1 Alle Aufgaben

Aufgabe 1.1. (Code verstehen – mittel) Gegeben sei folgender C-Code:

Codebeispiel 1: Mysteriöses Programm

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main(int argc, char *argv[])
5  {
6      int a = 0;
7      for (int i = 1; i < argc; i++) {
8          int b = atoi(argv[i]);
9          a += b;
10         printf("%d: %d (%d)\n", i, b, a);
11     }
12     printf("%d\n", a);
13
14     int b = 0;
15     if (argc > 1) { b = a/(argc-1); }
16     printf("%d\n", b);
17 }
```

Sind die Aufrufe „legal“? Was wird ausgegeben bei folgenden Eingaben (nicht ausprobieren) und was steht in den Variablen a, b und c zu welchem Zeitpunkt?

1. `$ ./prog abc` (hierzu s. [2, S. 307])
2. `$ ./prog`
3. `$ ./prog 0`
4. `$ ./prog 1 2 3`
5. `$ ./prog 1 2 4`

Aufgabe 1.2. (C-Code in Assembler auf Papier kompilieren – schwer) Gegeben sei folgender C-Code:

Codebeispiel 2: Ausgabe von Konsolenargument

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    for (int i = 0; i < argc; i++) {
        printf("%d: %s\n", i+1, argv[i]);
    }
}
```

Übersetzen Sie den Code manuell in (in etwa) äquivalenten Assembler-Code (NASM).

Aufgabe 1.3. (Schaltjahrberechnung – leicht) Ein Schaltjahr ist ein Jahr, welches durch 400 teilbar ist, oder zwar durch 4 aber nicht durch 100 teilbar ist. Dies entspricht der Formel

$$L(Y) = 4 \mid Y \wedge 100 \nmid Y \vee 400 \mid Y.$$

Schreiben Sie ein Programm, dass sein erste Argument ausliest und berechnet ob es sich um ein Schaltjahr handelt. Kommt Ihr Programm mit der falschen Anzahl an Argumenten zurecht? Wie ist es mit Eingaben, die keine Zahlen sind?

Codebeispiel 3: Beispielausgabe

```
$ ./leap 2008
The year 2008 is a leap year.
$ ./leap 2100
The year 2100 is not a leap year.
$ ./leap 2000
The year 2000 is a leap year.
```

Aufgabe 1.4. (Ostertagberechnung – leicht) Implementieren Sie den Algorithmus von Gauß zur Bestimmung des Ostertages [1] in einem gegebenen Jahr. Nehmen Sie die Jahreszahl als Argument entgegen.

Codebeispiel 4: Beispielausgabe

```
$ ./easter 2024
The easter date of the year 2024 is the 31. of March
$ ./easter 2018
The easter date of the year 2018 is the 1. of April
```

Aufgabe 1.5. (Binomialkoeffizienten – leicht) Schreiben Sie eine Funktion

```
unsigned long choose(unsigned long n, unsigned long k)
```

die den Binomialkoeffizienten $\binom{n}{k}$ errechnet. Dieser ist definiert Quotient zweier Produkte:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{1 \cdot 2 \cdots k}$$

Schreiben Sie ein Programm, das Ihre Implementierung von `choose` aufruft und vergewissern Sie sich, dass Ihre Implementierung korrekt ist.

Untersuchen Sie, was passiert, wenn Sie größere Werte eingeben. Versuchen Sie, das Verhalten zu erklären. Können Sie Abhilfe schaffen?

Analysieren Sie die Laufzeit Ihrer Implementierung und denken Sie darüber nach, ob es einen besseren Algorithmus zur Berechnung des Binomialkoeffizienten gibt.

Aufgabe 1.6. (Rate die Zahl – leicht) Schreiben Sie ein Programm, dass den Benutzer zum Raten einer Zahl auffordert. Dem Benutzer soll nach jedem Versuch angezeigt werden, ob die geratene Zahl größer, kleiner, oder gleich der vom Programm erdachten Zahl ist. Hat der Benutzer richtig geraten, so soll ihm angezeigt werden, wie viele Versuche er gebraucht hat. Seien Sie kreativ in der Ausgestaltung des Spiels.

