicht initialisiert* (↔ Java)

Sichtbarkeit von Variablen

Man kann die **Sichtbarkeit von Variablen** in C nur auf vier Ebenen kontrollieren:

- *global* → jede Variable, die nicht als `static` gekennzeichnet ist, außerhalb einer Funktion
- *global innerhalb einer Datei* → jede Variable, die als `static` gekennzeichnet ist, außerhalb einer Funktion
- in einer *Funktion* → lokale Variablen in einer Funktion
- in einem *Block* → lokale Variablen in einem Block, in einer Funktion

```c
/* Sichtbar im gesamten Programm */
int global;

/* Sichtbar nur in dieser Datei */
static int file_global;

void f(void) {
    /* Sichtbar in der ganzen Funktion */
    int function_local;

    {
      /* Sichtbar nur im Block */
      int block_local;
    }
}
```

static

Das Schlüsselwort `static` hat in C also eine ganz andere Bedeutung als in Java. Während es in Java die Variable unabhängig vom Objekt macht – also auf gewisse Weise globaler – schränkt es in C die Sichtbarkeit auf die aktuelle Quelltext-Datei ein.

In C besitzen nur globale Variablen durch die Definition einen bekannten und vordefinierten Wert, nämlich `0`. Für lokale Variablen gilt dies nicht, d. h. sie müssen initialisiert werden oder sie tragen einen **undefinierten Wert**, also einen beliebigen, Wert.

undefinierten Wert

```c
#include <stdio.h>

int global; /* wird mit 0 initialisiert */

void f(void) {
    int local; /* Zuweisung fehlt hier */
    printf("global=%d, local=%d", global, local);
}

int main(int argc, char** argv) {
    f();
    return 0;
}
```

Ausgabe

```
global=0, local=21849
```

Der Wert ist nicht wirklich beliebig, sondern repräsentiert die Daten, die aus vorangegangenen Funktionsaufrufen auf dem Stack an der entsprechenden Stelle gespeichert sind. Mit der Option `-Wall` würde der C-Compiler vor diesem Problem warnen:

```
initialization.c:7:3: warning: 'local' is used uninitialized in this
                              function [-Wuninitialized]
    7 |   printf("global=%d, local=%d", global, local);
      |   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Welche Größe haben Datenobjekte in C?

- Alle Datenobjekte in C haben eine *feste Größe* während ihrer Lebenszeit (Ausnahme: dynamisch allozierter Speicher)
- Größe wird bei der Erzeugung festgelegt
  - *globale* Variablen beim Kompilieren (Daten-Segment)
  - *lokale* Variablen beim Funktionsaufruf (Stack)
  - *dynamische* Daten durch den Programmierer (Heap)
- `sizeof`-Operator liefert die Größe eines Objektes in Bytes

  sizeof
  - wird zur Compilezeit bestimmt, keine Laufzeitinformation
  - `sizeof(var)`: Größe der Variable (in Bytes)
  - `sizeof(typ)`: Größe des Typs (in Bytes)

```c
int a;
long int b;
double c;

/* sizeof von Variablen */
printf("sizeof(a): %lu\n", sizeof(a)); /* -> sizeof(a): 4 */
printf("sizeof(b): %lu\n", sizeof(b)); /* -> sizeof(b): 8 */
printf("sizeof(c): %lu\n", sizeof(c)); /* -> sizeof(c): 8 */

/* sizeof eines Datentyps */
printf("sizeof(float): %lu\n", sizeof(float)); /* -> sizeof(float): 4 */
```

Es ist wichtig, im Kopf zu behalten, dass `sizeof` vom Compiler ausgewertet wird und dann der konstante Wert mit der Größe des Datenobjektes in das Programm eingesetzt wird. Deswegen ist es auch egal, ob man eine Variable oder einen Datentyp angibt. Der Compiler ersetzt in der ersten Variante die Variable durch den Datentyp und bestimmt dann dessen Größe.

Zur Laufzeit muss man in einem C-Programm selbst Buch über die Größe von dynamisch erzeugten oder an Funktionen übergebenen Datenobjekten führen. Dies wird beim Thema Arrays noch etwas genauer diskutiert, soll hier aber kurz vorweggenommen werden.

```c
#include <stdio.h>

void f(char data[]) {
    printf("sizeof(data): %lu\n", sizeof(data));
}

int main() {
    char data[100];
    f(data); /* -> sizeof(data): 8 */
}
```

Bei der Option `-Wall` warnt der Compiler auch vor diesem Fehler:

```
sizeof.c: In function 'f':
sizeof.c:4:41: warning: 'sizeof' on array function parameter 'data' ↩
    will return size of 'char *' [-Wsizeof-array-argument]
     4 |     printf("sizeof(data): %lu\n", sizeof(data));
       |                                         ^
sizeof.c:3:13: note: declared here
     3 | void f(char data[]) {
       |        ~~~~~^~~~~~
```

Kapitel 2

Pointer

2.1 Video zum Kapitel [25]

Link zu YouTube

2.2 Speicherlayout eines Prozesses [26]

Prozesse

Programme werden vom Betriebssystem als **Prozesse** ausgeführt. Prozesse sind aus Sicherheitsgründen vollständig voneinander isoliert und haben eigene Ressourcen. Vor allem ist der Speicher von Prozessen vollständig getrennt, d. h. ein Prozess kann nicht auf den Speicher eines anderen Prozesses zugreifen. Prozesse können daher nur über den Umweg des Betriebssystems miteinander in Kontakt treten. Die Technik hierzu nennt man **Interprozess-Kommunikation (interprocess communication) IPC**. Beispiele für IPC sind Sockets, Pipes und Shared Memory. Wegen der vollständigen Isolation sind Prozesse relativ schwergewichtig, d. h. das Starten und die Verwaltung von Prozessen benötigt vergleichsweise viele Ressourcen.

Interprozess-Kommunikation
IPC

Threads

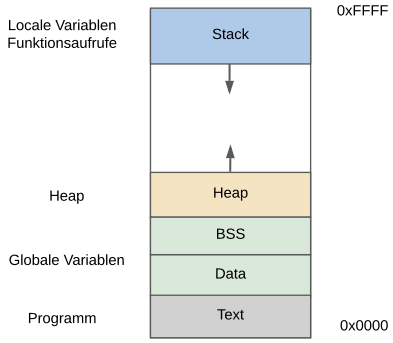
Innerhalb eines Prozesses gibt es einen oder mehrere **Threads**, die man sich als Ausführungspfade vorstellen kann. Im Gegensatz zu Prozessen sind Threads leichtgewichtiger, d. h. sie lassen sich mit deutlich weniger Aufwand starten und verwalten. Da Threads innerhalb eines Prozesses laufen, haben sie keinen getrennten Heap, sondern teilen sich den Heap und kommunizieren über diesen miteinander. Der Stack ist aber bei Threads getrennt: Jeder Thread hat seinen eigenen Stack.

Der Speicher eines Prozesses besteht aus sechs Teilen.

Prozess-Speicher besteht aus

- *Text-Segment*: Programmcode
- *Daten-Segment*: Vorinitialisierte Daten

- *BSS-Segment*: Nicht initialisierte Daten
- *Heap-Segment*: Dynamisch allozierter Speicher
- *Stack-Segment*: Lokale Variablen
- *Programm-Counter*: Stelle im Text-Segment, die aktuell ausgeführt wird



Die Aufgaben der Bereiche sind:

- **Text-Segment** (feste Größe): Enthält den Programmcode und kann zur Laufzeit nicht beschrieben werden. Dafür ist es als Executable gekennzeichnet, d. h. der Program-Counter darf sich bei Adressen befinden, die im Text-Segment liegen. **Text-Segment**
- **Daten-Segment** (feste Größe): Enthält die globalen, initialisierten Variablen des Programms. Alle globalen Variablen, die einen Wert ungleich `0` haben, liegen im Daten-Segment. **Daten-Segment**
- **BSS-Segment** (feste Größe): Enthält den Speicher für die globalen Variablen, die nicht initialisiert wurden. **BSS-Segment**
- **Heap-Segment** (dynamische Größe): Dynamisch allozierter Speicher, der von der Programmiererin manuell verwaltet wird (siehe unten). **Heap-Segment**
- **Stack-Segment** (dynamische Größe): Dynamisch allozierter Speicher, der für die lokalen Variablen verwendet wird. Das Belegen und Freigeben des Speichers wird automatisch vom Compiler durchgeführt, der Entwickler muss sich nicht darum kümmern. Deswegen werden lokale Variablen in C auch als **automatische Variablen** bezeichnet. **Stack-Segment** **automatische Variablen**
- **Programm-Counter**: Prozessor-Register, das den aktuell auszuführenden Befehl speichert. **Programm-Counter**

Der Trennung in Daten- und BSS-Segment liegt eine Optimierung zugrunde: Damit die ausführbare Datei (Executable) nicht zu groß wird, enthält sie nur das Text- und Daten-Segment sowie eine Information, wie groß das BSS-Segment sein muss. Das BSS-Segment wird dann vom Betriebssystem beim Laden des Programms anhand dieser Information alloziert, das Executable bläht sich somit nicht unnötig auf.

Aus der Zeit des begrenzten Adressraums (32-Bit-Systeme können nur 4 GB an Speicher adressieren), hat sich die Konvention entwickelt, dass der Heap nach oben und der Stack nach unten wächst. So konnte man den Speicher immer maximal ausnutzen. Beim Erzeugen neuer

Variablen auf dem Stack, bekommen diese *kleinere* Adressen, als die vorhergehenden, beim Heap hingegen, wachsen die Adressen an. Obwohl man dies heute bei 64-Bit-Adressen nicht mehr bräuchte, hat sich die Konvention auch auf den 64-Bit-Plattformen gehalten.

2.3 Pointer-Typen [28]

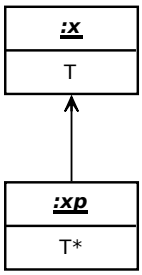
Pointer

Ein zentraler Datentyp in C, den man aus Java nicht kennt, ist der **Pointer**. Ohne Pointer kann man keine komplexeren C-Programme schreiben, weil Pointer die einzige Möglichkeit sind, an Funktionen übergebene Daten zu modifizieren oder dynamisch allozierten Speicher zu verwalten.

Jetzt gibt es allerdings nicht einen einzigen Typ, der Pointer repräsentiert, sondern es gibt genau so viele Pointer-Typen, wie es Datentypen in C gibt und noch einen mehr (`void*`), aber dazu später.

Pointer-Typ
Zeigertyp

- Jeder Datentyp `T` hat einen passenden **Pointer-Typ** bzw. **Zeigertyp** `T*`
- der Wert von `T*` ist die Adresse eines Objektes mit dem Typ `T`
- sei `xp` eine Variable vom Typ `T*` und `x` eine Variable vom Typ `T`
  - `xp = &x` weist `xp` die Adresse von `x` zu
  - `*xp` dereferenziert den Pointer und bezieht sich auf `x`
  - `*xp` hat den Typ `T`



Pointervariable

Ein Pointer ist – wie der Namen schon sagt – ein Zeiger auf ein Datenobjekt. Pointer sind genauso Variablen, wie andere auch. Der Inhalt der **Pointervariable** ist die Speicher-Adresse des Datenobjektes, auf das der Pointer zeigt.

&
Adress-Operator
*
Indirektions-Operator

Die Adresse eines Datenobjektes erhält man mit dem &-Operator, **Adress-Operator**. Auf das Datenobjekt hinter dem Pointer greift man mit dem *-Operator zu (**Indirektions-Operator**). Man spricht hier auch von *Dereferenzieren*.

Das Beispiel zeigt, wie ein Pointer `xp` auf das Datenobjekt `x` benutzt wird, um die Daten in `x` zu ändern. Die Adresse des Datenobjektes ändert sich nicht, wohl aber sein Inhalt.


```c
int x = 7;
int* xp = &x;

printf("xp=%p\n", xp); /* xp=0x7fff5302f7ac */
printf("*xp=%d\n", *xp); /* *xp=7 */
printf("x=%d\n", x);   /* x=7 */

*xp = 42;
printf("xp=%p\n", xp); /* xp=0x7fff5302f7ac */
printf("*xp=%d\n", *xp); /* *xp=42 */
printf("x=%d\n", x);   /* x=42 */
```

Der Format-Ausdruck `%p` im `printf` erlaubt es, die Adresse eines Pointers auszugeben. Der besseren Lesbarkeit wegen verwendet `%p` eine hexadezimale Darstellung.

Wichtig ist, dass die Typsicherheit von C auch für die Pointer gilt. Das heißt, dass ein Pointer vom Typ `T*` nur auf Objekte vom Typ `T` zeigen kann, der Typ des Objektes „färbt“ also auf den Typ des Pointers ab.

Im Prinzip sind C-Pointer nicht groß unterschiedlich zu Objekt-Referenzen in Java. Auch sie zeigen auf ein Objekt und tragen dessen Adresse in sich. Der Unterschied ist allerdings, dass Java-Referenzen die Information zur Speicher-Adresse des Objektes nicht preisgeben und automatisch dereferenziert werden, somit kein `*`-Operator benötigt wird.


```java
class MyInt {
    // Wir müssen den int in eine Klasse verpacken, weil int
    // ein primitiver Datentyp ist und keine Referenzen erlaubt.
    // Integer scheidet auch aus, weil es immutable ist.
    public int x;

    public MyInt(int x) {
        this.x = x;
    }
}

class Main {
    public static void main(String[] args) {
        MyInt xp = new MyInt(7);
        MyInt xp2 = xp;

        System.out.println("x=" + xp.x); // 7
        System.out.println("x=" + xp2.x); // 7

        xp.x = 42;
        System.out.println("x=" + xp.x); // 42
        System.out.println("x=" + xp2.x); // 42
```

```
    }
}
```

2.4 Pointer-Arithmetik [30]

Man kann sich C-Pointer wie Java-Referenzen vorstellen. Allerdings erlaubt C, dass man mit Pointern direkt rechnen kann. Hierbei wird einfach die Adresse, die in dem Pointer gespeichert ist, für die Rechenoperation verwendet. Der Compiler muss also für die Pointer-Arithmetik kaum Magie betreiben und fasst den Pointer einfach als Zahlenwert auf.

Die einzige Besonderheit ist, dass der Compiler bei Addition und Subtraktion die Größe des Datentyps berücksichtigt, auf den der Pointer zeigt.

Unter **Pointer-Arithmetik** versteht man, dass man mit Pointern rechnen kann

Pointer-Arithmetik

- Pointer können mit == und != *verglichen* werden
- Pointer können mit ++ *inkrementiert* und mit -- *dekrementiert* werden
- Integer können zu Pointern *addiert* (+) oder *subtrahiert* (-) werden
- bei den arithmetischen Operationen wird die Größe des Datentyps berücksichtigt, auf den der Pointer zeigt
  - bei einem `char*`-Pointer erhöht ++ den Wert um 1
  - bei einem `int*`-Pointer erhöht ++ den Wert um 4
  - bei einem `long*`-Pointer erhöht ++ den Wert um 8
  - ...

Das folgende Beispiel zeigt, wie man mithilfe der Pointer-Arithmetik die einzelnen Bytes einer Zahl auslesen kann. Dabei sieht man auch, dass die Zahlen auf Intel-Plattformen im Little-Endian-Format abgelegt sind.

```
typedef unsigned char byte;
int i = 0xcafebabe;
byte* bp = (byte*) &i;

while (bp < (byte*)&i + 4) {
  printf("%x ", *bp & 0xff);
  bp++;
}
```

Ausgabe

```
be ba fe ca
```

Die Funktionsweise des Programms zu verstehen, kann als kleine Übung in Pointer-Arithmetik verstanden werden. Die Casts auf `byte*` sind nötig, da man Pointer unterschiedlichen Typs

nicht vergleichen und zuweisen darf (siehe unten). Beachten Sie auch, dass es man die `while`-Schleife auch als `while (bp < (byte*)(&i + 1)){` hätte schreiben können – es kommt also sehr auf die Klammern an.

2.5 Strict Aliasing Rule [31]

Strict Aliasing Rule

Die **Strict Aliasing Rule** ist eine Regel, die festlegt, wie Zeiger auf unterschiedliche Datentypen verwendet werden können. Sie wurde eingeführt, um die Optimierungsmöglichkeiten des Compilers zu verbessern.

Sie besagt, dass auf ein Objekt (Variable oder Speicherbereich) nicht über Zeiger eines anderen Typs zugegriffen werden darf, es sei denn, es handelt sich um einen `char` oder `unsigned char`-Pointer. Das bedeutet, dass zwei Pointer unterschiedlicher Typen nicht auf dasselbe Objekt zeigen dürfen.

Durch diese Regel kann der Compiler effizienteren Maschinencode generieren, da er sicher sein kann, dass bestimmte Speicherzugriffe keine Seiteneffekte auf andere Objekte haben.

Strict Aliasing Rule

- Zwei Pointer unterschiedlichen Typs dürfen nicht auf dasselbe Objekt zeigen
- Ausnahme: `char*` und `unsigned char*` dürfen auf andere Typen zeigen

Verboten

```
int i = 99;
float *fp = (float*) &i; /* Fehler!!! */
*fp += 0.7f;
```

Erlaubt

```
int i = 99;
char *cp = (char*) &i; /* OK */
*cp = 5;
```

Ein Verstoß gegen diese Regel führt zu sogenanntem „undefined behavior“. Auf dieses Konzept und seine Konsequenzen wird in einem späteren Kapitel noch detailliert eingegangen.

Will man Daten zwischen zwei unterschiedlichen Typen austauschen, muss man über die `memcpy`-Funktion gehen und die Daten umkopieren.

- Verwendung von `memcpy` für Konvertierung

memcpy

```
int i;
float cp = 42.23;

/* Korrekte Umwandlung mit memcpy */
```

```
memcpy(&i, &cp, sizeof(float));

printf("%d\n", i); /* 1109977989 */
```

Die kopierten Daten dürfen sich bei `memcpy` nicht überlappen.

2.6 void-Pointer [33]

Nicht immer kennt man den Typ des Objektes, auf das ein Pointer zeigt. Deswegen braucht man einen Pointer-Typ, der auf jedes beliebige Objekt zeigen kann, so wie in Java eine Referenz vom Typ `Object` jedes Objekt referenzieren kann.

Diese Aufgabe übernimmt in C der `void*`-Pointer.

`void*`

Void-Pointer `void*` ist der generische Pointer

Void-Pointer

- kann jedem anderen Pointer zugewiesen werden (mit Cast)
- jeder andere Pointer kann dem `void`-Pointer zugewiesen werden
- verhält sich bei der Pointer-Arithmetik wie ein `char*`-Pointer

```
int* ip;
void* vp = ip;
sizeof(*vp) == 1;
```

Mithilfe eines Casts kann man den `void*`-Pointer einem Pointer eines anderen Pointer-Typs zuweisen.

