Hochschule Bochum
Bochum University
of Applied Sciences

Software-Entwicklung für eingebettete Systeme

Sommersemester 2012

Prof. Dr. Peter Gerwinski

Inhaltsverzeichnis

Stand: 16. April 2012

0 Eingebettete Systeme

0.1 Was sind eingebettete Systeme?

Was ein eingebettetes System ist und was nicht, ist nicht im mathematischen Sinne scharf abgegrenzt. Eine Möglichkeit für eine sinnvolle Definition lautet:

> Der Ausdruck eingebettetes System (auch engl. embedded system) bezeichnet einen elektronischen Rechner oder auch Computer, der in einen technischen Kontext eingebunden (eingebettet) ist.
>
> http://de.wikipedia.org/wiki/Eingebettetes_System

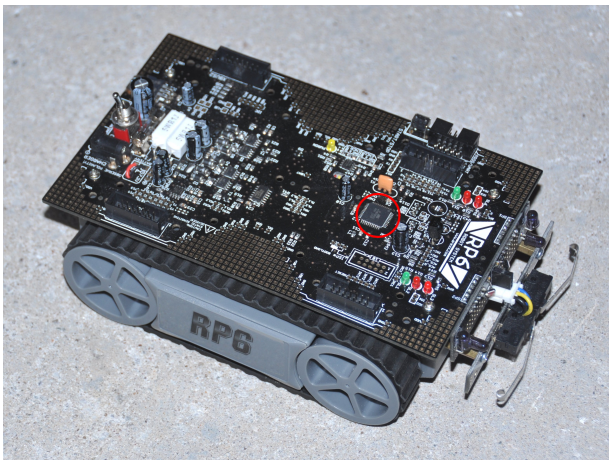
Im Gegensatz zu einem Allzweck-Computer mit allgemeinen Ein- und Ausgabegeräten wie Tastatur und Bildschirm erfüllt ein eingebettetes System eine ganz bestimmte Aufgabe. Es sind in der Regel keine Mechanismen vorgesehen, mit denen der Benutzer das System selbst prüfen oder reparieren kann. Stattdessen erwartet der Benutzer, daß das eingebettete System in allen Anwendungssituationen absolut zuverlässig arbeitet.

Aus diesen Gründen gelten bei der Programmierung eingebetteter Systeme besonders hohe Anforderungen an die Qualität der Software.

0.2 Beispiele

Täglich benutzen wir zahlreiche eingebettete Systeme, häufig ohne uns dessen bewußt zu sein: Armbanduhren, Mobiltelefone, Haushaltsgeräte, Automobile, Aufzüge, Unterhaltungselektronik, Flugzeuge, ...

- Viele aktuelle Kaffeemaschinen sind elektronisch gesteuert und bieten die Wahl zwischen mehreren Getränkeprogrammen.
- Aktuelle Durchlauferhitzer sind elektronisch geregelt und erlauben es, zahlreiche Parameter zu konfigurieren.
- Mit dem Chronos eZ430 (Bild rechts) bietet der Chip-Hersteller Texas Instruments eine Armbanduhr an, die man mit Hilfe eines PC selbst programmieren kann.
- Der unten abgebildete Roboter Robby RP6 von Arexx wird von einem selbst programmierbaren Mikro-Controller (rot markiert) gesteuert.



- Einige Barcode-Lesegeräte (z. B. das hier abgebildete Intermec CK1) sind voll ausgestatte, selbst programmierbare Computer mit Betriebssystem.

- Das in Velbert ansässige Unternehmen TFC stellt Flugzeugkabinensimulatoren her, die von bis zu ca. 30 vernetzen Industrie-PCs gesteuert werden. Nicht nur zahlreiche Komponenten eines derartigen Simulators, sondern auch der Simulator als Ganzes kann als ein eingebettetes System aufgefaßt werden.

Das Bild zeigt einen TFC-Kabinensimulator für einen Airbus A320 für den Kunden Czech Airlines (CSA) während der Kalibrierung des Motion-Systems.

- Der Mähroboter R40Li von GARDENA mäht (laut Werbung) selbständig Rasenflächen bis maximal $400\,m^2$ „nach dem Zufallsbewegungsprinzip [...], ohne Streifenbildung“.
- Als eine Art Umkehrung kann man ein Smartphone bereits wieder als ein Allzweckgerät betrachten, das aus dem eingebetteten System „Mobiltelefon“ hervorgegangen, aber selbst kein eingebettetes System mehr ist.