Verwenden Sie `rand()` aus `<stdlib.h>`, um eine Zufallszahl zu erzeugen, nachdem sie den Zufallszahlengenerator mit `srand(time(NULL))` aus `<time.h>` initialisiert haben. Verwenden Sie `scanf`, um eine Zahl vom Benutzer zu lesen. Lesen Sie die entsprechenden Seiten aus dem Online-Handbuch, wenn Sie sich in den Details unsicher sind.

Aufgabe 1.7. (Bitfunktionen – leicht) Schreiben Sie C-Funktionen zur Berechnung der folgenden Bit-Operationen. Jede Funktion soll ein Argument vom Typ `unsigned int` entgegen nehmen und das Ergebnis erneut als `unsigned int` ausgeben.

sadd(x) zählt, wie viele Bits in `x` gesetzt sind

ctz(x) zählt, wie oft `x` glatt durch 2 teilbar ist

rev(x) kehrt die Reihenfolge der Bits von `x` um

bswap(x) kehrt die Reihenfolge der Bytes von `x` um

Schreiben Sie ein Programm, das Ihre Implementierungen ausprobiert und vergewissern Sie sich ihrer Korrektheit.

Versuchen Sie, Ihre Implementierung hinsichtlich der Performanz zu optimieren.

Aufgabe 1.8. (Wurzelziehen – mittel) Die Mathematikbibliothek `-lm` enthält die Funktion `sqrt()`, die die Quadratwurzel einer Gleitkommazahl bestimmt.

Implementieren Sie Ihre eigene Quadratwurzelfunktion `my_sqrt()` und vergewissern Sie sich der Korrektheit ihrer Implementierung.

Untersuchen Sie die Genauigkeit Ihrer Implementierung indem Sie sie mit `sqrt()` vergleichen.

Aufgabe 1.9. (Maschinen-Epsilon – mittel) Das Maschinen-Epsilon ist die kleinste positive Gleitkommazahl ε, sodass $1.0 \neq 1.0 + \varepsilon$. Idealerweise wäre $\varepsilon = +0$, aber leider haben Gleitkommazahlen nur endliche Genauigkeit.

Finden Sie eine Möglichkeit, das Maschinen-Epsilon zu bestimmen. Bestimmen Sie das Maschinen-Epsilon für die Typen `float`, `double`, und `long double`. Der Platzhalter `%e` ist hilfreich, um derartig kleine Zahlen mit `printf` auszugeben.

Versuchen Sie das Ergebnis zu interpretieren.

Aufgabe 1.10. (Kommentarentferner – sehr schwer) Schreiben Sie ein Programm, welches aus einer C-Quelltextdatei alle Kommentare entfernt. Ihr Programm muss in der Lage sein, sowohl Zeilen-Kommentare der Form `// ...` als auch Bereichs-Kommentare der Form `/* ... */` zu erkennen, und zu entfernen. Dazu müssen Sie Fortsetzungszeilen erkennen und Kommentare innerhalb von Zeichenketten geschickt ignorieren. Lesen Sie ggf. den C-Standard, um die genauen Feinheiten der Syntax zu nachzuvollziehen.

Aufgabe 1.11. (Einheitenumrechner – leicht) Schreiben Sie ein interaktives Programm, mit dem der Benutzer zwischen verschiedenen Einheiten umrechnen kann. Der Benutzer soll die Möglichkeit haben, zwei Einheiten und einen Betrag einzugeben, dass Programm gibt den Betrag dann in der Zieleinheit aus.

Denken Sie darüber nach, wie man das Programm gestalten kann, sodass eine Erweiterung auf zusätzliche Einheiten möglichst einfach ist. Hierzu ist es ggf. hilfreich, im Programm eine Tabelle anzulegen, die für jede unterstützte Einheit die Art der Einheit (z. B. Währung, Länge, Masse) und ihren Umrechnungsfaktor speichert.

Aufgabe 1.12. (Cache-Lokalität – leicht) Ein CPU-Cache kann konsekutive Bytes in n-Byte–Blöcken aus dem Hauptspeicher vorlagern, um den Zugriff zu beschleunigen.

Schreibt ein Programm welches

- eine $n \times m$-Matrix initalisiert
- die Werte darin aufsummiert, erst die Spalten, dann die Zeilen
- die Werte darin aufsummiert, erst die Zeilen, dann die Spalten

Welche Version ist schneller? Warum (wie sieht die Matrix im Speicher aus)?