```
int i = 7;
void* vp = &i;
int* ip = (int*) vp; /* cast von void* auf int*
printf("%d\n", *ip); /* Ausgabe: 7 */
```

Eine Variable vom Typ `void*` kann man nicht dereferenzieren, weil der Compiler den dahinter liegenden Datentype nicht kennen kann. Im obigen Beispiel würde deswegen `*vp = 5;` zu einem Compilerfehler führen. Nur mit einem entsprechendem Cast kann man dem Compiler die nötige Information geben, z. B. `*((int*)vp)`.

```
int i = 7;
void* vp = &i;
printf("%d\n", *((int*)vp) ); /* Ausgabe: 7 */
```

Kommt ein `void*`-Pointer in einem Ausdruck der Pointer-Arithmetik vor, dann wird er wie ein `char*`-Pointer betrachtet, sodass z. B. ein ++ die Adresse um 1 erhöht.

```c
char s[] = "Hallo"; /* char[] ist identisch mit char* */
void* vp = s;
printf("%p\n", vp); /* 0x7ffd650d9126 */
vp++;
printf("%p\n", vp); /* 0x7ffd650d9127 */
```

2.7 NULL-Pointer [34]

Welchen Wert hat ein Pointer, der auf nichts zeigt? Java verwendet hier für seine Referenzen den Wert `null`, C einfach den Zahlenwert `0`, da `0` keine gültige Adresse ist. Da aber `0` keine Adresse ist, ist der technisch korrekte Wert des sogenannten Null-Pointers `(void*)0`.

Will man ausdrücken, dass ein Pointer auf nichts zeigt, setzt man ihn auf `0` (`NULL`) → **null pointer**

NULL

null pointer

> An integer constant expression with the value 0, or such an expression cast to type void *, is called a null pointer constant. If a null pointer constant is converted to a pointer type, the resulting pointer, called a null pointer, is guaranteed to compare unequal to a pointer to any object or function.
>
> ISO C specification §6.3.2.3

Beispiel: null-Pointer als Zahl

```c
/* null Pointer */
int *px = (void*) 0;
```

Damit man nicht ständig `(void*)0` schreiben muss, liefert C ein Präprozessormakro `NULL`, das genau durch diesen Ausdruck ersetzt wird.

Beispiel: null-Pointer über NULL Macro

```c
/* null Pointer mit NULL Macro */
int *px = NULL;
```

2.8 Adresse von Objekten [36]

Wie kommt man an die Adresse eines Datenobjektes, um diese in einem Pointer zu speichern?

- **Adresse** eines Datenobjekts mit dem Typ `typ`, z. B. `int x`

  Adresse

  - kann über `&x` geholt werden
  - hat den Datentyp `type*`, z. B. `int*`
  - ist 4 Byte (32 Bit-Plattform) oder 8 Byte (64 Bit-Plattform) groß

- kann selbst wieder über einen Pointer referenziert werden `type**`, z. B. `int**`

- **Größe** eines Datenobjekts mit dem Typ `typ`, z. B. `int x` Größe
  - kann über `sizeof(typ)` oder `sizeof(x)` bestimmt werden
  - hat den Datentyp `size_t` (nicht `int`!)
  - wird in Bytes gemessen

Pointer auf Pointer werden in C hauptsächlich für zwei Anwendungsfälle eingesetzt: Pointer auf Pointer

1. Mehrdimensionale Arrays, also z. B. `char**` für ein Array von Zeichenketten, wie wir es aus `argv`kennen.
2. Übergabe von Pointern an Funktionen, die in der Funktion verändert werden sollen (pass-by-value).

Das folgende Beispiel zeigt die Verwendung von einem Pointer auf einen Pointer. Hier wird der Parameter `mem` in der Funktion `my_malloc` geschrieben und muss deshalb, wegen des Pass-By-Value in C, als `int**` übergeben werden. Die Fehlerbehandlung ist weggelassen.

```c
#include <stdlib.h>
#include <stdio.h>

void my_malloc(int** mem, size_t num) {
    *mem = malloc(num * sizeof(int));
}

int main(int argc, char** argv) {
    int* array;
    my_malloc(&array, 10); /* error handling fehlt */
    array[0] = 23;
    array[1] = 42;
    printf("%d %d\n", array[0], array[1]); /* -> 23 42 */
    free(array);
}
```

Die Funktion `sizeof()` liefert einen Wert vom Typ `size_t`, weil dieser Typ speziell dafür entworfen wurde, die Größe von Objekten in Byte zu repräsentieren und dabei die Anforderungen der Plattform zu berücksichtigen. Es ist ein vorzeichenloser Ganzzahltyp, der so dimensioniert ist, dass er groß genug ist, um die größte mögliche Größe eines Objekts auf der Plattform zu speichern. `int` wäre zum eine vorzeichenbehaftet und zum anderen zu klein für 64 Bit-Plattformen.

2.9 Dynamische Speicherverwaltung [37]

In C muss sich die Entwicklerin selbst um die Verwaltung des Speichers kümmern. Dazu bietet die Sprache in der Standardbibliothek eine Reihe von Funktionen an, um Speicher zu allozieren und wieder freizugeben.

- `void* malloc (size_t size)` — malloc
  - alloziert einen Speicherblock mit `size` Bytes
  - Speicher wird nicht initialisiert
- `void* calloc (size_t n, size_t elsize)` — calloc
  - alloziert Speicher für ein Array mit `n` Elementen der Größe `elsize`
  - initialisiert den Speicher mit `0`
- `void* realloc (void *ptr, size_t size)` — realloc
  - vergrößert den Speicherblock, auf den `ptr` zeigt auf die Größe `size` Bytes
  - gibt einen Pointer zurück (kann andere Adresse sein als `ptr`)
  - kopiert den Inhalt des Blocks, auf den `ptr` zeigt um
- `void free (void *ptr);` — free
  - gibt den Speicher frei, auf den `ptr` zeigt
  - Speicher muss mit einer der x`alloc`-Funktionen alloziert worden sein
  - darf nur genau ein mal pro `ptr` aufgerufen werden
  - darf mit `NULL` aufgerufen werden

Wenn man Speicher mit `malloc` anfordert, muss man den Speicher danach selbst initialisieren. Hierzu dient die Funktion `memset`. Bei `calloc` kann man sich das ersparen, da die Funktion den Speicher selbständig initialisiert. Hier ist aber zu beachten, dass man nicht die Größe des gewünschten Speichers in Byte angibt, sondern die Anzahl der Elemente mit einer bestimmten Größe. Wenn man die Größe auf 1 setzt, kann man natürlich auch wieder die Größe in Bytes angeben.

Bei `realloc` muss man immer daran denken, dass man einen komplett neuen Speicherbereich bekommen kann, und nicht der alte vergrößert wurde. D. h. man muss die zurückgegebene Adresse für weitere Zugriffe verwenden.

C ist eine Programmiersprache, die keine Fehler verzeiht und keinerlei Sicherheitsnetz bietet. Im Gegenzug läuft das Programm mit maximaler Performance, weil keine komplexen Laufzeitprüfungen stattfinden.

Todsünden im Umgang mit Pointern

- Zuweisung an einen Pointer, der auf kein Objekt zeigt
  ```
  int* p;
  *p = 7; /*schreibt irgendwo in den Speicher */
  ```
- Speicher mit `malloc()` allozieren und nicht wieder freigeben (⇒ *Memory-Leak*)
- Rückgabewert von `malloc()` nicht auf `NULL` prüfen
- vor oder hinter den allozierten Speicher greifen

- Speicher mehr als einmal mit `free()` freigeben
  ⇒ *Double Free*
- Zugriff auf Speicher, nachdem er mit `free()` freigegeben wurde
  ⇒ *Use after Free*
- Pointer auf Stack-Speicher aus Funktion herausgeben

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

Jedes dieser Probleme kann schwerwiegende Sicherheitslücken aufreißen oder das Programm sporadisch zum Absturz bringen. Fehler, die durch das Memory-Management entstehen sind schwer zu finden und können lange unbemerkt bleiben.

Memory-Leaks

- **Memory-Leaks** verbrauchen langsam den gesamten verfügbaren Speicher, bis das Programm abstürzt. Es gibt C-Programme, die lange Zeit laufen und dann plötzlich wegen Speichermangel crashen. Insbesondere für langlaufende Server-Prozesse ist dies ein großes Problem. Da sich Memory-Leaks nur im laufenden Betrieb und nach einiger Zeit zeigen, sind sie schwer zu finden.

malloc never fails

- `malloc()`-Rückgabewert nicht prüfen: Es gibt Entwickler, die glauben, dass ein Aufruf von `malloc()` immer erfolgreich ist (**malloc never fails**). Dies muss aber nicht so sein, da der Speicher ausgehen oder der angeforderte Speicherblock zu groß sein könnte. In diesem Fall gibt `malloc()` den Wert `NULL` zurück und das Programm sollte sich im Normalfall sauber beendet. Prüft es den Rückgabewert aber nicht, verwendet es einen Zeiger mit dem Wert `NULL` für seine weiteren Operationen, was zu unvorhersagbaren Problemen führen kann.

Segmentation Fault

- Vor- oder hinter den Speicher greifen: Über `malloc()` allozierter Speicher liegt irgendwo im RAM des Computers. Direkt davor oder dahinter können sich andere Daten befinden oder Verwaltungsstrukturen des Heaps. Es ist ebenso denkbar, dass der Speicher überhaupt nicht beim Betriebssystem angefordert wurde und ein **Segmentation Fault** ausgelöst wird. In jedem Fall kann man viele Probleme provozieren, wenn man auf Adressen zugreift, die einem nicht gehören. Von Abstürzen bis ernsthaften Sicherheitsmängeln ist alles denkbar.

double free

- Speicher mehrmals freigeben (**double free**): Wenn Speicher mit `free()` freigegeben wird, dann werden entsprechenden Verwaltungsinformationen im Heap aktualisiert. Gibt man den Speicher dann erneut frei, ist vollkommen undefiniert, was dann passiert. Absturz, Sicherheitsloch… lassen Sie sich überraschen.

use after free

- Zugriff nach Freigabe (**use after free**): Nachdem der Speicher mit `free()` freigegeben wurde kann er an das Betriebssystem zurückgegeben worden sein, erneut vergeben worden sein oder immer noch frei. Benutzen darf man ihn auf jeden Fall nicht mehr.

pointer to stack memory

- Pointer auf Stack-Speicher aus Funktion herausgeben (**pointer to stack memory**): Daten auf dem Stack werden nach dem Zurückkehren der Funktion automatisch freigegeben. Tatsächlich werden die Daten aber aus Performance-Gründen nicht überschrieben, sondern stehen weiterhin im Hauptspeicher. Deswegen kann es sein, dass ein Pointer auf den Stack einer bereits beendeten Funktion noch eine Weile sinnvolle Daten lie-

fert, und zwar so lange, bis dieser Stackspeicher von einem anderen Funktionsaufruf überschrieben wurde. Definiert ist das Verhalten hier aber nicht.

Für den letzten Fall hier ein Beispiel, mit clang 7 compiliert. Die Ausgabe hängt davon ab, wie der Compiler das Stack-Layout durchführt.

```c
#include <stdio.h>

void secret(void) {
    char secret[6] = { 'P', 's', 's', 't', '!', '\0' };
    printf("%p\n", secret);
}

char* no_secret(void) {
    long l = 0;
    return (char*) &l;
}

int main(int argc, char** argv) {
    char* s = no_secret();
    printf("%p\n", s);
    secret();
    printf("%s\n", s);
}
```

Ausgabe

```
0x7ffe463aaa78
0x7ffe463aaa7a
pPsst!
```

Kapitel 3

Funktionen

3.1 Video zum Kapitel [42]

Link zu YouTube

3.2 Syntax [43]

In C-Programmen stehen einem, neben der einzelnen Datei, nur die Funktionen als Strukturierungsmöglichkeit zur Verfügung. D. h. C-Programme bestehen aus Funktionen, die Daten bekommen und Daten zurückgeben.

Die **Syntax** für die Deklaration von C-Funktionen ist identisch zu der von Methoden in Java – kein Wunder, da Java sich die Syntax hier einfach bei C abgeschaut hat.

Syntax

- Syntax ist identisch zur Methodendeklaration in Java

```
RETURN_TYPE NAME(TYPE PARAM, TYPE PARAM, ...) {
    RUMPF
}
```

- Wert wird mit `return` zurück gegeben
- hat die Funktion keinen Rückgabewert, so ist deklariert man den Rückgabetyp `void`

Beispiel

```
int add(int a, int b) {
   return a + b;
}
```

Was in C natürlich fehlt ist die Angabe einer Sichtbarkeit mit `public`, `protected` und `private`, wie wir sie aus Java kennen. Mit dem Schlüsselwort `static` (siehe unten) kann man die Sichtbarkeit einer Funktion auf die aktuelle Quelltextdatei beschränken. Ohne Zusatz sind Funktionen immer global sichtbar.

3.3 Leere Parameterliste [44]

leerer Parameterliste

void

Anders als in Java, wird eine Funktion mit **leerer Parameterliste** nicht mit zwei leeren Klammern `()` deklariert, sondern die Abwesenheit von Parametern muss durch das Schlüsselwort `void` angezeigt werden.

Wenn eine Funktion einfach nur mit einer leeren Liste `()` deklariert wird, bedeutet dies, dass diese Funktion beliebig viele Parameter nehmen kann.

An dieser Stelle weicht C von C++ ab, denn C++ hält sich an den Java-Stil und signalisiert fehlende Parameter mit `()`.

Achtung: Abweichung zu Java

- `func()` ⇒ Funktion mit *beliebig vielen Parametern*
- `func(void)` ⇒ Funktion *ohne Parameter*

```c
int f1() { } /* beliebig viele Parameter */

int f2(void) { } /* keine Paramater */

int main(int argc, char** argv) {
  f1();
  f1(1, 2); /* <- Kein Compile-Fehler */
  f();
  f2(1, 2); /* <- Fehler */
}
```

Dieser Mechanismus ist auch der Grund, warum man manchmal C-Programme sieht, bei denen die `main`-Funktion als `int main()` deklariert ist und nicht als `int main(int argc, char** argv)`. Hier interessieren die Parameter nicht und man lässt sie bei der Definition der Funktion einfach weg.

Es ist im Rahmen einer defensiven C-Programmierung sinnvoll, immer alle Funktionen ohne Parameter mit `void` zu deklarieren, damit der Compiler falsche Aufrufe sofort erkennt und meldet.

3.4 Deklaration vs. Definition [45]

Java kennt keine strenge Unterscheidung zwischen der Deklaration und Definition einer Funktion. In C sind diese beiden Begriffe aber streng auseinanderzuhalten. Eine Deklaration macht die Funktion nur bekannt und kann beliebig oft erfolgen, eine Definition liefert hingegen ihren Funktionsrumpf – also ihre Implementierung – und darf nur einmal in einem Programm vorkommen.

Deklaration
Prototyp

- (Funktions-) **Deklaration** (aka **Prototyp**) macht dem Compiler bekannt
  - den Namen der Funktion
  - den Rückgabetyp
  - die Parameter

Definition

- (Funktions-) **Definition** liefert die Implementierung einer Funktion
- Funktion kann
  - *beliebig oft deklariert*
  - nur *einmal definiert* werden
- durch die Definition wird die Funktion auch deklariert

Diese Unterscheidung wurde schon früher diskutiert.

Anders als in Java müssen in C die Funktionen vor ihrer Benutzung deklariert sein das heißt der Compiler schaut nicht weiter unten im Quelltext, ob dort noch irgendwo die Funktion vorkommt. Eine Definition liefert auch gleichzeitig eine Deklaration der Funktion.

- Die *Deklaration* muss vor der Benutzung erfolgen
  - ist die Funktion vor der Benutzung definiert, muss sie nicht mehr deklariert werden
  - Deklarationen finden sich meist in `.h`-Dateien
  - Definitionen finden sich meist in `.c`-Dateien

Deklaration