0.3 Ziele dieser Veranstaltung

Die im Modulhandbuch für die Bachelorstudiengänge Mechatronik und Informationstechnologie [KIA/KIS/-grundständig] der Hochschule Bochum, Campus Velbert/Heiligenhaus (Stand: 7. März 2012, Seite 50) aufgeführten Ziele dieser Veranstaltung lauten:

> Die Studierenden kennen die Besonderheiten und Werkzeuge der Software-Entwicklung für eingebettete Systeme und sind in der Lage, anhand industrietypischer Anforderungen Software für eingebettete Systeme zu entwickeln.

Konkret gehört dazu:

- wissen, was machbar ist
- das Maximum aus der Hardware herausholen
- robuste, wartbare Programme schreiben

Die Überprüfung der Lernziele erfolgt durch eine Projektarbeit mit Kolloquium. Die Projektarbeit besteht darin, für ein selbst gewähltes eingebettetes System eine industrietypische Anforderung selbständig umzusetzen und die Ergebnisse des Projekts in industrietypischer Weise zu präsentieren. Damit ist keine

Produktpräsentation gegenüber Endkunden gemeint, sondern eine Präsentation unter Technikern, z. B. gegenüber neuen Kollegen, die gleichzeitig die erreichten Ergebnisse dokumentiert.

Eingebettete Systeme müssen in erster Linie zuverlässig sein. Ein wesentliches Bewertungskriterium ist daher die Qualität der Software.

Beispiele für mögliche Projektarbeiten:

- RP6 mit Einplatinen-Computer und Vektor-Algorithmus als „Rasenmäher“
- 3d-Indoor-Positioning-System für Quadcopter
- Lichtschranken-Meßstrecke zur Anwendung im Physik-Praktikum der Hochschule
- Verbesserungen an Flugzeugkabinensimulatoren (reales Industrieprojekt)
- Scheibenwischersteuerung (für ein reales Fahrzeug)
- Haus-Informationssystem

1 Programmieren in C

1.1 Ausdrücke und Anweisungen

Das klassische „Hello, world!“-Programm in modernem C (Datei: hello-1.c) lautet:

```
#include <stdio.h>

int main (void)
{
  printf ("Hello,␣world!\n");
  return 0;
}
```

In Hinblick auf Optimierung der Qualität unserer Software werden wir im folgenden die Bestandteile des Programms vollständig analysieren.

- Die Präprozessor-Direktive **#include** <stdio.h> bindet den Inhalt der Datei stdio.h, die sich in einem Standard-Verzeichnis befindet (angezeigt durch die spitzen Klammern), als Text in den Quelltext ein. Dieser Mechanismus wurde im vergangenen Wintersemester in der Veranstaltung „Angewandte Informatik“ ausführlich erklärt.
- Bei **int** main (**void**) mit einem nachfolgenden Block in geschweiften Klammern handelt es sich um die Definition einer Funktion. Dies wird in Abschnitt 1.6 ausführlich behandelt.
- Zwischen den geschweiften Klammern befinden sich, jeweils mit einem Semikolon abgeschlossen, zwei *Anweisungen*.
- Innerhalb der Anweisungen kommen die folgenden *Ausdrücke* vor:
  - die String-Konstante "Hello,␣world!\n",
  - der Funktionsaufruf printf ("Hello,␣world!\n"),
  - die Konstante 0 als Argument der **return**-Anweisung.

Die Begriffe „Ausdruck“ und „Anweisung“ sind in vielen Programmiersprachen zentrale Begriffe.

In C gilt für Ausdrücke:

- Jeder Ausdruck hat einen *Wert* und einen *Typ*.
- Eine *Konstante* ist ein Ausdruck:
  - "Hello,␣world!\n" ist eine Konstante vom Typ *String* – „Zeichenkette“. (Tatsächlich ist „String“ in C kein elementarer Typ. Dies wird in Abschnitt 1.8 ausführlich behandelt.)

  - 0 ist eine Konstante vom Typ *Integer* – „ganze Zahl“.

- Funktionsaufrufe sind Ausdrücke. Sie bestehen aus dem Namen der Funktion, gefolgt von einem Klammerpaar. Wenn man der Funktion Argumente übergeben möchte, stehen diese, durch Kommata getrennt, zwischen den Klammern. Der Funktionsaufruf printf ("Hello,␣world!\n") hat – aus dem Beispielquelltext allein heraus nicht erkennbar – den Typ Integer.