Die Arrays sollten sehr groß sein, um einen Effekt zu messen

Aufgabe 1.13. (Überlauf von time_t – leicht) In der Manpage `tunefs(8)` gibt/gab es das Kommentar „Take this out and a Unix Daemon will dog your steps from now until the `time_t`'s wrap around“ zu dem Satz „You can tune a file system, but you cannot tune a fish“ (ein Daemon ist ein Hintergrundprogramm). Das `time_t` bezieht sich auf einen Datentyp aus `time.h`, welcher die aktuelle POSIX oder UNIX Epoch time hält, d. i. die Sekunden seit dem 1.1.1970.

Die Aufgabe ist: Schreibt ein Programm, welches berechnet, wann dieser Datentyp nicht mehr die aktuelle Zeit darstellen kann.

Aufgabe 1.14. (Kopierprogramm – leicht) Schreiben Sie ein Programm, mit dem Sie eine Datei in eine andere Kopieren können. Der Nutzer soll in der Lage sein, das Programm wie folgt aufzurufen:

```
./kopiere Quelle Ziel
```

Folgende Funktionen sind zur Lösung der Aufgabe hilfreich: `fopen`(3), `fread`(3), `fwrite`(3), und `fclose`(3).

Implementieren Sie den Kopiervorgang, indem Sie wiederholt in einer Schleife ein Stück der Eingabedatei in einen Puffer einlesen und dieses in die Ausgabedatei schreiben. Lesen Sie die Dokumentation von `fread`(3), spezifisch den Abschnitt *Return Value*, um herauszufinden, wie Sie das Ende der Eingabedatei erkennen.

Experimentieren Sie mit verschiedenen Puffergrößen. Können Sie einen Unterschied in der Geschwindigkeit feststellen?

Aufgabe 1.15. (verlinkte Listen – leicht) Schreiben Sie ein Programm, dass Zahlen einließt, bis `-1` eingegeben wird. Jede Zahl soll als Glied in eine verlinkte Liste eingehängt werden. Ist die Eingabe abgeschlossen, sollen die Zahlen in umgekehrter Reihenfolge ausgegeben werden. Allozieren Sie die Glieder der Liste mit **malloc**(3). Sie können folgenden Coderahmen für die Struktur der Liste verwenden.

```
1 /* Glied einer verlinkten Liste */
2 struct glied {
3     int wert;
4     struct glied *nachfolger;
5 };
```

Aufgabe 1.16. (Listen sortieren – mittel) Modifizieren Sie das Programm aus der vorherigen Aufgabe, sodass es die eingelesenen Zahlen sortiert ausgibt. Sortieren Sie dazu die von Ihrem Programm angelegte Liste mit dem Algorithmus *Mergesort*.

Aufgabe 1.17. (Stringsuche – mittel) Schreiben Sie Ihre eigene Variante der Funktion `strstr`(3). Schreiben Sie ein Programm, um diese zu testen. Hierzu bietet es sich an, dass Ihre eigene Implementierung mit der von System bereitgestellten Implementierung zu vergleich.

Analysieren Sie die Laufzeit Ihrer Implementierung und recherchieren Sie bessere Algorithmen zur Suche von Strings in Strings.

Aufgabe 1.18. (`env`(1) – mittel) Schreiben Sie eine Implementierung des Programmes `env`(1). Dieses hat zwei Aufgaben:

Ohne Argumente aufgerufen, gibt `env`(1) den Inhalt der zur Zeit gesetzten Umgebungsvariablen aus. Diese können Sie durch Inspektion der globalen Variable `environ`(7) auslesen.

Werden Argumente übergeben, so wird jedes Argument der Form

`Schluessel=Wert`

mit `putenv`(3) als Variable in die Umgebung aufgenommen. Alle Argumente ab dem ersten Argument, das nicht diese Form hat, werden als Name und Argumente eines auszuführenden Programmes verstanden, welches nach Setzen der Umgebungsvariablen mit `execvp`(2) ausgeführt wird.

Weitere Funktionalität von `env`(1) ist nicht Teil der Aufgabe.

Aufgabe 1.19. (Stack als Array – mittel) Ein Stack ist eine Datenstruktur auf der man auf das oberste Element zugreifen kann: Man kann es löschen (POP), angucken (PEEK) oder ein anderes Element „darüber legen“ (PUSH).