```
int add(int a, int b);
```

Definition

```
int add(int a, int b) {
    return a + b;
}
```

Im folgenden Beispiel wird die `add`-Funktion von `calc()` bereits vor deren Definition verwendet. Deswegen muss eine Deklaration erfolgen. Andernfalls käme es zu einem Compiler-Fehler.

Beispiel: Deklaration vs. Definition

```c
/* Deklaration */
int add(int a, int b);

/* Benutzung */
void calc(void) {
    int result;
    result = add(5, 7);
}

/* Definition nach der Benutzung */
int add(int a, int b) {
    return a + b;
}
```

Da die Definition einer Funktion auch gleichzeitig deren Deklaration beinhaltet, muss in diesem Beispiel keine Deklaration mehr erfolgen: add ist vor der Benutzung bereits definiert und damit deklariert.

Beispiel: Deklaration vs. Definition

```c
/* Definition vor der Benutzung, keine Deklaration nötig */
int add(int a, int b) {
    return a + b;
}

/* Benutzung */
void calc() {
    int result;
    result = add(5, 7);
}
```

3.5 Aufteilung .c und .h-Datei [49]

Wie bereits diskutiert, ist es in C üblich die Deklaration und Definition auf verschiedene Dateien aufzuteilen. Die Deklarationen findet man üblicherweise in den .h-Dateien (**Header-Files**) die Definitionen in den dazugehörigen .c-Dateien.

Header-Files

Wenn eine Funktion nur innerhalb einer Quelltextdatei benötigt wird und diese auch nicht an andere Teile des Programms exportiert werden soll, nimmt man sie nicht in die Header-Datei auf.

- Deklarationen finden sich in .h-Dateien, Definitionen in .c-Dateien
- .c-Datei sollte ihre korrespondierende .h-Datei selbst inkludieren (verhindert Abweichungen Definition ↔ Deklaration)

greeter.h

```h
#ifndef GREETER_H
#define GREETER_H
/* Enum für die verschiedenen Tageszeiten. */
typedef enum { MORGEN, ABEND } TagesZeit;

/*
 * Grüßt die als n übergebene Person passend zur Tageszeit z.
 */
void greet(TagesZeit z, char* n);
#endif
```

Die Header-Datei enthält die Deklarationen der Funktion `greet()` und einer von dieser Funktion verwendeten Enumeration `TagesZeit`. Damit stellt sie alle Informationen zur Verfügung, die ein Nutzer der Funktion benötigt.

greeter.c

```c
#include "greeter.h"
#include <stdio.h>

void greet(TagesZeit zeit, char* name) {
    printf("%s %s", zeit == MORGEN ? "Guten Morgen" : "Guten Abend", name);
}
```

user.c

```c
#include "greeter.h"
int main(int argc, char** argv) {
    greet(ABEND, "Thomas");
}
```

In der dazugehörigen `.c`-Datei finden sich die Implementierung der Funktion (ihre Definition). Der Verwender der Funktionalität `user.c` inkludiert die Header-Datei und kennt somit die Deklaration der Funktion `greet()`. Deren Definition ist unwichtig und wird vom Linker später hinzugefügt.

3.6 Pass-by-Value [51]

Pass-By-Value

Pass-By-Reference

Beim Aufruf einer Funktion in C werden die Parameter als **Pass-By-Value** übergeben. Damit entspricht das Verhalten von C exakt dem von Java: Alle Werte werden beim Funktionsaufruf kopiert. Möchte man stattdessen die Inhalte der Variablen in der Funktion verändern, also ein **Pass-By-Reference** durchführen, muss man dies mit Pointern simulieren. Java verhält sich hier genauso, nur dass Objekte als Referenzen übergeben werden, die beim Funktionsaufruf kopiert werden.

- Alle Funktionsparameter werden per *Pass-By-Value* übergeben
- Mit Pointern kann man aber jederzeit *Pass-By-Reference* simulieren

Pass-By-Value

```
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main(int argc, char** argv) {
    int a = 42, b = 23;
    swap(a, b);
    printf("a=%d, b=%d", a, b); /* a=42, b=23 */
}
```

Die `swap`-Funktion funktioniert nicht, weil die Integer-Werte `a` und `b` innerhalb der Funktion *Kopien* der Werte von `main` sind.

Will man, dass die Funktion die Werte vertauschen kann, muss man Pointer verwenden.

Pass-By-Value mit Pointer

```
void swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main(int argc, char** argv) {
    int a = 42, b = 23;
    swap(&a, &b);
    printf("a=%d, b=%d", a, b); /* a=23, b=42 */
}
```

Durch die Verwendung eines Pointers, sind in der Funktion keine Kopien der Werte vorhandne, sondern die Funktion kennt die Speicheradressen der Variablen. So kann sie deren Inhalte vertauschen.

3.7 static [53]

`static`

Das Schlüsselwort `static` ist für Java-Programmierer besonders verwirrend, weil es in C eine völlig andere Bedeutung hat: Während es in Java dafür sorgt, dass Variablen und Methoden an der Klasse hängen und nicht mehr am Objekt, schränkt des in C bei globalen Variablen und Funktionen deren Sichtbarkeit ein.

- Werden *Funktionen* oder *globale Variablen* als `static` gekennzeichnet, dann sind sie außerhalb der Datei nicht sichtbar (→ `protected` in Java)

```
static char gruss[] = "Guten Tag";
static void gruesse(void) {
    printf("%s\n", gruss);
}
```

Deklariert man in C eine lokale Variabel als der `static`, dann handelt es sich gar nicht um eine lokale Variable, sondern um eine globale Variable, deren Sichtbarkeit auf diese Funktion beschränkt ist. Wie alle globalen Variablen wird sie nicht auf dem Stack, sondern im Daten-Segment alloziert. Die Initialisierung der Variabel in der Funktion wird nur ein einziges Mal, beim ersten Aufruf durchgeführt.

- Lokale Variablen mit `static` werden nicht auf dem Stack alloziert, sondern im Daten-Segment und werden nur einmal initialisiert

```
#include <stdio.h>

void counter(void) {
    static int c = 1;
    printf("%d\n", c++);
}

int main(void) {
    counter(); /* 1 */
    counter(); /* 2 */
    counter(); /* 3 */
}
```

In dem Beispiel sieht man deutlich, dass die Variable `c` über die Funktionsaufrufe hinweg ihren Wert behält. Beim ersten Aufruf von `counter()` wird sie mit dem Wert 1 initialisiert, bei allen weiteren Aufrufen findet Initialisierung aber nicht mehr statt, sodass sie ihren Wert über die Funktionsaufrufe hinweg behält.

3.8 Schlüsselwort const [55]

`const`

Ein Schlüsselwort in C, das es in Java überhaupt nicht gibt, ist `const`. In Java hat man darauf verzichtet, weil man es als zu komplex angesehen hat und sich stattdessen für `final` entschieden, das eine andere Semantik als `const` hat. Mit `const` kann man in C kennzeichnen, dass eine Variabel nicht verändert werden darf.

Relativ kompliziert wird die Verwendung von `const`, wenn es sich um Pointer handelt: Das `const` kann sich entweder auf den Pointer beziehen oder aber auf das Objekt auf das der Pointer zeigt oder sogar auf beides.

- Mit `const` kann deklariert werden, dass ein Argument nicht geändert wird
- bei Funktionen meist nur sinnvoll für Pointer (wg. Pass-by-Value)
- bei Pointern muss man unterscheiden
  - Pointer auf einen konstanten Wert: Pointer kann geändert werden, Wert nicht
    `const int *ptr` oder `int const *ptr`
  - Konstanter Pointer auf einen Wert: Pointer kann nicht geändert werden, Wert jedoch schon
    `int *const ptr`
  - Konstanter Pointer auf einen konstanten Wert: Weder Pointer noch Wert können geändert werden
    `const int *const ptr`

Bei Funktionsparametern ist `const` im Allgemeinen nur bei Pointern sinnvoll, weil die Parameter ohnehin by-value übergeben werden und `const` hier wenig bringt.

Konstanter Wert

```
int i = 10, j = 20;
const int *ptr = &i; /* Pointer auf konstanten Wert */

ptr = &j; /* Ok */
*ptr = 100; /* Fehler */
```

Konstanter Pointer

```
int i = 10, j = 20;
int *const ptr = &i; /* Konstanter Pointer */

ptr = &j; /* Fehler */
*ptr = 100; /* Ok */
```

Konstanter Pointer auf konstanten Wert

```
int i = 10, j = 20;
const int *const ptr = &i; /* Konstanter Pointer */

ptr = &j; /* Fehler */
*ptr = 100; /* Fehler */
```

Da der Compiler bei `const` deklarierten Objekten Optimierungen durchführen kann, z. B. die Daten in Read-Only-Speicher legen, ist es verboten ein `const` einfach durch einne Cast zu entfernen.

`const`

- darf nicht durch einen Cast entfernt werden, wenn das Original-Datenobjekt als `const` deklariert wurde
- führt zu *undefined behavior*

```
int i = 12;
const int j = 12;
const int *ip = &i;
const int *jp = &j;
*(int *)ip = 42; /* OK aber nicht schön */
*(int *)jp = 42; /* undefined behavior */
```

Die Abgründe des **undefined behaviors** werden weiter unten noch diskutiert und näher betrachtet.

undefined behaviors

3.9 Schlüsselwort extern [59]

Den Unterschied zwischen der Deklaration und der Definition einer Funktion kann der Compiler am fehlenden Rumpf der Funktion erkennen, sodass kein spezielles Schlüsselwort benötigt wird, um die Fälle zu unterscheiden. Deswegen wird hier das Schlüsselwort `extern` nicht benötigt und der Compiler fügt es automatisch hinzu.

extern

Bei Variablen gibt es keine Möglichkeit, ohne zusätzliches Schlüsselwort, die Deklaration von der Definition zu unterscheiden. Deswegen *muss* hier bei der Deklaration `extern` eingesetzt werden.

- Mit `extern` kann man Variablen deklarieren, die an anderer Stelle (anderer Datei) deklariert werden
- `extern`-Deklaration führt keine Initialisierung durch

file1.c

```
extern char buffer[]; /* kein Speicher reservieren */

int main(int argc, char** argv) {
    printf("%s\n", buffer);
}
```

file2.c

```
char buffer[] = "Hallo Welt!"; /* Speicher reservieren */
```

Eine Variable muss – genauso wie eine Funktion – vor ihrer Benutzung deklariert werden. Wenn eine Variable, die in einer anderen Datei (**Compilationseinheit**) definiert wurde, benutzt werden soll, muss man sie entsprechend deklarieren. Durch das Schlüsselwort `extern` vor der Variable wird angezeigt, dass kein Speicher reserviert werden soll, da dies an anderer Stelle bei der Definition erfolgt.

Compilationseinheit

In diesem Beispiel wird `buffer` in `file1.c` nur deklariert und in `file2.c` dann definiert. Der Linker kümmert sich am Ende darum, dass die `main`-Funktion auf die richtige Variable im Speicher zugreift.

Die Objektdateien enthalten die notwendigen Informationen zu den erwarteten bzw. bereitgestellten Variablen wieder in Form von Exports und Imports.

```
$ rabin2 -i file1.o
[Imports]
nth vaddr      bind   type   lib name
4   ---------- GLOBAL NOTYPE     buffer
5   ---------- GLOBAL NOTYPE     puts

$ rabin2 -E file2.o
[Exports]
nth paddr      vaddr      bind   type size lib name   demangled
2   0x00000040 0x08000040 GLOBAL OBJ  12       buffer
```

In den Expoers von `file2.o` kann man erkennen, dass die Größe der Variable dem Linker über die Exports bekannt gegeben wird.

3.10 Funktionspointer [60]

C hat – anders als Java – von Anfang an Funktionen als „first class citizens“ betrachtet, sodass man Funktionen wie jedes andere Datenobjekt verwenden kann. Das Mittel dazu sind **Funktionspointer**, d. h. Pointer, die auf Funktionen zeigen. Diese Pointer können wie alle anderen Pointer auch übergeben und zugewiesen werden. Pointer-Arithmetik scheidet aus naheliegenden Gründen aus.

Funktionspointer

Leider ist die Syntax für die Deklaration von Variablen, die Funktionspointer enthalten können etwas sehr kompliziert ausgefallen, sodass es sogar Webseiten gibt, die die Syntax übersetzen.

Funktionen können über *Funktionspointer* wie Daten übergeben und verwaltet werden

- `int func()`: Funktion, die einen `int` zurückgibt
- `int *func()`: Funktion, die einen Pointer auf einen `int` zurückgibt
- `int (*func)()`: Pointer auf eine Funktion, die einen `int` zurückgibt
- `int *(*func)()`: Pointer auf eine Funktion, die einen Pointer auf einen `int` zurückgibt
- `int *(*func)(char*)`: Pointer auf eine Funktion, die einen Pointer auf einen `int` zurückgibt und einen Pointer auf `char` als Parameter hat

Über den **Aufrufoperator** `()` kann die Funktion, auf die der Funktionspointer zeigt, dann aufgerufen werden.

Aufrufoperator
()

Beispiel: Funktionspointer

```c
int rechne(int (*func)(int a, int b), int a, int b) {
    return (*func)(a, b);
}

int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }

int main(int argc, char** argv) {
    int r;

    r = rechne(add, 2, 17); /* 19 */
    printf("%d\n", r);

    r = rechne(sub, 2, 17); /* -15 */
    printf("%d\n", r);
}
```

Das Beispiel verwendet Funktionspointer, um der Funktion `rechne()` zur Laufzeit eine beliebige Implementierung einer arithmetischen Operation zu übergeben.

Hier einmal dasselbe Beispiel in Java unter Verwendung von Lambdas:

```java
import java.util.function.IntBinaryOperator;

public class Rechner {

    public static int rechne(IntBinaryOperator func, int a, int b) {
        return func.applyAsInt(a, b);
    }

    public static void main(String[] args) {
        IntBinaryOperator sub = (a, b) -> a - b;
        IntBinaryOperator add = (a, b) -> a + b;

        int r;

        r = rechne(add, 2, 17);
        System.out.println(r); /* 19 */


        r = rechne(sub, 2, 17);
        System.out.println(r); /* -15 */
    }
}
```

3.11 Varag-Funktionen [62]

In C ermöglichen *vararg-Funktionen* die Definition von Funktionen mit einer variablen Anzahl von Argumenten. Das bedeutet, dass eine Funktion eine unterschiedliche Anzahl von Argumenten akzeptieren kann, die zur Laufzeit übergeben werden.

Da das Konzept von Funktionen mit variabler Anzahl von Parameters keine Fähigkeit der Prozessoren ist, C aber ja direkt auf der Hardware läuft, muss dieses Feature recht umständlich durch gewisse Tricks realisiert werden. Eine direkte Unterstützung auf der Hardwareseite gibt es nicht.

Vararg-Funktionen

- Syntax `funktionsname(PARAM, [PARAM], ...)`
- Ellipse `...` zeigt die variablen Parameter an
- Variable Anzahl von Parametern (vgl. `printf`)
- Müssen mindestens einen „normalen“ Parameter haben
- Müssen über `va_start`, `va_arg`, `va_end` und `va_list` aus `<stdarg.h>` realisiert werden
  $\Rightarrow$ *kein Teil der Sprache*

Die Anforderung an einen „normalen“ Parameter kommt daher, dass der Stack durchwandert werden muss, um die weiteren Parameter einzusammeln. Der normale Parameter liefert die Adresse, ab der diese Untersuchung stattfinden muss.

Als Konsequenz können die Parameter einer solchen Funktion nicht in Prozessorregistern übergeben werden, sondern müssen immer auf den Stack gelegt werden. Dies hat durchaus Auswirkungen auf die Performance der Funktionen.

- `va_list`
  Datentyp (Liste) für den Zugriff auf die Parameter
- `va_start(va_list ap, last)`
  Initialisiert die Liste ausgehend vom letzten „normalen“ Parameter `last`
- `type va_arg(va_list ap, type)`
  Liefert den nächsten Parameter. Typ ist `type`
- `va_end(va_list ap)`
  Gibt den Speicher für die Argumente frei

`va_start`, `va_arg` und `va_end` sind keine C-Funktionen, sondern Makros. Dies erlaubt, dass man z. B. den Typ `va_arg` angibt, was bei einer Funktion nicht möglich wäre.

1. `va_start`: Dieses Makro initialisiert eine `va_list`, die verwendet wird, um auf die variablen Argumente in der Funktion zuzugreifen. Es erwartet den Namen des letzten festen Arguments als Parameter.
2. `va_arg`: Dieses Makro ermöglicht den Zugriff auf den nächsten Parameter in der `va_list`. Der zweite Parameter ist der Datentyp, auf den zugegriffen werden soll. Es gibt den

Wert des Arguments zurück und aktualisiert die `va_list`, um auf das nächste Argument zu verweisen.

3. `va_end`: Dieses Makro muss aufgerufen werden, um die `va_list` nach der Verwendung zu bereinigen.