Für Anweisungen gilt:

- Jede Anweisung wird mit einem Semikolon abgeschlossen.
- Ein Ausdruck ist eine Anweisung. Hierbei wird der Wert des Ausdrucks ignoriert. Insbesondere erzeugt der Funktionsaufruf printf ("Hello,␣world!\n") einen Wert vom Typ Integer, der aber im Beispielprogramm ignoriert wird.
- Eine spezielle Anweisung ist die *leere Anweisung*. Sie wird durch „nichts“ dargestellt. In einem Programm sieht man nur das Semikolon.
- Eine weitere spezielle Anweisung ist die **return**-Anweisung. Sie beendet die aktuelle Funktion und gibt an die aufrufende Funktion einen Wert zurück, der dem **return** als Argument übergeben wird. Insbesondere ist **return** *kein* Funktionsaufruf und *kein* Ausdruck. Das Argument von **return** ist ein Ausdruck. Es hat einen Typ und einen Wert, **return** selbst hingegen hat dies nicht.
- Es kommt gelegentlich vor, daß man mehrere Anweisungen ausgeführt haben möchte, wo nur eine einzige Anweisung erlaubt ist. In solch einem Fall kann man die Anweisungen mit geschweiften Klammern zu einem *Anweisungsblock* zusammenfassen. Hinter dem Block steht *kein* zusätzliches Semikolon.

Beide Listen sind unvollständig. Im folgenden werden weitere Arten von Ausdrücken und Anweisungen eingeführt werden.

Das folgende Beispielprogramm (Datei statements.c) ist ein gültiges, wenn auch nicht besonders sinnvolles C-Programm:

```
int main (void)
{
  42;
  { 137; }
  {;}
  {}
  "Hello,␣world!\n";
  return 0;
  ;
  return 1;
  return "Hello,␣world!\n";
}
```

Es besteht aus

- der Integer-Konstanten 42,
  die als Ausdruck, dessen Wert ignoriert wird, eine gültige Anweisung darstellt,
- dem Anweisungsblock { 137; }, der eine einzige Anweisung enthält,
  die ebenfalls aus einem Ausdruck besteht, dessen Wert ignoriert wird,
- einem weiteren Anweisungsblock, der eine leere Anweisung enthält,
- einem weiteren Anweisungsblock, der nichts enthält,
- der String-Konstanten "Hello,␣world!\n", deren Wert wiederum ignoriert wird,
- der **return**-Anweisung mit dem Argument 0. Hiermit wird das Programm beendet.

Danach folgen – syntaktisch korrekt, aber wenig sinnvoll – weitere Anweisungen, die nicht mehr ausgeführt werden:

- eine weitere leere Anweisung,
- eine weitere **return**-Anweisung mit Argument 1,
- eine weitere **return**-Anweisung mit Argument "Hello,␣world!\n".

Daraus, daß die letzte Zeile vom Compiler – wenn auch mit Warnung – akzeptiert wird, können wir bereits erkennen, daß es sich bei String nicht um einen elementaren Typ handelt, sondern um etwas, was sich implizit in eine Integer umwandeln läßt. (Genaugenommen handelt es sich um einen Pointer. Dies wird in Abschnitt 1.8 ausführlich behandelt.)

1.2 Seiteneffekte

Wenn ein Ausdruck nicht nur einen Wert zurückliefert, sondern zusätzlich noch etwas anderes bewirkt, dann heißt dieses andere ein *Seiteneffekt*.

Seiteneffekte sind – speziell in C – meistens gewollt. Sie können jedoch auch ungewollt auftreten und schwer lokalisierbare Fehler bewirken. Um dies zu vermeiden, sollte man sich als Programmierer der möglichen Seiteneffekte der verwendeten Ausdrücke immer bewußt sein.

Der Funktionsaufruf printf() hat einen gewünschten Seiteneffekt, nämlich die Ausgabe der übergebenen Parameter. Daneben hat der Funktionsaufruf printf() noch einen „Haupteffekt“, nämlich den zurückgelieferten Wert: die Anzahl der ausgegebenen Zeichen.

Meistens wird man den von printf() zurückgegebenen Wert ignorieren; hier ist der Seiteneffekt der Funktion wichtiger als ihr „Haupteffekt“. Sollten allerdings Menschenleben davon abhängen, ob die Zeichen tatsächlich geschrieben wurden – z. B. bei der Steuerung einer Bremse eines Fahrzeuges –, tut man gut daran, den zurückgegebenen Wert zu prüfen, um auf Übertragungsfehler reagieren zu können.

Dort, wo der Wert eines Ausdrucks ignoriert wird, sind nur Ausdrücke mit Seiteneffekten sinnvoll.

Für die bisher kennengelernten Ausdrücke gilt:

- Eine Konstante hat keinen Seiteneffekt.
- Ein Funktionsaufruf kann einen Seiteneffekt haben.
  Ob dies der Fall ist, hängt von der aufgerufenen Funktion ab.