Implementiert diese Funktionalität mithilfe einer Struktur in welcher die Daten in einem Array gespeichert werden. Als Idee: Die Struktur benötigt mindestens Einträge um 1. die aktuelle Position des Stacks zu speichern und 2. zu wissen, ob der Stack leer oder voll ist. Diese wird dann als Pointer an die Funktionen übergeben, samt möglicherweise weiteren benötigten Argumenten.

Aufgabe 1.20. (Nutzung des Stacks: Taschenrechner – schwer) Entwickeln Sie einen Taschenrechner der mit sogenannter umgekehrter polnischer Notation bzw. Postfix-Notation arbeitet, d. h. an Stelle von $a + b$ würde man schreiben $a\ b\ +$. Das Praktische dabei ist, dass man Klammern komplett weglassen kann. Dies wird deutlich, schreibt man unterschiedlich-geklammerte Ausdrücke in PN:

$(2 + 3) \cdot 4$	$2\ 3\ +\ 4\ \cdot$
$2 + (3 \cdot 4)$	$4\ 3\ \cdot\ 2\ +$

Die Idee bei der Implementierung ist einen Stack zu nutzen:

- Wird eine Zahl gelesen, wird diese auf den Stack gepusht
- Wird ein Operator gelesen, werden n Elemente vom Stack gepopt, wobei n die Anzahl der Operanden ist, die dieser Operator benötigt. Das Ergebnis wird berechnet und wieder auf den Stack gepusht.

Am Ende ist das Ergebnis das einzige Elemente auf dem Stack.

Testet den Algorithmus erst von Hand um euch die Funktionsweise klar zu machen!

Implementation: Ihr könnt die Eingabe entweder über Programmargumente annehmen (etwas einfacher), oder direkt von `stdin` einlesen. In letzterem Fall habt ihr die Möglichkeit euer Programm auch so zu erweitern, dass nach einer erfolgreichen Berechnung ihr weiterrechnen könnt und nicht das Programm erneut starten müsst. Das ist gerade dann sinnvoll, wenn ihr später Variablen einbaut, das heißt ihr könnt Ergebnisse zwischenspeichern um sie später wiederzubenutzen.

Aufgabe 1.21. (Worthäufigkeit – schwer) Schreiben Sie ein Programm, das Text aus der Eingabe ließt und zählt, wie oft jedes Wort in diesem vorkommt. Ist die Eingabe erschöpft, so geben Sie alle gefundenen Worte und ihre Häufigkeiten möglichst nach Häufigkeit sortiert aus. Achten Sie darauf, dass Satz- und Leerzeichen nicht Teil von Worten sind.

Machen Sie sinnvolle Annahmen über die Eingabe, z. B. dass kein Wort länger als 16 Buchstaben ist und dass es insgesamt nicht mehr als 1000 verschiedene Worte gibt. Denken Sie über eine geeignete Datenstruktur zur Ablage der Worte. Sie können zu diesem Zweck den oben angebeben Code-Rahmen um weitere Felder und Strukturen erweitern.

Aufgabe 1.22. (base64 – schwer) Schreiben Sie zwei Programme, die Dateien im Base64-Format kodieren bzw. aus dem Base64-Format dekodieren. Recherchieren Sie dazu wie das Format Base64 funktioniert. Sie können die Korrektheit Ihrer Implementierung gegen das Linux-Programm `base64` testen.

Aufgabe 1.23. (Euler-Touren – sehr schwer) Schreiben Sie ein Programm, welches eine Euler-Tour durch einen ungerichteten Graphen findet. Lesen Sie den Graphen von der Standardeingabe als Liste von Zahlenpaaren ein. Jedes Zahlenpaar sagt Ihnen, dass es zwsichen den Knoten mit diesen Zahlen eine Kante gibt. Wie oben beenden Sie die Eingabe mit `-1`. Geben Sie die Tour als Folge von Knoten, die Sie besuchen aus. Gibt es keine Euler-Tour, so geben Sie stattdessen `-1` aus.

Recherchieren Sie mögliche Algorithmen zur Lösung des Problems und wählen Sie eine geeignete Datenstruktur. Es ist ggf. sinnvoll, den Graphen als Array von Listen adjazenter Knoten zu repräsentieren.