```c
#include <stdio.h>
#include <stdarg.h>

void printNumbers(int count, ...) {
    va_list args; /* Struktur für Argumente */
    va_start(args, count); /* Vorgang beginnen */

    for (int i = 0; i < count; i++) {
        int num = va_arg(args, int); /* nächsten Wert holen */
        printf("%d ", num);
    }

    va_end(args); /* Speicher freigeben */
}
```

In diesem Beispiel akzeptiert die Funktion `printNumbers` eine variable Anzahl von Argumenten. Sie verwendet die `va_list`, um auf die Argumente zuzugreifen, und druckt sie nacheinander aus. Die Anzahl der Argumente wird als erstes Argument `count` übergeben.

Ausgabe: `10 20 30`

Vorsicht bei der Benutzung

- Keine Prüfung der Parameteranzahl oder Typen durch den Compiler
- Anzahl muss übergeben werden oder sich anderweitig ergeben
- Risiko von schwerwiegenden Fehlern

Fehlerhafte Benutzung

```c
int main(int argc, char** argv) {
    printNumbers(4, 10.9); /* falsche Anzahl */
    return 0;
}
```

Ausgabe

```
1407264456 1407264472 91233720 0
```

Die Verwendung von Vararg-Funktionen in C birgt einige Risiken und Herausforderungen, die berücksichtigt werden sollten:

1. *Mangelnde Typsicherheit:* Vararg-Funktionen bieten keine Typsicherheit. Der Entwickler muss sicherstellen, dass die Argumente korrekt interpretiert werden, da dies nicht vom Compiler überprüft wird.
2. *Keine Informationen über die Anzahl der Argumente*: Da die Anzahl der Argumente zur Laufzeit festgelegt wird, gibt es keine statische Prüfung auf die korrekte Anzahl der Argumente. Dies kann zu Fehlern führen, wenn die Funktion mit einer falschen Anzahl von Argumenten aufgerufen wird oder wenn die Reihenfolge der Argumente nicht richtig ist.
3. *Schwierigkeiten bei der Fehlersuche*: Da der Compiler keine Überprüfung auf Typen oder Anzahl der Argumente durchführt, kann die Fehlersuche in Vararg-Funktionen schwierig sein.
4. *Plattform- und Compilerabhängigkeit*: Die Implementierung von Vararg-Funktionen kann von Plattform zu Plattform und von Compiler zu Compiler variieren. Dies kann zu Inkompatibilitäten führen, insbesondere wenn der Code auf verschiedenen Systemen oder mit unterschiedlichen Compilern verwendet wird.

Kapitel 4

Arrays

4.1 Video zum Kapitel [68]

Link zu YouTube

4.2 Übersicht [69]

C ist keine objektorientierte Programmiersprache, sondern rein prozedural. Deswegen kann man in C auch keine Objekte im Sinne von Java verwalten.

Trotzdem kann man mit Arrays, Structs und Unions höherwertige Datentypen erzeugen.

- C ist nicht objektorientiert ⇒ *keine* Klassen
- Drei Möglichkeiten, höhere Datenstrukturen zu definieren
  - **Arrays**: Sammlung von gleichartigen Werten (→ Java Array) — Arrays
  - **Structs**: Sammlung von Variablen unterschiedlichen Typs (→ Java Klasse ohne Methoden) — Structs
  - **Unions**: Mehrere Variablen, aber nur *ein* Wert zu einer Zeit (nichts Vergleichbares in Java) — Unions

4.3 Arrays [70]

Der einfachste, komplexe Datentyp in C ist das **Array**. Arrays in C sehen denen in Java sehr ähnlich, es gibt aber einige subtile Unterschiede, auf die man als Java-Programmierer schnell hereinfallen kann. — Array

- *Deklaration* über Typ und Anzahl der Elemente
  z. B. `int werte[100]`
- *Zugriff* mit `[x]`
  - Indices gehen von 0 – N-1
  - *keine Überprüfung* der Grenzen
- Werden im Speicher *zusammenhängend* abgelegt (= Java)
- C weiß nicht, wie lang ein Array ist (↔ Java)
  - nur im deklarierenden Block kann man mit `sizeof(a)/ sizeof(a[0])` die Größe bestimmen
  - Programmierer muss die Länge speichern
  - `int x[10]; x[10] = 42;` → subtile Fehler (Speicher-Überschreiber)

Die Deklaration von Arrays und der Zugriff auf die Elemente entspricht weitgehend dem Vorgehen in Java.

```
int i;
int x[3];
x[0] = 1;
x[1] = 2;
x[2] = 3;

for (i = 0; i < 3; i++) {
    printf("%d ", x[i]); /* -> 1 2 3 */
}
```

Man kann, wie in Java, das Array direkt bei der Deklaration mit deiner Liste { . . . } initialisieren. Tut man dies, darf man die Länge weglassen, die der Compiler dann selbst aus der Liste ermittelt.

```
int i;
int x[] = { 1, 2, 3 };
x[2] = 4;

for (i = 0; i < 3; i++) {
    printf("%d ", x[i]); /* -> 1 2 4 */
}
```

Allerdings überprüft der C-Compiler nicht, ob der Zugriff überhaupt innerhalb der Array-Grenzen liegt:

```
int x[] = { 1, 2, 3 };
printf("%d ", x[4]); /* undefiniertes Ergebnis, was auch immer im Speicher liegt */
```

Deswegen muss man als C-Programmierer*in immer die Länge des Arrays an Funktionen, die man aufruft, mitgeben. Das ist der Grund, warum die `main`-Funktion die Signatur `int main(int, char**)` hat, da man andernfalls die Länge der Argumentenliste nicht kennen würde.

- Bei der Übergabe an eine Funktion ist die Größe nicht mehr bekannt
- Größe muss zusätzlich übergeben werden (sehr häufige Fehlerquelle)

```c
#include <stdio.h>
void listElements(int a[], int len) {
    printf("size in function: %ld\n", sizeof(a) / sizeof(int)); /* -> 2 */
    for (int i = 0; i < len; i++) {
        printf("%d\n", a[i]);
   }
}

int main(int argc, char** argv) {
    int a[] = { 1, 2, 3, 4, 5 };
    printf("size in main: %ld\n", sizeof(a) / sizeof(int)); /* -> 5 */
    listElements(a, 5);
}
```

Man sieht in dem Beispiel deutlich, dass in der Funktion `listElements` das Array `a` nur noch als ein Zeiger vom Typ `int*` mit 8 Byte Länge vorliegt, sodass `sizeof(a)/ sizeof(int)` 2 ergibt. Die Funktion kann die Länge des Arrays nicht mehr bestimmen und muss sie deshalb als Parameter `len` übergeben bekommen.

C betrachtet Arrays einfach nur als einen zusammenhängenden Speicherblock. Der Compiler beschafft ausreichend Speicher, um die deklarierten Elemente abzulegen. Bei einem Array `T[n]` vom Typ `T` nach der Formel `sizeof(T)* n`.

Die Array-Variable ist dann – aus Sicht des Compilers – einfach nur ein Zeiger vom Typ `T*` auf das erste Element des Arrays.

Zeiger auf das erste Element

- Arrays werden als **Zeiger auf das erste Element** verwaltet
  `a` ist dasselbe wie `&a[0]`
- Einen Array-Parameter einer Funktion kann man als Array (`int a[]`) oder Pointer (`int* a`) deklarieren
- Man kann Arrays auch per Pointer-Arithmetik behandeln

```c
int a[] = { 1, 2, 3, 4, 5 };
int *p = &a[0]; /* Pointer auf erstes Element */
int *e; /* Pointer auf aktuelles Element */

for (e = p; e < p + 5; e++) {
    printf("%d\n", *e);
```

```
}
```

Der Code des Beispiels ist identisch mit:

```
int i;
int a[] = { 1, 2, 3, 4, 5 };

for (i = 0; i < 5; i++) {
    printf("%d\n", a[i]);
}
```

Aufgrund des Pointer-Characters eines Arrays, sind folgende Konstrukte identisch:

- `&a[0]` ⇔ `a`
- `a[0]` ⇔ `*a`
- `a[i]` ⇔ `*(a + i)`

4.4 Dynamische Arrays [73]

Die Größe eines Arrays muss bei der Deklaration (z. B. `int x[10]`) durch eine Konstante bestimmt sein, da der Compiler bereits den Speicher für das Array reservieren muss. D. h. der Ausdruck `SIZE` in `T v[SIZE]` muss eine Konstante zur Compilezeit sein. Kann man die Größe nicht durch eine Konstante ausdrücken, bleibt nur das Array mithilfe von `malloc()` und `free()` dynamisch zu erzeugen und zu verwalten.

- Die Größe eines Arrays für Deklarationen der Art `int a[10]` müssen zur Compilezeit bekannt sein
- **Dynamische Arrays** müssen mit `malloc` oder `calloc` erzeugt werden

Dynamische Arrays

```
int *a, size = 4;

a = (int*) malloc(sizeof(int) * size); /* Speicher für Array holen */

/* Elemente zuweisen */
a[0] = 1;
a[1] = 2;
*(a + 2) = 3;
*(a + 3) = 4;
/* Array benutzen */
free(a); /* Speicher freigeben */
```

Das vollständige Programm sieht wie folgt aus:

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int *a, size = 4;

    a = (int*) malloc(sizeof(int) * size); /* Speicher für Array holen */

    /* Elemente zuweisen */
    a[0] = 1;
    a[1] = 2;
    *(a + 2) = 3;
    *(a + 3) = 4;

    free(a); /* Speicher freigeben */
}
```

Das Programm hat das Problem, dass es von Hand die Größe des benötigten Speichers per `sizeof(int)* size` berechnet. Eleganter ist es die `calloc`-Funktion zu verwenden, die den Speicher auch direkt mit `0x00` füllt und so verhindert, dass man alte Werte im Speicher sieht.

Verwendung von calloc

```c
int *a, size = 4;

a = (int*) calloc(size, sizeof(int)); /* Speicher für Array holen */

/* Elemente zuweisen */
a[0] = 1;
a[1] = 2;
*(a + 2) = 3;
*(a + 3) = 4;

/* Array benutzen */

free(a); /* Speicher freigeben */
```

4.5 Variable Length Arrays (VLA) [75]

Sowohl die statische Größenangabe zur Compilezeit als auch die Programmierung von dynamischen Arrays mit `malloc` ist teilweise sehr mühsam. Bei `malloc` kommt noch hinzu, dass eine Allokation auf dem Heap mehr Rechenzeit benötigt, als die Verwendung des Stacks, weil der Speicher auf dem Heap verwaltet werden muss. Es gibt zwar mit `alloca` eine Möglichkeit dynamisch zur Laufzeit Speicher auf dem Stack zu belegen, diese Funktion ist aber nicht standardisiert und damit nicht auf allen Plattformen verfügbar.

Aus diesem Grund hat C99 eine weitere Möglichkeit eröffnet, Arrays dynamisch zu erstellen und auf dem Stack zu alloziert.

Variable Length Arrays (VLA) – seit C99

Variable Length Arrays
VLA

- Größe des Arrays ist zur Compilezeit noch *unbekannt*
- Speicher wird erst zur Laufzeit auf dem *Stack* alloziert
- können nur Block- oder Funktions-Scope haben
- können als Funktionsparameter verwendet werden
- kennen, wie normale Arrays, ihre eigene Größe nicht

```c
for (int i = 1; i < 10; i++) {
    int a[i]; /* VLA deklarieren */
    printf("%zu\n", sizeof(a)); /* 4, 8, 12, ... */
}
```

Das VLAs nicht als globale Variablen existieren können liegt daran, dass für globale Variablen der Platz bereits beim Compilieren im Datensegment reserviert werden muss. Zur Laufzeit kann dieses nicht einfach vergrößert werden.

Eine Größe von `0` ist für VLAs nicht zulässig. Außerdem kann man sie – logischerweise – nicht mit einem Array-Initializer `{...}` initialisieren, weil die Größe zur Compilezeit nicht bekannt und damit die Anzahl der Elemente im Initializer ebenfalls nicht bestimmt sind.

Der Speicher für VLAs wird auf dem Stack alloziert, sodass man einen plattformübergreifenden Ersatz für die nicht-standardisierte `alloca`-Funktion bekommt.

alloca

VLAs als *Funktionsparameter*

- Größe muss in der Parameterliste vor dem VLA stehen
  `void f(size_t x, int a[x]){ }`
- Bei Funktions-Prototypen kann ein `*` verwendet werden
  `void f(size_t x, int a[*]);`

Kapitel 5

Strings

5.1 Zeichen [78]

Im Gegensatz zu anderen Programmiersprachen kennt C keinen Datentyp zur Darstellung von **Zeichenketten** (**Strings**), sondern bildet diese einfach als ein Array von Zeichen ab. Damit ist der einzige von C zur Verfügung gestellte Datentyp, um Zeichenketten zu bilden, das einzelne Zeichen: **char**.

Zeichenketten
Strings

char

- Der Datentyp `char` hat 8 Bit und kann so *nur ASCII-Zeichen* darstellen
- für Unicode gibt es spezielle Datentypen (`wchar_t` aus `wchar.h`)
- `ctype.h` bietet Funktionen, um Zeichen auf ihre Art zu prüfen

Funktion	Test	Zeichen
`isalnum(ch)`	alphanumerisch	[a-zA-Z0-9]
`isalpha(ch)`	Buchstabe	[a-zA-Z]
`isdigit(ch)`	Ziffer	[0-9]
`ispunct(ch)`	Satzzeichen	[~!@#%^&...]
`isspace(ch)`	Whitespace	[\t\n]
`isupper(ch)`	Großbuchstabe	[A-Z]
`islower(ch)`	Kleinbuchstabe	[a-z]

Die Bitbreite des Datentyps `char` ist fest in der C-Spezifikation mit 8 Bit verankert und kann nicht geändert werden. Deswegen muss man bei der Verwendung von Unicode auf andere Datentypen ausweichen.

Ein einzelnes Zeichen, das durch den Datentyp `char` repräsentiert wird, kann auch durch ein Literal direkt im Quelltext angegeben werden.

Zeichenliterale repräsentieren ein einziges Zeichen, umschlossen von einfachen Anführungszeichen `'`

Zeichenliterale

Bestimmte Sonderzeichen müssen **escaped**, d. h. speziell notiert, werden

escaped

Escape-Sequenz	Bedeutung
\'	Anführungszeichen '
\"	Anführungszeichen "
\\	Backslash \
\f	Seitenvorschub
\t	Tabulator
\n	Zeilenvorschub **(newline)**
\r	Wagenrücklauf **(carriage return)**
\b	Backspace
\f	Seitenvorschub **(form feed)**

Beispiel:

```c
char a = 'a';
char b = 'b';
char tab = '\t';
char newline = '\n';

printf("%c%c%c%c", a, tab, b, newline); /* a b */
```

5.2 Strings [80]

C-Strings sind eine ausgesprochene Enttäuschung, wenn man andere Programmiersprachen gewöhnt ist. Sie sind hochgradig speichereffizient, erfordern im Gegenzug aber viel Arbeit und Sorgfalt vom Programmierer.

- In Java sind Strings Objekte mit Methoden etc.
- In C gibt es keinen Datentyp für Strings
- Strings sind nur ein Array von `char`, das durch ein `NUL` (`'\0'`) beendet wird
- Ein **String-Literal** (z. B. `"pr3 rocks"`) String-Literal
  - alloziert automatisch Speicher, inklusive Platz für das `'\0'`-Zeichen
  - hat als Wert die Adresse des ersten Zeichens (Pointer)
  - kann *nicht verändert* werden (beliebter Fehler)
- für andere Strings, die keine Literale sind, muss vom Programm Speicher alloziert werden

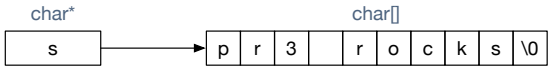
Beachten Sie, dass `NUL` nicht dasselbe wie `NULL` ist. `NUL` bezeichnet das Zeichenliteral `'\0'`, `NULL` hingegen den `void*`-Pointer `(void*)0`. Beide enthalten zwar den Wert `0`, aber sie haben unterschiedliche Datentypen, nämlich `char` und `void*`.

An einer Stelle haben die C-Entwickler Gnade gezeigt und ein Literal für Strings in C spezifiziert. D. h. man kann Strings im Programm direkt angeben (`"pr3 rocks"`) und muss sie nicht als `char[]`-Literal (`{ 'p', 'r', '3', '', 'r', 'o', 'c', 'k', 's', '\0'}` schreiben.

```c
char s[] = "pr3 rocks";

printf("%ld\n", sizeof(s)); /* -> 10 */

s[3] = '2'; /* Pfui, Fehler. Ist ein Literal!!! */
```



Bei der Deklaration eines solchen Literals passieren drei Dinge:

- Es wird ein `char`-Array mit dem Platz für die angegebenen Zeichen + 1 Byte für das `NUL` am Ende alloziert.
- Es wird ein `char*`-Pointer alloziert.
- Dem Pointer wird die Adresse des Arrays zugewiesen.

Eine Definition wie `char s[] = "a"` benötigt deshalb auf einer 64-Bit-Plattform 10 Byte: 8 Byte für den Pointer `s` und zwei Byte für die Zeichen `{'a', '\0'}`.

5.3 Pointer auf Strings [82]

Da C-Strings Arrays sind und Arrays in C als **Pointer auf das erste Element** aufgefasst werden können, kann man C-Strings ebenfalls als Pointer auf das erste Zeichen verstehen.

Pointer auf das erste Element

- Strings werden einfach über einen *Pointer auf das erste Zeichen* verwaltet
  - man kann sie wie Arrays behandeln
    `char s[] = "xyz";`
  - man kann sie wie `char*`-Pointer behandeln
    `char* s = "xyz";`
- *Länge* ergibt sich nur über das `'\0'` am Ende

Pointer-Style

```c
char* s = "pr3 rocks";
char* p;

for (p = s; *p; p++) {
    printf("%c", *p);
}
```

Array-Style

```c
char s[] = "pr3 rocks";
```

```c
int i;

for (i = 0; s[i] != '\0'; i++) {
    printf("%c", s[i]);
}
```

Das Konstrukt `*p` in der `while`-Schleife mag auf den ersten Blick erstaunlich erscheinen, es funktioniert aber, weil in C das Ende eines Strings mit einem Null-Byte (`0x00`) angezeigt wird und `0` grundsätzlich `false` ist.