Wenn wir von Ausdrücken auf beliebige Anweisungen verallgemeinern, sprechen wir von *Effekten*:

- Ein als Anweisung verwendeter Ausdruck hat genau dann einen Effekt,
  wenn er einen Seiteneffekt hat.
- Die leere Anweisung hat keinen Effekt.
- Die **return**-Anweisung hat einen Effekt.
- Anweisungen, die nach einer **return**-Anweisung erfolgen, haben keinen Effekt.
- Ein Anweisungsblock hat genau dann einen Effekt,
  wenn mindestens eine darin enthaltene Anweisung einen Effekt hat.

Wenn man einen Funktionsaufruf als einen speziellen, anderswo definierten Anweisungsblock auffaßt, dann hat der Funktionsaufruf genau dann einen Seiteneffekt, wenn mindestens eine im Anweisungsblock der Funktion enthaltene Anweisung einen Effekt hat.

1.3 Funktionen aufrufen

Im Vergleich mit anderen Sprachen kennt C nur sehr wenig „Compiler-Magie“, sondern man kann *alle* Funktionen unter Verwendung dokumentierter Sprachelemente selbst schreiben. Wie dies geht, wird in Abschnitt 1.6 ausführlich behandelt.

Um überhaupt ein lauffähiges C-Programm zu erhalten, müssen wir jedoch eine Funktion selbst schreiben, nämlich die Funktion main(). Wir verwenden die in den o. a. Beispielprogrammen vorgestellte Schreibweise *vorübergehend* als „Magie“. Was die Schreibweise im einzelnen bedeutet, wird in Abschnitt 1.6 ausführlich behandelt.

Auch die Funktion printf() wurde irgendwo „selbst geschrieben“, nämlich in der zusammen mit dem Betriebssystem ausgelieferten C-Bibliothek (unter Unix: Datei libc.a). Wenn uns der Quelltext der C-Bibliothek zur Verfügung steht, können wir dort nachlesen, wie man Funktionen in der Art von printf() selbst schreibt. (Speziell printf() ist allerdings eher kompliziert und eignet sich daher nur bedingt als Vorbild für eigene Funktionen.)

Wie man printf() in eigenen Programmen *verwenden* kann, ist in Kapitel 3 des Unix-Handbuchs dokumentiert. (In der Unix-Shell: `man 3 printf` eingeben. Kurzschreibweise: `printf(3)`)

main() ist diejenige Funktion in unserem Programm, die vom Betriebssystem aufgerufen wird, der sog. Einsprungpunkt unseres Programms. Für jeglichen Aufruf irgendwelcher anderer Teile des Programms sind wir selbst zuständig. (Die Bezeichnung „wir selbst“ schließt von uns verwendete Bibliotheken mit ein.)

Das nächste Beispielprogramm (Datei hello-2.c) illustriert, wie leicht es in C ist, einen Absturz zu provozieren:

```
#include <stdio.h>

int main (void)
{
  printf (42);
  return 0;
}
```

Die Funktion printf() erwartet als ersten Parameter einen Ausdruck vom Typ String. Wir übergeben stattdessen einen Ausdruck vom Typ Integer. Wir erhalten vom Compiler zwar eine Warnmeldung; er erzeugt aber trotzdem ein ausführbares Programm. Die Integer-Konstante wird dabei implizit in eine String-Konstante – genauer: einen Zeiger auf den Text – umgewandelt.

Wenn man das Program laufen läßt, greift es lesend auf Speicherzellen außerhalb des ihm zugänglichen Speicherbereichs zu. In Umgebungen mit Speicherschutzmechanismen beendet das Betriebssystem das Programm und erzeugt eine Fehlermeldung (unter Unix: Speicherschutzverletzung – segmentation fault). In einer Umgebung ohne Speicherschutzmechanismus gibt das Programm den quasi zufälligen „Text“ aus, der sich an den entsprechenden Speicherzellen befindet. Je nachdem, was der Text bewirken soll, kann dies fatale Folgen haben. (Man denke z. B. an einen Steuerbefehl an ein Flugzeugtriebwerk.)

Das folgende Beispiel (Datei hello-3.c) zeigt, wie man Zahlenwerte ausgeben kann:

```
#include <stdio.h>

int main (void)
{
  printf ("%d\n", 42);
  return 0;
}
```

Die sog. *Formatangabe* %d als Bestandteil des ersten an printf() übergebenen Parameters wird bei der Ausgabe durch die Textdarstellung des zweiten Parameters ersetzt. Die Abkürzung %d steht dabei für eine ganze Zahl in dezimaler Schreibweise.