Aufgabe 1.24. (PPM-Bilder lesen / schreiben – leicht) PPM (portable pixmap) ist eine Familie einfacher Bildformate. Recherchieren Sie im Internet, wie PPM-Bildformate funktionieren und schreiben Sie eine Funktion, die ein PPM-Bild einließt, und als Datenstruktur im Computer ablegt. Sie brauchen nur PPM-Bilder vom Typ `P3` zu unterstützen.

Das Bild soll durch folgende Struktur dargestellt werden:

```
/* ein einzelnes Pixel */
struct pixel {
    int rot, weiss, gruen;
};

/* ein PPM Typ P3 Bild */
struct ppm_bild {
    /* Kopfdaten */
    int breite, hoehe;
    int max_helligkeit;

    /* mit malloc() allozieren */
    struct pixel *leinwand;
};
```

Anschließend schreiben Sie eine Funktion, die eine `ppm_bild` Struktur als Typ `P3` PPM-Bild in eine Datei schreibt.

Prüfen Sie die Korrektheit Ihrer Funktion, indem Sie diese in einem Programm verwenden, welches ein PPM-Bild einließt und wieder in eine Datei schreibt. Das entstehende Bild muss identisch zum Original sein.

Aufgabe 1.25. (PPM-Bildverarbeitung – mittel/schwer) Schreiben Sie ein einfaches Bildverarbeitungsprogramm, welches auf Typ `P3` Bildern arbeitet. Das Bildverarbeitungsprogramm soll in der Lage sein, einfache Transformationen auf dem Bild auszuführen. Hier sind ein paar Beispiele nach aufsteigender Schwierigkeit sortiert:

- Konvertierung von Farbe nach Graustufen oder Schwarzweiß
- Zuschneiden des Bildes
- Drehung um 90°, 180°, und 270°, sowie Spiegelung
- Filterung mit Faltungsmatrizen (Gaußfilter, Schärfungsfilter, Kantenfilter, etc.)
- Skalierung
- Drehung um beliebige Winkel

Sie können sich auch gerne eigene Operationen ausdenken.

Aufgabe 1.26. (Termauswerter – schwer) Schreiben Sie ein Programm, in das Sie einen C-Ausdruck mit Variablen eingeben können. Das Programm soll den Nutzer nach Belegungen für die Variablen fragen und den Wert des Terms ermitteln. Verwenden Sie für alle Variablen den Typ `long long int`. Sie können zur Lösung der Aufgabe den *Shunting-Yard-Algorithmus* verwenden.

Aufgabe 1.27. (tar(1) – schwer) In dieser Aufgabe sollen Sie eine vereinfachte Version des UNIX-Archivprogramms `tar` implementieren, um Archive im Format „ustar“ (UNIX Standard Tape ARchiver) zu erstellen. Ein Archiv ist eine Datei, welche die Inhalte von anderen Dateien enthalt, nebst Meta-Informationen über eben diese – archivierten – Dateien (d. h. Dateiname, zuletzt modifiziert, …).

Archivformat Ein TAR-Archiv besteht aus Blöcken von 512 Bit-Oktets (für uns: Bytes) und schreibt schlicht die Inhalte der Dateien – in solche Blöcke unterteilt – hintereinander. Vor jedem Dateiinhalt steht noch ein Header der die Meta-Informationen bereithält, welchen wir als Datenstruktur in der Datei `ti3tar.h` für euch mitliefern. Bei uns werden alle Daten in ASCII encodiert, das bedeutet, dass auch Zahlen, in Basis 8(!), als Strings gespeichert werden.

Ganz am Ende müssen noch 2 512-Byte Blöcke mit NUL-Bytes geschrieben werden, um das Archiv zu beenden.

a) Euer Programm soll den Schlüssel `x` (extrahiere Archiv) und den Modifikator `f name` (Angabe eines Dateinamen) annehmen; diese werden beim Aufruf zu einem String kombiniert. Ist kein `f` angegeben, soll von `stdin` gelesen werden.

   Aufrufe könnten also sein:

```
# Extrahiere Archiv "archiv.tar"
tar xf archiv.tar
# dito, lese aber ein von stdin
tar x
# *nicht* gültig
```

   Da ihr bis jetzt nur Archive extrahieren könnt, sollte ein Fehler ausgegeben werden, wird der Schlüssel `x` nicht angegeben.

b) Nun erweitert euer Programm, dass es den Schlüssel `c` (erstelle Archiv) versteht. Dieser soll ebenfalls mit `f` kombinierbar sein. Ist diesmal beim Extrahieren kein `f` angegeben, soll nach `stdout` ausgegeben werden. Die zu archivierenden Dateien folgen darauf.