5.4 Strings kopieren [84]

Will man in C einen String kopieren, kann man dies nicht einfach durch eine Zuweisung machen. Denn bei einer Zuweisung würden die Pointer kopiert und nicht der Inhalt. Dieses Verhaltens entspricht dem von Java, nur dass dies dort nicht auffällt, weil Strings unveränderlich sind und sich deshalb nicht erkennen lässt, dass beide Variablen auf dasselbe Objekt zeigen.

- *Zuweisung* `s = t`
  - kopiert den Pointer, nicht den Inhalt (→ Java)
  - zwei Pointer zeigen auf denselben String (→ Java)
- *Kopieren* von Strings mit `strcpy (t, s)` `strcpy`
  - Speicher für Ziel-String muss vorher angelegt seine
  - Speicher muss groß genug sein (*Vorsicht*)
- *Kopieren* von Strings mit `strdup (s)` `strdup`
  - legt den Speicher selbständig an
  - gibt Pointer auf den neuen String zurück
  - Speicher muss mit `free ()` freigegeben werden `free`

```c
char s[] = "pr3 rocks";

/* Pointer wird kopiert */
char* t = s;
printf("t=%p\n", t); /* t=0x7fff5fa4179e */
printf("s=%p\n", s); /* s=0x7fff5fa4179e */

/* Inhalt wird kopiert */
t = strdup(s);
printf("t=%p\n", t); /* t=0x7f7f524027d0 */
printf("s=%p\n", s); /* s=0x7fff5fa4179e */

strcpy(t, "tpe rocks"); /* es gibt bessere Funktionen */
printf("t=%p\n", t); /* t=0x7f7f524027d0 */
printf("%s\n", t); /* tpe rocks */
```

```
free(t); /* Speicher wieder freigeben */
```

Das Beispiel verwendet zwar die Funktion `strcpy`, diese sollte man aber in der Realität nicht einsetzen, weil sie keine Überprüfung der Länge durchführt. Beim Kopieren eines Strings in einen Buffer, sollte immer die Funktion `strncopy` eingesetzt werden, der man zusätzlich noch die Länge des Puffers mitgeben kann.

```
#define BUFFER_SIZE 20
char buffer[BUFFER_SIZE];
char s[] = "pr3 rocks";

strncpy(buffer, s, BUFFER_SIZE);

/* Wenn der string genau BUFFER_SIZE lang oder länger ist, schreibt
   strncpy kein NUL ans Ende. Deswegen setzen wir dies manuell,
   sodass der String auf jeden Fall terminiert ist */
buffer[BUFFER_SIZE - 1] = '\0';
```

5.5 String-Funktionen [86]

C lässt einen nicht ganz alleine, sondern bietet für den Umgang mit C-Strings eine Reihe von Funktionen an, die die wichtigsten Aufgaben im Umgang mit Strings lösen.

`string.h` enthält eine Reihe von String-Funktionen

- `#include <string.h>`
- Strings müssen *NULL-terminiert* sein
- alle Zielarrays müssen groß genug sein → *keine Prüfung* bei den Standardfunktionen

Einige gängige Funktionen sind

- `char *strcpy(char *dest, char *source)`
  Kopiert `source` nach `dest`
- `char *strlcpy(char *dest, char *source, int num)`
  Kopiert `source` nach `dest` aber maximal `num` Zeichen
- `size_t strlen(const char *source)`
  Länge des Strings (ohne NUL)
- `char *strchr(const char *source, const char ch)`
  Pointer auf erstes Auftreten von `ch` in `source`
- `char *strstr(const char *source, const char *search)`
  Pointer auf erstes Auftreten von `search` in `source`

Beispiel für die Funktionen:

```c
char s[] = "pr3 ist super!";

printf("Länge: %lu\n", strlen(s)); /* Länge: 14 */

char* zahl = strchr(s, '3');
printf("Index '3': %ld\n", zahl - s); /* Index '3': 2 */
printf("Reststring: %s\n", zahl); /* Reststring: 3 ist super! */

char* super = strstr(s, "super");
printf("Index: %ld\n", super - s); /* Index: 8 */
printf("Reststring: %s\n", super); /* Reststring: super! */
```

5.6 String-Formatierung [88]

Umwandlung von Strings ↔ Daten mit

- `sscanf`: `int sscanf(char *string, char *format, ...)` sscanf
  - liest den Inhalt des Strings entsprechend `format`
  - schreibt die Ergebnisse in das 3., 4., 5. etc. Argument
  - gibt die Anzahl der erfolgreichen Umwandlungen zurück
- `sprintf`: `int sprintf(char *buffer, char *format, ...)` sprintf
  - erzeugt einen String entsprechend `format`
  - formatiert 3., 4., 5. etc. Argument
  - schreibt den String in den Puffer (Größe!)
  - gibt die Anzahl der erfolgreichen Umwandlungen zurück

Auch `sprintf` sollte heute nicht mehr verwendet werden, sondern `snprintf`, das eine Längenprüfung zulässt.

Es gibt neben den sehr vielseitigen `sscanf` und `sprintf`-Funktionen noch einfachere Umwandlungen in Zahlen mit den `atoX`-Funktionen:

- `int atoi(const char *nptr);`
- `long atol(const char *nptr);`
- `long long atoll(const char *nptr);`
- `double atof(const char *nptr);`

Die Funktionsweise dürfte sich aus den Signaturen ergeben.

Die String-Formatierung bzw. Umwandlung wird in C durch sogenannte **Formatstrings** gesteuert, die eine Reihe von speziellen Zeichen enthalten, die man am % erkennt. Formatstrings

Codes für den Format-String von `sscanf` (Auswahl)

Code	Bedeutung	Datentyp
%c	einzelnes Zeichen	char
%d	Ganzzahl	int
%f	Fließkommazahl	float
%s	String	char*
%[^c]	alles bis um nächsten Zeichen c	char*

Beispiel: sscanf

```c
char* message = "Hallo Freunde";
char* zahlen = "1 2 3 4";
char s[10], t[10];
int a, b, c, d;

sscanf(message, "%s %s", s, t);
printf("s=%s\n", s); /* Hallo */
printf("t=%s\n", t); /* Freunde */

sscanf(zahlen, "%d %d %d %d", &a, &b, &c, &d);
printf("%d-%d-%d-%d\n", a, b, c, d); /* 1-2-3-4 */
```

Codes für `sprintf`

- Codes für den Format-String von `sprintf` (Auswahl)
- Syntax: `%[width][.precision][length]type`

Code	Bedeutung	Datentyp
%nc	Zeichen mit n Leerzeichen	char
%nd	Integer mit n Leerzeichen	int
%nld	Long mit n Leerzeichen	long
%n.mf	Fließkommazahl mit n, m	float, double
%n.ms	m Zeichen eines Strings der Breite n	char*
%p	Pointer-Adresse	void*

Beispiel sprintf

```c
char s[] = "Text1";
char t[] = "Text2";
char buffer[255];

sprintf(buffer, "%10s %-10s %s", s, t, "Huhu");
printf("%s\n", buffer);

sprintf(buffer, "%10.2f", 21.12345678);
printf("%s\n", buffer);
```

Ausgabe

```
     Text1 Text2      Huhu
     21.12
```

Der POSIX-Standard erweitert die Format-Strings um ein sogenanntes Parameter-Feld, mit dem man die jeweiligen Parameter direkt über ihren Index (beginnend bei 1) adressieren kann.

Da POSIX sich nur auf Unix-Betriebssysteme bezieht, findet man diese Erweiterung unter Windows in einer speziellen Funktion `printf_p`.

POSIX-Erweiterung: **Parameter-Feld** Parameter-Feld

- Syntax: `%n$[width][.precision][length]type`
- adressiert den Parameter mit dem Index `n`, beginnend bei 1
- unter Windows `printf_p`-Funktion

```
printf("%2$d %1$d %2$d %1$d", 1, 2);
```

Ausgabe

```
2 1 2 1
```

Anzahl der geschriebenen Zeichen

- Syntax: `%n`
- Schreibt die Anzahl der bisher ausgegebenen Zeichen in die Variable

```
int count;
printf("012345678%n", &count);
printf(" -> %d", count);
```

Ausgabe

```
012345678 -> 9
```

Achtung: Öffnet das Tor für diverse Angriffe

Die Möglichkeit, durch das `%n`-Formatzeichen eine Schreiboperation auszulösen, eröffnet die Möglichkeit über Fehler im Formatstring erhebliche Sicherheitslücken aufzureißen. Außerdem macht es den Format-String von `printf` zu einer vollständigen Programmiersprache (turing-vollständig).

Der Gewinner des International Obfuscated C Code Contest 2020 hat ein Tic-Tac-Toe nur als Formatstring unter Verwendung von `%n` implementiert.

Gewinner des International Obfuscated C Code Contest 2020

```
#include <stdio.h>
```

```
#define N(a)      "%"#a"$hhn"
#define O(a,b)    "%10$"#a"d"N(b)
#define U         "%10$.*37$d"
#define G(a)      "%"#a"$s"
#define H(a,b)    G(a)G(b)
#define T(a)      a a
#define s(a)      T(a)T(a)
#define A(a)      s(a)T(a)a
#define n(a)      A(a)a
#define D(a)      n(a)A(a)
#define C(a)      D(a)a
#define R         C(C(N(12)G(12)))
#define o(a,b,c)  C(H(a,a))D(G(a))C(H(b,b)G(b))n(G(b))O(32,c)R
#define SS        O(78,55)R "\n\033[2J\n%26$s";
#define E(a,b,c,d) H(a,b)G(c)O(253,11)R G(11)O(255,11)R H(11,d)N(d)O(253,35)R
#define S(a,b)    O(254,11)H(a,b)N(68)R G(68)O(255,68)N(12)H(12,68)G(67)N(67)

char* fmt = O(10,39)N(40)N(41)N(42)N(43)N(66)N(69)N(24)O(22,65)O(5,70)O(8,44)N(
           45)N(46)N   (47)N(48)N(   49)N( 50)N(    51)N(52)N(53   )O( 28,
           54)O(5,       55) O(2,   56)O(3,57)O(     4,58 )O(13,   73)O(4,
           71 )N(  72)O   (20,59    )N(60)N(61)N(     62)N (63)N   (64)R R
           E(1,2,   3,13   )E(4,    5,6,13)E(7,8,9      ,13)E(1,4   ,7,13)E
           (2,5,8,       13)E(    3,6,9,13)E(1,5,       9,13)E(3   ,5,7,13
           )E(14,15,   16,23)   E(17,18,19,23)E(        20, 21,   22,23)E
           (14,17,20,23)E(15,  18,21,23)E(16,19,  22     ,23)E(   14, 18,
           22,23)E(16,18,20,  23)R U O(255 ,38)R  G (     38)O(   255,36)
           R H(13,23)O(255,   11)R H(11,36) O(254  ,36)    R G(    36 ) O(
           255,36)R S(1,14   )S(2,15)S(3, 16)S(4,  17 )S     (5,    18)S(6,
           19)S(7,20)S(8,   21)S(9    ,22)H(13,23   )H(36,    67    )N(11)R
           G(11)""O(255,   25 )R       s(C(G(11)   ))n (G(         11) )G(
           11)N(54)R C(   "aa")   s(A(   G(25)))T   (G(25))N        (69)R o
           (14,1,26)o(   15, 2,   27)o   (16,3,28   )o( 17,4,       29)o(18
           ,5,30)o(19    ,6,31)o(       20,7,32)o   (21,8,33)o      (22 ,9,
           34)n(C(U)    )N( 68)R H(   36,13)G(23)   N(11)R C(D(    G(11)))
           D(G(11))G(68)N(68)R G(68)O(49,35)R H(13,23)G(67)N(11)R C(H(11,11)G(
           11))A(G(11))C(H(36,36)G(36))s(G(36))O(32,58)R C(D(G(36)))A(G(36))SS

#define arg d+6,d+8,d+10,d+12,d+14,d+16,d+18,d+20,d+22,0,d+46,d+52,d+48,d+24,d\
           +26,d+28,d+30,d+32,d+34,d+36,d+38,d+40,d+50,(scanf(d+126,d+4),d+(6\
           -2)+18*(1-d[2]%2)+d[4]*2),d,d+66,d+68,d+70, d+78,d+80,d+82,d+90,d+\
           92,d+94,d+97,d+54,d[2],d+2,d+71,d+77,d+83,d+89,d+95,d+72,d+73,d+74\
           ,d+75,d+76,d+84,d+85,d+86,d+87,d+88,d+100,d+101,d+96,d+102,d+99,d+\
           67,d+69,d+79,d+81,d+91,d+93,d+98,d+103,d+58,d+60,d+98,d+126,d+127,\
           d+128,d+129

char d[538] = {1,0,10,0,10};

int main(int argc, char** argv) {
    while(*d) printf(fmt, arg);
}
```

5.7 Sichere String-Funktionen [95]

Stringverarbeitung sollte nur mit Funktionen erfolgen, welche die Länge des Zielpuffers prüfen, wie z. B. `snprintf`, anstelle von `sprintf`. Diese **sicheren String-Funktionen** bieten einen Schutz vor Pufferüberläufen, die auftreten können, wenn der Ausgabepuffer nicht groß genug ist, um den gesamten generierten String aufzunehmen. Funktionen wie `snprintf` stellen sicher, dass der generierte String in den angegebenen Puffer passt und keinen Speicher außerhalb des Puffers überschreibt.

sicheren String-Funktionen

Man sollte nur sichere String-Funktionen verwenden

- `strcpy` → `strlcpy` — strlcpy
- `strcat` → `strncat` — strncat
- `strcmp` → `strncmp` — strncmp
- `sprintf` → `snprintf` — snprintf
- `strlen` → `strnlen` — strnlen

Tatsächlich ist die sichere Variante von `strcpy` nicht `strncpy`, sondern `strlcpy`. Der Grund liegt darin, dass `strncpy` Strings erzeugt, die nicht nullterminiert sind, wenn der Eingabestring `n` oder mehr Zeichen enthält. Damit laufen dann andere Funktionen, wie z. B. `strlen` über das Ende des Strings hinaus, weil der Nullterminator fehlt.

Kapitel 6

Struct und Union

6.1 Video zum Kapitel [97]

Link zu YouTube

6.2 Strukturen [98]

Mit Arrays kann ich nur Daten eines Typs speichern und über einen Index darauf zugreifen. Wenn es darum geht, verschiedene Daten in einer strukturierten Form abzulegen oder aber über einen Namen zu adressieren, bieten sich in C der Datentyp `struct` an.

`struct`

Strukturen

- **Strukturen** (**structs**) sind ähnlich zu Java Objekten (nur ohne Methoden)
- jeder andere Typ kann *Komponente* einer Struktur sein
- Zugriff mit `struct.field`
- Werden auf dem Stack angelegt (↔ Java-Klassen: Heap)

```
struct { int x; int y; float w; } rect;

rect.x = 10;
rect.y = 20;
rect.w = 2.5;
```

Strukturen, die wie im Beispiel oben, deklariert wurde, werden – anders als in Java – auf dem Stack abgelegt und sind somit nur in der aktuellen Funktion benutzbar. Will man die Struktur auf dem Heap ablegen, muss man sie dynamisch erzeugen (siehe unten). Der Ausdruck

`struct { ... } NAME;` alloziert bereits den Speicher auf dem Stack, ist also nicht nur eine Deklaration, sondern auch direkt eine Definition der Variable `NAME`.

Wenn man von Java kommt, kann man sich Strukturen wie Java-Klassen vorstellen, bei denen alle Attribute `public` sind und keine Methoden existieren.

Beispiel in Java

```
// Dies ist nur eine Deklaration!
class Rect {
    int x;
    int y;
    float w;
}

class Main {
    public static void main(String args[]) {
      // Hier wird das Objekt erst angelegt.
      Rect rect = new Rect();

      rect.x = 10;
      rect.y = 20;
      rect.w = 2.5f;
    }
}
```

Die syntaktische Besonderheit, dass bei einer Struktur in C die Deklaration und Definition direkt zusammenfallen passt nicht zu modernen Programmieransätzen.

Deswegen kann man eine Struktur in C deklarieren und dann an späteren Stellen wieder referenzieren, ohne direkt eine Variable anzulegen. Dies erfolgt mit der Syntax `struct NAME {...};`. `NAME` ist in diesem Fall keine Variable, sondern ein sogenannter **Tag**, der die Struktur bezeichnet. Bei der Variablendeklaration muss man allerdings das `struct` wiederholen. Einen Ausweg schafft `typedef`.

Tag

Will man eine **Struktur-Definition** wiederverwenden

Struktur-Definition

- Struktur benennen → `struct` muss in der Variablendeklaration wiederholt werden
- `typedef` verwenden → `struct` muss nicht wiederholt werden

```
struct Complex { double real; double imag; };
struct Point { double x; double y; } corner;
struct Point p;
p.x = 7.0; corner.x = 9.0;
```

Diese Variante ist nicht besonders modern und man sollte heutzutage grundsätzlich auf ein `typedef` zurückgreifen.

```
typedef struct { double real; double imag; } Complex;
typedef struct { double x; double y; } Point;
Point p;
Complex z;
p.x = 7.0; z.real = 9.8;
```

Beachten Sie, dass beim `typedef` der Name der Struktur *hinten* steht, bei der Variante ohne `typedef` aber *vorne*.

Tags bilden einen eigenen Namensraum, der vom Namensraum der Variablen getrennt ist. Im Beispiel `struct S { int a };` heißt der Datentyp der Struktur nicht `S`, sondern `struct S`. Der Name `S` kann für Variablen, Enumerationen oder Unions wiederverwendet werden.

Initialisierung von Strukturen

Für die **Initialisierung von Strukturen** gibt es eine verkürzte Syntax, welche die Initialisierung direkt bei der Definition durchführt.

Initialisierung von Strukturen

```
typedef struct { double x; double y; } Point;

Point p1;
p1.x = 4.2;
p1.y = 2.3;

Point p2 = { 4.2, 2.3 };

Point p3 = { .y = 2.3, .x = 4.2 };

Point p4 = { .x = p3.x, .y = p3.y };
```

Wenn man die Elemente der Struktur bei der Initialisierung benennt, dann ist die Reihenfolge egal und er Code ist robuster gegenüber späteren Erweiterungen der Struktur.

Die Werte bei der Initialisierung der Struktur müssen keine Konstanten sein.

Da Strukturen sehr häufig dynamisch alloziert werden, verwaltet man sie über Pointer. Die Syntax für das Dereferenzieren der Pointer ist aber im Zusammenhang mit Strukturen sehr sperrig `(*struct_pointer).element`. Deswegen gibt es eine syntaktische Vereinfachung.

- Strukturen werden häufig *mit Pointern verwendet* (dynamische Erzeugung mit `malloc`)
- Kurzform für das Dereferenzieren des Pointers mit ->

Dereferenzieren einer Struktur

```
(*sp).element = 42;
y = (*sp).element;
```

Verwendung von ->

```
sp->element = 42;
y = sp->element;
```

6.3 Größe einer Struktur [102]

Für die dynamische Speicherverwaltung ist es wichtig zu wissen, wie viel Speicher eine Struktur verbraucht. Eine Struktur ist natürlich mindestens so groß, wie sie Summe der darin benutzten Datentypen. Allerdings kann sie auch mehr Speicher verbrauchen, und zwar dann, wenn der Compiler **Padding** einfügt.

Padding

- Die **Größe einer Struktur** ergibt sich aus der Größe der Elemente + Padding für Alignment der Adressen

Größe einer Struktur

```
struct {
   char x;
   int y;
   char z;
} s1;

printf("%ld\n", sizeof(s1)); /* -> 12 */

struct {
   char x, z;
   int y;
 } s2;

printf("%ld\n", sizeof(s2)); /* -> 8 */
```

Wie man sieht, verbrauchen die Strukturen unterschiedlich viel Speicher, obwohl sie genau dieselben Daten tragen. Der Grund ist das Padding, das der Compiler eingefügt hat, um die Adressen der Objekte zu **alignen** (auszurichten). Damit ist gemeint, dass die Adressen ganzzahlige Vielfache von einer Konstante (normalerweise 2, 4 oder 8) sind. Das **Alignment** beschleunigt den Zugriff auf die Daten durch den Prozessor, weil dieser nur ausgerichtete Adressen mit maximaler Geschwindigkeit lesen kann.

alignen

Alignment

Im Beispiel scheint der Compiler ein Alignment auf Adressen durchzuführen, die durch vier teilbar sind. Damit sieht die erste Struktur (aus Sicht des Compilers) wie folgte aus:

```
struct {
   char x;         /* Offset: 0 */
   char _padding[3]; /* Offset: 1 */
   int y;          /* Offset: 4 */
```

```
   char z;         /* Offset: 8 */
   char _padding[3]; /* Offset: 9 */
} s1;
```

Das Padding am Ende der Struktur dient dazu, dass die Gesamtgröße der Struktur ebenfalls ein Vielfaches des Alignments ist. Ohne das Padding wäre die Struktur 9 Bytes groß, sodass sie auf 12 Bytes aufgefüllt wird, damit ihre Größe wieder durch vier teilbar wird.

Bei der zweiten Struktur ist weniger Padding erforderlich.

```
struct {
   char x;         /* Offset: 0 */
   char z;         /* Offset: 1 */
   char _padding[2]; /* Offset: 2 */
   int y;          /* Offset: 4 */
 } s2;
```

6.4 Struktur dynamisch anlegen [103]

Ebenso wie Arrays möchte man Strukturen dynamisch verwalten können. Dies erfolgt dann analog mit `malloc()` und `free()`.

```
typedef struct { int x; int y; } Point2D;

/* Speicher allozieren */
Point2D *p = (Point2D*) malloc(sizeof(Point2D));

p->x = 12;
p->y = 23;

printf("x=%d, y=%d\n", p->x, p->y); /* x=12, y=23 */

/* Speicher freigeben */
free(p);
```

Die Verwendung des ->-Operators erspart einem beim Umgang mit Pointer auf Strukturen das ständige dereferenzieren mit *.

6.5 Unions [104]

Ein absoluter Exot unter den Datentypen ist die Union, die man aus Java und anderen Sprachen nicht kennt. Laut den C-Erfindern benutzt man sie dann, wenn man eine Variable benötigt,

die zu unterschiedlichen Zeiten unterschiedliche Typen aufnehmen soll aber immer dieselbe Größe haben soll.

Unions

- **Unions** sind ähnlich zu Strukturen, speichern aber immer nur einen Wert
- Die Größe entspricht dem größten Datentyp

```c
union u_tag {
    int ival;
    float fval;
    char *sval;
} u;
printf("sizeof=%ld\n", sizeof(u)); /* sizeof=8 */

u.ival = 12;
printf("u.ival=%d\n", u.ival); /* u.ival=12 */

u.fval = 8.0;
printf("u.fval=%f\n", u.fval); /* u.fval=8.0 */
printf("u.ival=%d\n", u.ival); /* u.ival=1090519040 */
```

Die Größe der Union im Beispiel ergibt sich aus dem größten Datentyp, der in diesem Fall `char*` ist, weil ein Pointer auf einer 64-Bit-Plattform 8 Byte benötigt.

Die letzte Zeile im Beispiel zeigt, das man wissen muss, welche Daten man in die Union geschrieben hat, da ein Zugriff über den falschen Typ zu seltsamen Ergebnissen führen kann. Das Ergebnis ergibt sich aus dem Bitmuster, die der `float`-Wert `8.0` hat, wenn man es als Integer interpretiert.

Da wir heute meistens genügend Speicher zur Verfügung haben, ist der Spareffekt der Union nicht mehr relevant.

6.6 Bitfelder [105]

Mit Bitfeldern kann man in C Variablen definieren, die eine bestimmte Anzahl von Bits haben. Die einzelnen Bits können benannt werden.

Bitfeld

- Ein **Bitfeld** (**bitfield**) ist eine Integer-Variable mit einer bestimmten Anzahl von Bits
  Syntax: `TYP [NAME] : BREITE`
- Bits können über Namen angesprochen werden
- Der Compiler packt die Daten aufeinanderfolgender Bitfelder eng zusammen

```c
struct Date {
    unsigned int day : 5;
    unsigned int month : 4;
    signed int year : 22;
    char isDST : 1;
```

```
};
```

Dieses Beispiel zeigte die Verwendung eines 32 Bit breiten Bitfeldes, um das aktuelle Datum in einem einzigen Integer zu speichern.

```c
#include <stdio.h>

typedef struct {
    unsigned int day : 5;
    unsigned int month : 4;
    signed int year : 22;
    char isDST : 1;
} Date;

int main(int argc, char** argv) {
    Date myDate = { 24, 12, 2017, 0 };
    printf("%ld\n", sizeof(myDate)); /* -> 4 */

    printf("%d-%d-%d\n", myDate.year, myDate.month, myDate.day);
    /* -> 2017-12-24 */
}
```

6.7 Bitmasken [107]

Bitfelder vereinfachen den Zugriff auf die einzelnen Bits eines Wertes und werden z. B. eingesetzt, um Protokolle zu implementieren. Eine Alternative dazu ist, die Bits selbst zu setzen und zu verwalten.

Alternativ zu Bitfeldern kann man die Daten auch selbst packen und entpacken

```c
#include <stdio.h>
#include <stdint.h>

#define DAY_SHIFT 27
#define MONTH_SHIFT 23
#define YEAR_SHIFT 1
#define DAY_MASK 0x1f
#define MONTH_MASK 0x0f
#define YEAR_MASK 0x3fffff
#define DST_MASK 0x01

uint32_t encode(int day, int month, int year) {
    return (((day & DAY_MASK) << DAY_SHIFT)
         | ((month & MONTH_MASK) << MONTH_SHIFT)
         | ((year & YEAR_MASK) << YEAR_SHIFT));
}
```

```c
void decode(uint32_t date, int *day, int *month, int *year) {
    *day = (date >> DAY_SHIFT) & DAY_MASK;
    *month = (date >> MONTH_SHIFT) & MONTH_MASK;
    *year = (date >> YEAR_SHIFT) & YEAR_MASK;
}

int main(int argc, char** argv) {
    uint32_t myDate;
    int day, month, year;

    myDate = encode(24, 12, 2017);
    printf("%u\n", myDate); /* 3321892802 */

    myDate = 763367326;
    decode(myDate, &day, &month, &year);
    printf("%d-%d-%d\n", year, month, day); /* 1999-11-5 */

}
```

Index

Wintersemester 2025/2026

C-Programmierung

Technische Hochschule Mannheim

Fortgeschrittene Themen

Prof. Thomas Smits

Stand 2025-07-21

Technische Hochschule mannheim

Inhaltsverzeichnis

Kapitel 1

Fehlerbehandlung

1.1 Video zum Kapitel [5]

Link zu YouTube

1.2 C und Fehlerbehandlung [6]

Wer bisher Java programmiert hat, ist es gewohnt, eine *strukturierte Fehlerbehandlung* zu bekommen, d. h. ein von der Sprache unterstütztes Konzept, um mit Fehlern umzugehen und sie zu behandeln. C++ und Java bieten **Exceptions** zur Fehlerbehandlung an. C hat aufgrund seines Alters kein entsprechendes Konzept.

Exceptions

- Das *Konzept zur Fehlerbehandlung* in C lässt sich kurz zusammenfassen:
  Es gibt kein Konzept
- Fehler werden durch *spezielle Rückgabewerte* der Funktionen signalisiert
- Aufrufer muss auf den Rückgabewert *prüfen* und *reagieren*
  - Fehler behandeln (Fehlerbehandlungsroutine anspringen → goto)
  - Fehler *ignorieren*
  - Fehler selbst über einen Rückgabewert *weitergeben*
- Zusätzlich wird eine globale Variable `errno` gesetzt

errno

Während man bei C die fehlende Fehlerbehandlung auf das Alter schieben kann, stellt sich die Frage, warum eine der modernsten Sprachen, nämlich *Go* (2007 erschienen), ebenfalls kein Konzept für Ausnahmen hat. Tatsächlich ist es so, dass Ausnahmen ein durchaus umstrittenes Konzept sind und nicht von allen Entwicklerinnen als die einzige Lösung für das Signalisieren

von Problemen angesehen wird. Die Entwickler von Go haben sich entschieden, dass Funktionen beliebig viele Rückgabewerte haben können und dass einer davon zum Anzeigen von Fehlern verwendet wird. Sie empfanden Ausnahmen als zu schwergewichtig und fürchteten, dass es sonst wie bei Java zu einer Flut von Ausnahmen kommt, die niemand mehr behandeln kann und will.

Der folgende Code zeigt ein Beispiel dafür, wie in C mit Fehlern umgegangen wird.

```c
if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
     DieWithError("socket () failed") ;
}
/* Bind to the local address */
if (bind(servSock, (struct sockaddr *)&echoServAddr,
        sizeof(echoServAddr)) < 0) {
    DieWithError ("bind () failed");
}
/* Mark the socket so it will listen for incoming connections */
if (listen(servSock, MAXPENDING) < 0) {
    DieWithError("listen() failed") ;
}
for (;;) { /* Run forever */
    if ((clntSock = accept(servSock, (struct sockaddr *) &echoClntAddr,
            &clntLen)) < 0) { /* Wait for a client to connect */
        DieWithError("accept() failed");
    }
}
```

Man sieht hier an dem C-Beispiel sehr deutlich die Probleme klassischer Fehlerbehandlung über Rückgabewerte:

- Die normale Programmlogik und die Fehlerbehandlung sind wild gemischt.
- Der Programmcode wird erheblich aufgebläht und sehr unübersichtlich, da manche IFs der Fehlerbehandlung dienen und andere wieder dem normalen Kontrollfluss.
- Der Rückgabewert von Funktionen ist doppeldeutig: Er transportiert sowohl Fehler- als auch normale Informationen.
- Es kann leicht passieren, dass man vergisst einen Fehlercode abzufragen, sodass Fehler überhaupt nicht behandelt werden.
- Es ist sehr schwer Fehler weiterzureichen, da die hier dargestellte Funktion vier verschiedene Fehlersituationen im eigenen Rückgabewert codieren müsste.

Es gibt einige allgemeine Konventionen, die man in C-Programmen bezüglich der Fehlerbehandlung antrifft.

- Funktionen, die einen *Pointer zurückgeben* (z. B. `malloc`)
  ⇒ Fehler wird durch `NULL` angezeigt
  ⇒ Fehlercode muss aus `errno` gelesen werden

- Funktionen, die *positive Werte zurückgeben* (z. B. open)
  ⇒ Fehlercode wird durch -1 angezeigt
  ⇒ Fehlercode muss aus `errno` gelesen werden
- Funktionen, die *normalerweise nichts zurückgeben* müssen (z. B. `pthread_create`)
  ⇒ Fehlercode *direkt als Rückgabewert* geliefert, `0` heißt alles OK
- `char* strerror(int errnum)` aus `<string.h>` `strerror(int errnum)`
  ⇒ Fehlernummern zu Fehlermeldung
- `void perror(const char *s)` `perror(const char *s)`
  ⇒ gibt den Text `s` und danach die Fehlerbeschreibung aus

```c
FILE* fh = fopen("/gibtsnicht");
if (!fh) {
    perror("Fehler beim Öffnen der Datei");
    exit(1);
}
```

Ausgabe

```
Fehler beim Öffnen der Datei: No such file or directory
```

1.3 Fehlernummern und Fehlerausgabe [9]

- Die **Fehlernummern** werden über `#define` definiert und sind spezifisch für die aufgerufenen Funktion (siehe `<errno.h>`) **Fehlernummern**
  - `#define EPERM 1 /* Operation not permitted */`
  - `#define ENOENT 2 /* No such file or directory */`
  - `#define ESRCH 3 /* No such process */`
  - `#define EINTR 4 /* Interrupted system call */`
  - `#define EIO 5 /* Input/output error */`
  - `#define ENXIO 6 /* Device not configured */`
  - ...

Beispiel für die Ausgabe eines Fehlers

```c
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char** argv) {
    FILE* f;
    if ((f = fopen("test", "r")) == NULL) {
        printf("%s\n", strerror(errno));
```

```
        exit(1);
    }
}
```

Ausgabe

```
No such file or directory
```

Durch Verwendung von `strerror` wird der Fehlercode, der beim Versuch die nicht-existente Datei zu öffnen, gesetzt wurde in den Text „No such file or directory“ übersetzt.

1.4 Beispiel: Fehler weitergeben [11]

```c
#define EFOPEN 1

FILE* fd = NULL;

int openFile(const char* file) {
    fd = fopen(file, "r");
    if (fd == NULL) {
        return EFOPEN;
    }
    else {
      return 0;
    }
}
```

Dieses C-Programm zeigt, wie man Fehler weitergeben kann. Es definiert eine Konstante `EFOPEN` und eine Funktion `openFile`, die versucht, eine Datei zu öffnen. Die Konstante namens `EFOPEN` wird verwendet, um einen Fehlercode zu repräsentieren, falls die Datei nicht geöffnet werden kann.

Die Funktion `openFile` versucht, eine Datei im Lesemodus (`"r"`) zu öffnen, wobei der Dateiname als Argument übergeben wird (`file`). Wenn `fd` `NULL` ist (d. h., die Datei konnte nicht geöffnet werden), gibt die Funktion `EFOPEN` zurück, um einen Fehler anzuzeigen. Falls die Datei erfolgreich geöffnet wird, gibt die Funktion `0` zurück, um anzuzeigen, dass keine Fehler aufgetreten sind.

Kapitel 2

Undefined Behavior

2.1 Besonderheiten von C [13]

C hat im Gegensatz zu anderen Programmiersprachen die Besonderheit, dass der Standard gewisse Fälle des Verhaltens von Programmen nicht abdeckt. Während also in Java für alle syntaktisch korrekten Statements das Ergebnis eindeutig definiert ist, lässt der C-Standard zu, dass ein Statement zwar syntaktisch korrekt ist, das Verhalten aber nicht festgelegt, die Semantik also *undefiniert* ist.