   Aufrufe könnten also sein:

```
# Archiviere folgende Dateien, und schreibe nach stdout
tar c datei1.txt datei2.c datei3.pdf
# dito, schreibe in Archiv "archiv.tar"
tar cf archiv.tar datei1.txt datei2.c datei3.pdf
```

c) Erweitert euer Programm, dass es auch Ordner archivieren kann, hierzu muss es beim Erstellen in die angebenen Ordner rekursieren (und den Ordner selber ebenfalls hinzufügen zum Archiv!). Beachtet, dass dann die `typeflag` nicht mehr `'0'` ist! Das Feld `size` sollte in aller Regel 0 sein.

Z) Passt euer Programm so an, sodass es symbolische Links versteht, d. h. diese ebenfalls archivieren kann. Hier muss ebenfalls die `typeflag` angepasst werden. Nutzt dazu die Funktion `lstat()`. Die Dateigröße *muss* hier als 0 angegeben werden.

Tipps

- *Testet* euer Programm indem ihr mit dem vorinstallierten `tar`-Programm Archive erstellt und die von eurem extrahiert oder andersherum. Da GNU/tar vorinstalliert ist, benötigt ihr möglicherweise einige Flags um Archive zu erstellen, die der obigen Beschreibung möglichst gut entsprechen:

```
# Die '\' lassen uns mehrzeilige Befehle schreiben
tar cf archiv.tar \
  --blocking-factor 1 --format=ustar \
  datei1.txt ...
```

- Um die von euch erstellten Archive auf Byte-Ebene mit denen von GNU/tar zu vergleichen, bietet sich es an, diese mit einem Hex-Editor o. Ä. zu betrachten, bspw. so:

```
# Zeige das Archiv in dem Format
#     pppppppp  bb bb bb bb ... bb |cccccc...|
# an, wobei
#  * pppppppp = Adresse,
#  * bb = Byte in Hex,
#  * c = Byte als ASCII
# und zeige das Scrollbar in dem Programm less an.
hexdump -v -C archiv.tar | less
```

  Alternativ kann man dies mit dem etwas komplexeren Tool Radare2 machen.

- Es ist unter Linux wahrscheinlich nötig, die Präprozessor-Konstante `_POSIX_C_SOURCE` auf den Wert `200809L` zu setzen, bevor ihr Systemheader inkludiert; ggf. ist auch `_XOPEN_SOURCE` auf den Wert `500` zu setzen.

- Es ist praktisch, die Kopieroperationen mit den Funktionen `fread()` und `fwrite()` zu machen, diese aber operieren nur auf Streams, und für `fstat()` benötigt ihr Dateideskriptoren. Nutzt deshalb `fileno()` um für einen Stream den darunterliegenden Dateideskriptor zu bekommen oder mit öffnet die Datei mit `open()` und weißt dieser mit `fdopen()` einen Stream zu.

- Um den Nutzernamen und Gruppennamen aus der uid und gid zu bekommen, gibt es die Funktionen `getpwuid()` respektive `getgrgid()`.

- Mit `s[n]printf()` könnt ihr angenehm Oktalzahlen in die dafür vorgesehenen Felder schreiben, analog mit `sscanf()` parsen.

Weitere Links

- Der POSIX-Standard zu dem Archivierungs-Tool `pax`, welches auch ustar implementiert (unter „ustar Interchange Format“).
  `http://pubs.opengroup.org/onlinepubs/9699919799/utilities/pax.html`

- Radare2 Hex-Editor und Reversing-Tool: `http://radare.org/`

Literatur

[1] Wikipedia contributors. *Gaußsche Osterformel: Eine ergänzte Osterformel*. 2018. URL: `https://de.wikipedia.org/wiki/Gau%C3%9Fsche_Osterformel#Eine_erg%C3%A4nzte_Osterformel`.

[2] Open-Std WG14. *C99 Standard*. C99 + TC1-3. N1256. International Standard, Committe Draft. Version C99. ISO/IEC. 7. Sep. 2007. URL: `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`.