C-Standard unterscheidet

- **implementation-defined behavior**: Eigenschaften, die von der Plattform und dem Compiler festgelegt werden können, z. B. `sizeof(int)` — implementation-defined behavior
- **unspecified behavior**: Verhalten, zu dem der Standard keine Aussage macht oder bei dem verschiedene Alternativen bestehen, z. B. Auswertungsreihenfolge von Funktionsparametern. — unspecified behavior
- **undefined behavior**: Der Standard definiert nicht, was bei einem entsprechenden Fehler passieren soll, z. B. Dereferenzieren des Null-Pointers — undefined behavior

Nachdem ein Statement mit undefined behavior aufgetreten ist, kann *alles passieren*

Unspecified behavior

Ein Beispiel für *unspecified behavior* ist die Reihenfolge, in der Funktionsparameter ausgewertet werden:

Unspecified Behavior

```c
#include <stdio.h>

void print_it(int a, int b, int c) {
    printf("%d, %d, %d\n", a, b, c);
}
int main(int argc, char** argv) {
    int x = 0;
    print_it(x++, x++, x++);
```

```
}
```

```
2, 1, 0
```

Unspecified behavior führt nicht zum Programmabsturz oder zu Folgefehlern aber der C-Code ist nicht mehr portabel und könnte sich auf einem anderen System oder bei einem anderen Compiler anders verhalten.

Undefined behavior

Nachdem ein Statement mit undefined behavior ausgeführt wurde, kann *alles passieren*, d. h. über den weiteren Programmablauf können keine Annahmen mehr getroffen werden. Der Compiler darf davon ausgehen, dass *undefined behavior* im Programm nicht vorkommt und kann den generierten Code entsprechend optimieren. Als Konsequenz können beliebige Fehler und Sicherheitslücken aus undefined behavior entstehen.

Undefined Behavior

```
char a[10];
a[12] = 'x';
```

Der C11-Standard definiert 199 Fälle von undefined behavior.

2.2 Beispiele für undefined behavior [14]

- Zugriff auf eine nicht initialisierte Variable
  `int c; c++;`
- Zugriff auf einen Null-Pointer
  `int *x = NULL; *x++;`
- Zugriff hinter Array-Grenze
  `char c[10]; c[12] = 'a';`
- Verstoß gegen strict aliasing rule
  `int i = 12; float *fp = (float*)&ip; *fp += 0.4;`
- Signed-Integer-Overflow
  `INT_MAX + 1`
- Zu große Shift-Operationen
  `uint32_t x = 1 << 32;`
- Negative Shift-Operation
  `uint32_t x = 1 << -1`
- Division durch 0
  `int i = 5 / 0;`
  `int i = 5 % 0;`

- Modulo einer negativen Zahl
  `int i = INT_MIN % -1;`
- Schreiben in String-Literal
  `char* t = "Hallo"; t[0] = 'h';`
- non-`void` Funktion ohne `return`
  `int f(){}`
- Endlosschleife ohne Seiteneffekte
  `for (;;){ int i = 0; }`
- Die `main`-Funktion direkt aufrufen
  `void f(){ main(); }`
- `free` mit Pointer aufrufen, der nicht von `malloc` kam
  `int i = 9; free(&i);`
- `memcpy` mit überlappenden Bereichen
  `char b[20]; memcpy(b, b + 10, 30);`
- `FILE`-Pointer verwenden, nachdem die Datei geschlossen wurde

Während der Overflow eines vorzeichenbehafteten Integer-Wertes in C undefiniert ist, sind die Overflows für `unsigned int` wohldefiniert und dürfen benutzt werden.

Korrekter Code

```
unsigned int i = UINT_MAX;
i++; /* 0 */

unsigned int k = 0;
k--; /* 4294967295 */
```

2.3 Gründe für undefined behavior [17]

Gründe für undefined behavior

- Schnellerer Code
- Einfachere Compiler-Implementierung
- Kompatibilität über Plattformen hinweg

Der Preis, den man für das Konzept des undefined behavior ist allerdings, dass in C-Programmen bereits einfache Fehler, wie ein Overflow bei einem `int` dazu führen können, dass erhebliche Sicherheitslücken entstehen.

Kapitel 3

Input/Output mit stdio.h

3.1 Video zum Kapitel [19]

Link zu YouTube

3.2 stdio-Bibliothek [20]

In C sind die Eingabe- und Ausgabeoperationen nicht Teil der Sprache, sondern werden durch eine Bibliothek zur Verfügung gestellt. Da diese Bibliothek im C-Standard definiert wird, sollte sie auf allen Plattformen und bei allen C-Compilern zur Verfügung stehen. Ausnahmen sind eingebettete Systeme, die kein Input/Output unterstützen.

Dreh und Angelpunkt der I/O in C ist die *stdio*-Bibliothek, die eine ganze Reihe von Funktionen anbietet.

- Eingabe/Ausgabe wird über die `stdio`-Bibliothek realisiert
  - `#include <stdio.h>`
  - wird automatisch gelinkt
- Definiert den Typ `FILE*` und Funktionen für Dateizugriff
- Definiert `stdin`, `stdout`, `stderr`

`stdio`

`stdin`, `stdout` und `stderr` sind vordefinierte `FILE`-Objekte, die den Zugriff auf die Standard-Ein- und -Ausgabe ermöglichen.

`stdin`
`stdout`
`stderr`

Neben den Funktionen in `stdio.h`, die im C-Standard definiert werden, gibt es eine Reihe von korrespondierenden **Low-Level-I/O-Funktionen**, die vom POSIX-Standard festgelegt werden. Diese Funktionen sind näher am Betriebssystem und bieten weniger Komfort. Außerdem sind

Low-Level-I/O-Funktionen

sie nur auf Systemen verfügbar, die den POSIX-Standard unterstützen, wozu allerdings die relevanten Betriebssysteme gehören.

Beispiele Low-Level-Funktionen sind:

- `int open(const char *pathname, int flags, mode_t mode);`
- `int creat(const char *pathname, mode_t mode);`
- `int close(int fd);`

Ken Thompson, einer der Erfinder von Unix, wurde einmal gefragt, was er heute anders machen würde, wenn er Unix noch einmal entwickeln dürfte. Seine Antwort war „I'd spell creat with an e."

Die Low-Level-Funktionen werden weiter unten diskutiert.

3.3 Dateien öffnen und schließen [21]

fopen

Bevor man eine Datei lesen (oder schreiben) kann, muss man sie mit `fopen` öffnen. Als Rückgabewert bekommt man einen Pointer auf ein `FILE`-Objekt zurück. Der genaue Inhalt dieses Objektes ist irrelevant, denn es dient ausschließlich dazu, bei den Funktionsaufrufen zu übergeben, auf welche Datei man sich gerade beziehen möchte.

`FILE *fopen(const char *path, const char *mode)`

- öffnet die Datei `path`
- Modus (`mode`)
  - `"r"`: Lesen
  - `"r+"`: Lesen und schreiben (ab Anfang)
  - `"w"`: Auf 0 Byte kürzen schreiben (ab Anfang)
  - `"w+"`: Auf 0 Byte kürzen, lesen und schreiben (ab Anfang)
  - `"a"`: Schreiben (am Ende anhängen)
  - `"a+"`: Lesen und schreiben (am Ende anhängen)
- bei Erfolg wird ein `FILE*` zurückgegeben, im Fehlerfall `NULL`

fclose

Eine Datei, die man einmal geöffnet hat, muss man nach der Verwendung wieder mit `fclose` schließen. Ein Prozess kann nur eine bestimmte Anzahl von Dateien geöffnet halten und es kommt zu Fehlern, wenn er diese Grenze überschreitet. Deswegen ist ein sorgfältiger Umgang mit den Ressourcen wichtig und was geöffnet wurde, sollte möglichst bald auch wieder geschlossen werden.

Wenn ein Programm beendet wird, werden zwar automatisch alle geöffneten Dateien geschlossen – es ist aber schlechter Stil, sich hierauf zu verlassen.

`int fclose(FILE *stream)`

- schließt die Datei `stream`
- `0` bei Erfolg, `EOF` im Fehlerfall

```c
FILE* handle = fopen("/tmp/file", "r");

if (handle == NULL) {
  /* Fehler */
  exit(1);
}

/* Mit Datei arbeiten */

if (fclose(handle)) {
  /* Fehler beim Schließen */
}
```

Ein Fehler beim Schließen einer Datei ist selten, aber wenn er auftritt auch nur schwer zu behandeln, deswegen wird man normalerweise nicht versuchen, das Schließen zu wiederholen.

Die Standardbibliothek spricht hier von **Streams** und nicht *Dateien*, weil die Methoden auch auf andere Arten von I/O angewendet werden können, z. B. die Console, die keine Datei ist.

Streams

3.4 Lesen und Schreiben von Binärdaten [23]

Wenn man eine Datei erfolgreich geöffnet hat, kann man auf die Daten in der Datei zugreifen. Ob man nur lesen, nur schreiben oder beides kann hängt davon ab, in welchem Modus man die Datei bei `fopen` geöffnet hat.

Verarbeitet man binäre Daten, werden folgende Funktionen eingesetzt:

- `size_t fread(void *buf, size_t size, size_t n, FILE *stream)`
- `size_t fwrite(const void *buf, size_t size, size_t n, FILE *stream)`

Hier ist

- `size` die Größe des verarbeiteten Objektes (`sizeof(x)`)
- `buf` der Puffer der geschrieben/in den gelesen wird
- `n` die Anzahl der verarbeiteten Objekte der Größe `size`
- Anzahl der gelesenen/geschriebenen Objekte wird zurückgegeben

`fread` und `fwrite` basieren auf der Vorstellung, dass man nicht einfach nur Bytes schreibt, sondern auch andere Daten, z. B. `int`-Werte oder auch Inhalte von Strukturen. Deswegen gibt man auch nicht an, dass man `x`-Bytes schreibt, sondern `n` Elemente der Größe `size`.

fread
fwrite

Teilweise will man sich nicht nur linear durch die Datei bewegen, sondern auch springen können. Hierzu dient die Funktion `fseek`, mit der man die Position für die nächste Lese- oder Schreiboperation verschieben kann. Hierbei kann man vom Anfang, vom Ende oder von der aktuellen Position aus das Ziel angeben.

fseek

- `int fseek(FILE *stream, long offset, int whence)`
  Verschiebt die aktuelle Position in der Datei
  Mögliche Werte für `whence`
  - `SEEK_SET`: vom Anfang aus
  - `SEEK_CUR`: von der aktuellen Position aus
  - `SEEK_END`: vom Ende aus
- `long ftell(FILE *stream)`
  Liefert die aktuelle Position in der Datei

Mit `ftell` kann man die aktuelle Position in der Datei auslesen. `ftell`

Beispiel: Daten schreiben

```c
/*  Datei zum Schreiben im Binärmodus öffnen */
FILE *fh = fopen("/tmp/data", "w+");

/* Fehlerbehandlung */
if (!fh) {
    perror("Datei: ");
    exit(1);
}

unsigned int magic_value = 0xcafebabe;
fwrite(&magic_value, sizeof(int), 1, fh);

/* An den Anfang springen und Daten lesen */
fseek(fh, 0, SEEK_SET);
unsigned int read_back;
fread(&read_back, sizeof(int), 1, fh);
printf("Aus Datei: %x\n", read_back);
```

Das Beispiel zeigt, wie eine Datei `/tmp/data` zum Lesen und Schreiben geöffnet und dann der Integer-Wert `0xcafebabe` hineingeschrieben wird. Danach wird der Dateizeiger wieder an den Anfang der Datei geschoben und die Daten werden zurück gelesen und ausgegeben.

3.5 Zeichen-I/O [26]

Funktionen für den Zugriff auf einzelne Zeichen. Alle Funktionen geben `EOF` zurück, wenn Datei/Stream zu Ende ist oder ein Fehler auftritt

- `int getchar()`
  Liest das nächste Zeichen von `stdin`
- `int fgetc(FILE *in)`
  List das nächste Zeichen aus Datei `in`
- `int putchar(int c)`
  Schreibt ein Zeichen auf `stdout`

- `int fputc(int c, FILE *out)`
  Schreibt ein Zeichen in Datei `out`

Bemerkenswert ist aber die Tatsache, dass die Funktion `getchar()` ein `int` als Rückgabetyp hat und kein `char`, obwohl ein Stream byteweise gelesen wird. Der größere Datentyp ist nötig, da durch ein `EOF` signalisiert wird, dass der Stream zu Ende ist. `EOF` hat den Wert –1. Da aber –1 ein gültiger `char`-Wert ist, hat man hier den größeren Datentyp nehmen müssen. Aus diesem Grund ist es *falsch* den Rückgabewert von `getchar()` *vor* dem Vergleich mit `EOF` zu casten. Erst muss mit `EOF` (als `int`) verglichen werden, dann gecastet, sonst bricht der Lesevorgang möglicherweise zu früh ab, nämlich wenn in den Daten zufällig ein `0xFF` vorkommt, das vorzeichenbehaftet einer –1 entspricht.

getchar()

Korrekter Umgang mit EOF

```
int c;
FILE* fd = fopen("test.txt", "r");
while ((c = fgetc(fd)) != EOF) {
    printf("%c", (unsigned char) c);
}
```

Dass die `putchar()`- und `fputc()`-Funktionen auch einen `int` und kein `unsigned char` nehmen, dient der Bequemlichkeit des Programmierers, der sich so teilweise einen Cast ersparen kann.

putchar()
fputc()

3.6 Zeilen-I/O [27]

Anstatt einzelne Zeichen kann man auch direkt auf Zeilen arbeiten. Die entsprechenden Funktionen akzeptieren bzw. liefern Zeichenketten in Form von `char*`.

Funktionen für den Zugriff auf Zeilen

- `char * fgets (char *buf, int size, FILE *in)`
  - liest die nächste Zeile von `in` in `buf`
  - kehrt bei `'\n'` zurück oder wenn `size - 1` Zeichen gelesen wurden
  - `'\n'` selbst wird auch zurückgegeben
  - gibt Zeiger auf `buf` zurück oder `NULL` im Fehlerfall
  - *auf keinen Fall* `gets(char*)` verwenden ⇒ *Buffer-Overflow*

fgets

- `int fputs (const char *str, FILE *out)`
  - schreibt den String `str` in die Datei `out`
  - stoppt beim `'\0'`
  - gibt die Anzahl der geschriebenen Zeichen zurück, oder `EOF` bei Fehler

fputs

Die Funktion `gets` ist so gefährlich, dass sie inzwischen aus dem Standard entfernt wurde und der Compiler bei ihrer Verwendung entsprechende Fehlermeldungen erzeugt.

gets

3.7 Formatierte I/O [28]

Analog zu den Funktionen `scanf`, `sprintf` etc., mit denen man Daten in Strings oder Strings in andere Datentypen umwandeln kann, gibt es entsprechende Funktionen auch für Eingabe- und Ausgabe. Anstatt des Strings ist hier die Datei die Quelle bzw. das Ziel der Daten.

- `int fscanf (FILE *in, const char *format, ...)` fscanf
  Analog zu `scanf` aber für Datei
- `int fprintf (FILE *out, const char *format, ...)` fprintf
  Analog zu `sprintf` aber für Datei

`fscanf(...)` sollte man nicht verwenden. Besser `fgets(str, ...)` zusammen mit `sscanf(str, ...)`

3.8 Von der Console lesen [29]

Ein Spezialfall der Ein- und Ausgabe ist der Zugriff auf die Konsole.

- Mit `fgets(str, ...)` und `sscanf(str, ...)` kann man Daten von der Konsole lesen

```c
int main(int argc, char** argv) {
    char buffer[255];
    int zahl;
    printf("Bitte geben Sie eine Zahl ein: ");

    if (fgets(buffer, 254, stdin) != 0) {
        sscanf(buffer, "%d", &zahl);
        printf("Die Zahl war %d\n", zahl);
    }
}
```

Ausgabe

```
Bitte geben Sie eine Zahl ein: 62
Die Zahl war 62
```

Kapitel 4

Unix Input/Output

4.1 Einführung [31]

Die C-Standardbibliothek bietet eine Reihe von I/O-Funktionen, die zum Lesen und Schreiben von Dateien verwendet werden können. Dazu gehören Funktionen wie `fopen`, `fread`, `fwrite`, `fclose` usw. Diese Funktionen abstrahieren die zugrunde liegenden Betriebssystemaufrufe und stellen eine plattformunabhängige Möglichkeit dar, Dateien zu verarbeiten.

Es gibt jedoch auch I/O-Funktionen auf Betriebssystemebene, wie beispielsweise die Funktionen `open`, `read`, `write` und `close` unter Unix-Systemen. Diese Funktionen bieten eine direkte Schnittstelle zum Betriebssystem und erlauben es, Dateien auf einer niedrigeren Ebene zu öffnen, zu lesen, zu schreiben und zu schließen.

- Bisherige Funktionen sind C-Standard aus <`stdio.h`>
  $\Rightarrow$ plattformunabhängig
- Unix und Windows haben eigene I/O-Bibliotheken
  - sind plattformabhängig
  - Kommunikation, die nicht im C-Standard vorgesehen ist (z. B. Netzwerk)
  - mehr Kontrolle und erweiterte Funktionen
- bei Linux zu finden in `fcntl.h`, `unistd.h`, `sys/socket.h` etc.

`fcntl.h`
`unistd.h`
`sys/socket.h`

Es gibt mehrere Gründe, warum zusätzlich zu den C-Standardbibliotheksfunktionen Betriebssystem-I/O-Funktionen verwendet werden:

1. *Spezifische Betriebssystemfunktionen*: Betriebssystem-I/O-Funktionen ermöglichen den Zugriff auf spezifische Funktionen und Eigenschaften des Betriebssystems. Je nach Betriebssystem können bestimmte I/O-Operationen nur über die entsprechenden Betriebssystemfunktionen durchgeführt werden. Zum Beispiel kann die `open`-Funktion unter Unix spezielle Flags wie den Zugriffsmodus oder Dateiattribute akzeptieren, die in der C-Standardbibliotheksfunktion `fopen` nicht verfügbar sind.
2. *Performance*: Betriebssystem-I/O-Funktionen bieten oft eine bessere Leistung als die entsprechenden Funktionen auf C-Ebene. Dies liegt daran, dass Betriebssystemfunktionen

direkt mit den internen Mechanismen des Betriebssystems interagieren können, um die I/O-Operationen zu optimieren.

3. *Erweiterte Funktionalität*: Betriebssystem-I/O-Funktionen können erweiterte Funktionen bieten, z. B. zur Verwaltung von Netzwerkverbindungen, Gerätedateien oder speziellen Ressourcen.
4. *Nicht-dateibasierte E/A*: Während die C-Standardbibliothek hauptsächlich auf die Verarbeitung von Dateien ausgerichtet ist, bieten Betriebssystem-I/O-Funktionen weitere Optionen, z. B. Netzwerkverbindungen, Prozesskommunikation über Pipes/Sockets oder auf andere Ressourcen wie serielle Schnittstellen oder Hardwaregeräte zuzugreifen.

4.2 File-Descriptor [32]

Alles ist eine Datei

Die Philosophie von Unix lautet **Alles ist eine Datei**. Dieses Konzept besagt, dass in Unix-artigen Betriebssystemen nahezu alle Ressourcen und Geräte als Dateien behandelt werden, unabhängig davon, ob es sich um tatsächliche Dateien auf dem Dateisystem handelt oder um andere Ressourcen wie Eingabegeräte, Netzwerkverbindungen, Drucker, Serielle-Schnittstellen usw.

Unix-Philosophie: *alles ist eine Datei*

- dieselben Funktionen können angewendet werden (`read`, `write`, `close`)
- **File Descriptor** (*FD*)

  File Descriptor

  - repräsentiert jede geöffnete Ressource
  - `unsigned int`-Wert
  - einer Ressource können mehrere File-Deskriptoren zugeordnet sein
- vordefinierte File-Deskriptoren (sind bereits automatisch geöffnet)
  - `0`: Standardeingabe (`stdin`)
  - `1`: Standardausgabe (`stdout`)
  - `2`: Standardfehlerausgabe (`stderr`)

Durch die Behandlung aller Ressourcen als Dateien können einheitliche Schnittstellen und Methoden für den Zugriff und die Verarbeitung verwendet werden. Statt spezifischer Funktionen für jedes Gerät oder jede Ressource zu haben, können Entwickler auf eine einheitliche Art und Weise auf sie zugreifen, indem sie Dateioperationen verwenden.

Ein entscheidendes Konzept, das mit der Philosophie „Alles ist eine Datei“ einhergeht, ist der *File Descriptor*. Er ist eine nicht-negative Ganzzahl (`unsigned int`), die eine geöffnete Ressource (z. B. Datei) repräsentiert und dazu dient, diese in den verschiedenen Funktionen anzusprechen.

Standardmäßig sind drei File-Deskriptoren geöffnet:

- File-Descriptor `0`: Standardeingabe (`stdin`) `stdin`
- File-Descriptor `1`: Standardausgabe (`stdout`) `stdout`

- File-Descriptor 2: Standardfehlerausgabe (`stderr`) stderr

Diese werden verwendet, um Eingabe von der Tastatur zu lesen (`stdin`), Ausgabe auf den Bildschirm zu schreiben (`stdout`) und Fehlermeldungen zu senden (`stderr`).

Schreiben auf stdout

```c
#include <string.h>
#include <unistd.h>

int main(int argc, char** argv) {
    char* text = "Hello, world!\n";
    size_t len = strlen(text);

    /* Direkt auf stdout per FD schreiben */
    write(1, text, len);

    /* FD NICHT schließen, wir haben ihn auch nicht geöffnet */
}
```

Die Funktion `write` wird aufgerufen, um den Inhalt der Zeichenkette `text` mit einer Länge von `len` direkt auf die Standardausgabe (`stdout`) zu schreiben. Der erste Parameter der Funktion ist 1, was dem Dateideskriptor für die Standardausgabe entspricht. Der zweite Parameter ist der Zeiger auf den Text, der geschrieben werden soll, und der dritte Parameter ist die Anzahl der zu schreibenden Zeichen.

4.3 File öffnen [34]

Wie bereits erwähnt versteckt sich hinter dem Begriff „Datei“ alles, was in irgendeiner Weise Daten liefern oder speichern kann. Deswegen muss im Folgenden immer, wenn der Begriff „File“ oder „Datei“ fällt klar sein, dass damit viel mehr als Dateien auf der Festplatte gemeint ist.

- `int open (const char *pathname, int flags)` open
  Öffnet eine Datei zum Lesen und/oder Schreiben
  - `flags` kontrolliert, wie Datei geöffnet wird, z. B. `O_WRONLY`
  - mehrere Flags werden über Oder `|` verknüpft
- `int creat (const char *pathname, mode_t mode)` creat
  entspricht `open` mit `flags` = `O_CREAT|O_WRONLY|O_TRUNC`, d. h. Datei wird angelegt und zum Schreiben geöffnet
- `int openat (int dirfd, const char *pathname, int flags)` openat
  Wie `open` allerdings ist `pathname` relativ zum Verzeichnis `dirdf`

`open` und `openat` sind überladen und erlauben es, die Berechtigungen der Datei beim Anlegen zu bestimmen. Hierzu haben beide einen weiteren Parameter `mode`.

- `int open(const char *pathname, int flags, mode_t mode)`
- `int openat(int dirfd, const char *pathname, int flags, mode_t mode)`

Beispiele für `mode` sind (siehe `man 2 open`)

- `S_IRWXU` `00700` user (file owner) has read, write, and execute permission
- `S_IRUSR` `00400` user has read permission
- `S_IWUSR` `00200` user has write permission
- `S_IXUSR` `00100` user has execute permission
- `S_IRWXG` `00070` group has read, write, and execute permission
- `S_IRGRP` `00040` group has read permission

4.4 Daten lesen und schreiben [35]

- `ssize_t read (int fd, void *buf, size_t count)` `read`
  Liest `count` Bytes vom File-Deskriptor `fd` in den Puffer `buf`.
- `ssize_t write (int fd, const void *buf, size_t count)` `write`
  Schreibt `count` Bytes aus dem Puffer `buf` in den File-Desktiptor `fd`
- `off_t lseek (int fd, off_t offset, int whence)` `lseek`
  Verschiebt die aktuelle Position in der Datei

Fehler werden über den Rückgabewert angezeigt, deswegen `ssize_t`

- `ssize_t` wird benutzt, um eine Anzahl Elementen *und* einen Fehler-Indikator zu speichern (`signed`)
- `size_t` wird benutzt, um eine Anzahl von von Elementen zu speichern (`unsigned`)
- `off_t` wird benutzt, um Dateigrößen *und* einen Fehler-Indikator zu speichern (`signed`)

In C gibt es drei verwandte, aber unterschiedliche Datentypen: `ssize_t`, `size_t` und `off_t`:

1. `ssize_t` (Signed Size Type): `ssize_t`
   - `ssize_t` ist ein vorzeichenbehafteter Ganzzahl-Datentyp, der zur Darstellung von Größen oder Ergebnissen von Operationen verwendet wird, bei denen das Vorzeichen relevant ist.
   - Der Datentyp `ssize_t` wird häufig in Funktionen verwendet, die Größen oder Ergebnisse zurückgeben können, bei denen negative Werte eine spezifische Bedeutung haben, beispielsweise Fehlercodes.
   - Beispiele für Funktionen, die `ssize_t` verwenden, sind `read`, `write` und `lseek`, bei denen die Rückgabewerte die Anzahl der gelesenen/schreibenden Bytes oder die Position im File-Deskriptor darstellen können. Negative Werte werden verwendet, um Fehler oder spezielle Zustände zu signalisieren.
2. `size_t` (Size Type): `size_t`
   - `size_t` ist ein vorzeichenloser Ganzzahl-Datentyp, der zur Darstellung von Größen oder Indizes verwendet wird.

- Der Datentyp `size_t` wird oft für Größen von Objekten, Arrays, Speicherblöcken oder Dateien verwendet.
- Beispiele für Funktionen oder Operationen, die `size_t` verwenden, sind `sizeof`, `strlen`, `malloc` und `fread`. Diese Funktionen geben die Größe eines Objekts, die Länge einer Zeichenkette oder die Anzahl der gelesenen Bytes zurück.

3. `off_t` (Offset Type): `off_t`
   - ist ein vorzeichenhafter Ganzzahl-Datentyp, der zur Darstellung von Dateioffset-Werten verwendet wird.
   - Er wird häufig in Zusammenhang mit Dateioperationen und Positionierungsoperationen verwendet, bei denen der Dateizeiger innerhalb einer Datei verschoben wird.

```c
#define BUFFER_SIZE 1024

int main(int argc, char** argv) {
    char buffer[BUFFER_SIZE];
    int source_fd = open("source.txt", O_RDONLY); /* zum Lesen öffnen */
    int destination_fd = open("destination.txt", /* zum Schreiben öffnen */
           O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);

    ssize_t bytes_read;
    while ((bytes_read = read(source_fd, buffer, BUFFER_SIZE)) > 0) {
        ssize_t bytes_written = write(destination_fd, buffer, bytes_read);
    }

    /* Schließen der Dateien */
    close(source_fd);
    close(destination_fd);
    return 0;
}
```

In diesem Beispielprogramm wird eine Quelldatei namens `source.txt` geöffnet, deren Inhalt gelesen und in eine Zieldatei namens `destination.txt` geschrieben. Die Funktionen `open`, `read` und `write` werden verwendet, um die Dateien zu öffnen und Daten zu lesen und zu schreiben.

Es ist zu beachten, dass Fehlerüberprüfungen und Fehlerbehandlungen in dem Beispielprogramm weggelassen wurde. Vollständig würde das Programm wie folgt aussehen:

Programm mit Fehlerbehandlung und #include

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

#define BUFFER_SIZE 1024
```

```c
int main(int argc, char** argv) {
    char buffer[BUFFER_SIZE];

    int source_fd = open("source.txt", O_RDONLY); /* zum Lesen öffnen */
    if (source_fd == -1) {
        perror("Fehler beim Öffnen der Quelldatei");
        exit(1);
    }

    /* zum Schreiben öffnen (falls sie nicht existiert, wird sie erstellt) */
    int destination_fd = open("destination.txt", /* zum Schreiben öffnen */
            O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
    if (destination_fd == -1) {
        perror("Fehler beim Öffnen der Zieldatei");
        exit(1);
    }

    /* Lesen von Daten aus der Quelldatei und Schreiben in die Zieldatei */
    ssize_t bytes_read;
    while ((bytes_read = read(source_fd, buffer, BUFFER_SIZE)) > 0) {
        /* Schreiben der Daten in die Zieldatei */
        ssize_t bytes_written = write(destination_fd, buffer, bytes_read);
        if (bytes_written == -1) {
            perror("Fehler beim Schreiben in die Zieldatei");
            exit(EXIT_FAILURE);
        }
    }

    /* Schließen der Dateien */
    close(source_fd);
    close(destination_fd);

    return 0;
}
```

4.5 Pipe [38]

Eine **Pipe** erlaubt es zwei Prozessen miteinander zu kommunizieren

Pipe

- **anonyme Pipe**: File-Deskriptor muss beiden Prozessen bekannt sein (`pipe`)
- **named Pipe**: Spezielle Datei im Filesystem, über die Prozesse kommunizieren können (`mkfifo`)

anonyme Pipe

`pipe`

named Pipe

`mkfifo`

Pipe hat zwei Enden (`pipe` liefert zwei File-Deskriptoren)

- *write End* in das geschrieben wird
- *read End* aus dem man die Daten wieder auslesen kann

Für eine bidirektionale Kommunikation benötigen die Prozesse zwei Pipes

Anonyme Pipes werden normalerweise zusammen mit `fork` benutzt, da so der Elternprozess die Pipe öffnet und dann die File-Deskriptoren an die per `fork` erzeugten Kinder weitergibt.

```c
int pipefd[2]; /* [0]: read, [1]: write */
char buffer[BUFFER_SIZE];
pipe(pipefd); /* Pipe erstellen */

/* Prozess duplizieren */
pid_t pid = fork();
if (pid > 0) { /* Elternprozess */
    close(pipefd[0]);
    const char* message = "Hallo, Kindprozess!";
    write(pipefd[1], message, strlen(message) + 1);
    close(pipefd[1]);
} else { /* Kindprozess */
    close(pipefd[1]);
    ssize_t bytes_read = read(pipefd[0], buffer, BUFFER_SIZE);
    printf("Nachricht vom Elternprozess erhalten: %s\n", buffer);
    close(pipefd[0]);
}
```

In diesem Beispiel wird eine Pipe mit der Funktion `pipe` erstellt. Der Prozess wird mit `fork` dupliziert, wodurch ein Elternprozess und ein Kindprozess entstehen. Der Elternprozess schreibt eine Nachricht in die Pipe und der Kindprozess liest diese Nachricht aus der Pipe.

Die unbenutzten Dateideskriptoren werden geschlossen, um Ressourcenlecks zu vermeiden. Im Elternprozess wird das Lese-Ende der Pipe (`pipefd[0]`) geschlossen, während im Kindprozess das Schreib-Ende der Pipe (`pipefd[1]`) geschlossen wird.

Der Elternprozess schreibt die Nachricht „Hallo, Kindprozess!“ in die Pipe mithilfe der Funktion `write`. Der Kindprozess liest die Nachricht aus der Pipe mithilfe der Funktion `read` und gibt sie auf der Konsole aus.

Vollständige Version mit Fehlerbehandlung und #include

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define BUFFER_SIZE 1024

int main(int argc, char** argv) {
    int pipefd[2];
    pid_t pid;
    char buffer[BUFFER_SIZE];
```

```
    /* Pipe erstellen */
    if (pipe(pipefd) == -1) {
        perror("Fehler beim Erstellen der Pipe");
        exit(EXIT_FAILURE);
    }

    int read_end = pipefd[0];
    int write_end = pipefd[1];

    /* Prozess duplizieren */
    if ((pid = fork()) == -1) {
        perror("Fehler beim Forken des Prozesses");
        exit(EXIT_FAILURE);
    }

    if (pid > 0) {
        /* Elternprozess (schreibt in die Pipe) */
        close(read_end); /* Schließe das Lese-Ende der Pipe */

        /* Daten in die Pipe schreiben */
        const char* message = "Hallo, Kindprozess!";
        write(write_end, message, strlen(message) + 1);

        /* Schließe das Schreib-Ende der Pipe */
        close(write_end);
    } else {
        /* Kindprozess (liest aus der Pipe) */
        close(write_end); /* Schließe das Schreib-Ende der Pipe */

      /* Daten aus der Pipe lesen */
      ssize_t bytes_read = read(read_end, buffer, BUFFER_SIZE);
      if (bytes_read == -1) {
          perror("Fehler beim Lesen aus der Pipe");
          exit(EXIT_FAILURE);
      }

      printf("Nachricht vom Elternprozess erhalten: %s\n", buffer);

      /* Schließe das Lese-Ende der Pipe */
      close(read_end);
    }

    return 0;
}
```

4.6 Socket [40]

Netzwerkverbindungen erfolgen über *Socket*

1. Socket mit `socket` erstellen `socket`
   ⇒ gibt einen Dateideskriptor (r/w) zurück

2. Socket mit `bind` an IP-Adresse und Portnummer zu binden (nur Server) `bind`
3. Socket mit `connect` mit einem Server verbinden (nur Client) `connect`
4. Mit `read` und `write` Daten lesen und schreiben
5. Socket mit `close` wieder schließen

Client

```c
char buffer[BUFFER_SIZE];

/* Socket erstellen */
int socket_fd = socket(AF_INET, SOCK_STREAM, 0);

/* Serveradresse und Portnummer konfigurieren */
struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(8080);
server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

/* Verbindung zum Server herstellen */
connect(socket_fd, (struct sockaddr*)&server_addr, sizeof(server_addr));
/* Daten vom Socket lesen */
ssize_t bytes_read = read(socket_fd, buffer, BUFFER_SIZE);
close(socket_fd); /* Socket schließen */
```

Im vorherigen Beispiel wurden die Includes und die Fehlerbehandlung weggelassen. Vollständig sieht es wie folgt aus.

Client mit Fehlerbehandlung

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define BUFFER_SIZE 1024

int main(int argc, char** argv) {
    char buffer[BUFFER_SIZE];
    struct sockaddr_in server_addr;

    /* Socket erstellen */
    int socket_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (socket_fd == -1) {
        perror("Fehler beim Erstellen des Sockets");
        exit(EXIT_FAILURE);
    }
```

```
    /* Serveradresse und Portnummer konfigurieren */
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(8080);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    /* Verbindung zum Server herstellen */
    if (connect(socket_fd, (struct sockaddr*)&server_addr, sizeof(server_addr)) == ↩
        -1) {
        perror("Fehler beim Verbinden mit dem Server");
        exit(EXIT_FAILURE);
    }

    /* Daten vom Socket lesen */
    ssize_t bytes_read = read(socket_fd, buffer, BUFFER_SIZE);
    if (bytes_read == -1) {
        perror("Fehler beim Lesen vom Socket");
        exit(EXIT_FAILURE);
    }

    /* Verarbeitung der gelesenen Daten */
    /* ... */

    /* Socket schließen */
    close(socket_fd);

    return 0;
}
```

Index