Wie die Beispiele hello-3a.c und hello-4.c illustrieren, ist es in C legal, zusätzliche (überflüssige) Parameter an eine Funktion zu übergeben. Diese werden von der Funktion ignoriert. (Falls die Ausdrücke Seiteneffekte haben, hat die Übergabe allerdings dennoch einen Effekt.)

Beispielprogramm: hello-3a.c

```
#include <stdio.h>

int main (void)
{
  printf ("%d\n", 42, 137);
  return 0;
}
```

Beispielprogramm: hello-4.c

```
#include <stdio.h>

int main (void)
{
  printf ("%d\n", 42, "Hello, world!\n", 137);
  return 0;
}
```

Im Beispiel hello-3b.c

```
#include <stdio.h>

int main (void)
{
  printf ("%d\n", "Hello, world!\n");
  return 0;
}
```

wird an eine Funktion eine String-Konstante übergeben, obwohl ein Integer-Ausdruck erwartet wurde. Dies hat wieder eine implizite Umwandlung (und eine Warnung) zur Folge. Die ausgegebene Zahl ist die Speicheradresse des Textes.

Das Beispiel hello-3c.c

```
#include <stdio.h>

int main (void)
{
  printf ("%d\n", printf ("Hello, world!\n"));
  return 0;
}
```

illustriert, daß die Funktion printf() tatsächlich einen Integer-Wert zurückgibt, nämlich die Anzahl der ausgegebenen Zeichen, hier also 14. (Die Zeichenfolge \n am Ende der String-Konstanten steht für ein einzelnes Zeichen, nämlich eine Zeilenschaltung.)

1.4 Operatoren

Die Programmiersprache C kennt stellt eine große Anzahl von *Operatoren* zur Verfügung. Die in der folgenden kleinen Auswahl gelisteten Operatoren sollten Ihnen von früheren Veranstaltungen her bekannt sein. (foo ist eine metasyntaktische Variable.)

Unäre Operatoren: foo() −foo foo−− foo++ −−foo ++foo

Binäre Operatoren: + − ∗ / % & | ^ << >> %

Insbesondere zählt der Aufruf einer Funktion als Anwendung eines unären Operators auf die Funktion.

(Es gibt auch einen ternären Operator, d. h. einen Operator mit drei Operanden. Dieser wird in einem nachfolgenden Abschnitt ausführlich behandelt werden.)

Ein binärer Operator verbindet zwei *Operanden* zu einem Ausdruck.

Beispiel: Die Summe 23 + 19 zweier ganzer Zahlen ist ein Ausdruck vom Typ Integer mit dem Wert 42.

Das in der Datei mathe-1.c vorgestellte Beispielprogramm ist gleichwertig zu hello-3.c:

```
#include <stdio.h>

int main (void)
{
  printf ("%d\n", 23 + 19);
  return 0;
}
```

Die unären Operatoren −− und ++ haben Seiteneffekte. −− bewirkt ein Dekrementieren (1 subtrahieren), ++ ein Inkrementieren (1 addieren) des Operanden. Diese Operatoren können daher nicht auf Werte, sondern nur auf Variable angewendet werden. Dies ist der Grund, weshalb das folgende Beispielprogramm (Datei: mathe-1a.c) nicht compiliert werden kann:

```
#include <stdio.h>

int main (void)
{
  int foo = 7;
  printf ("%d\n", --foo++);
  return 0;
}
```

(Variable werden im nächsten Abschnitt ausführlich behandelt.)

Wenn der Operator vorangestellt ist, ist der Rückgabewert die bereits dekrementierte bzw. inkrementierte Variable. Wenn der Operator nachgestellt wird, ist der Rückgabewert der vorherige Wert der Variablen.

1.5 Variable

Eine Variablendeklaration in C besteht aus der Angabe des Typs, gefolgt von einem Namen und einem optionalen Initialisierer.

```
1  #include <stdio.h>
2
3  int a, b = 3;
4
5  int main (void)
6  {
7    printf ("a = %d, b = %d\n", a, b);
8    static int a = 7;
9    int b = 5;
10   printf ("a = %d, b = %d\n", a, b);
11   a = b = 12;
12   printf ("a = %d, b = %d\n", a, b);
13   return 0;
14 }
```

In Zeile 3 werden zwei *globale Variable* deklariert. Die zweite wird explizit auf den Wert 3 initialisiert, die erste implizit auf den Wert 0. Diese beiden Werte werden in Zeile 7 ausgegeben.

Zeile 8 und 9 deklarieren *lokale Variable*. Diese sind nur innerhalb des umgebenden Blocks, hier der Funktion main(), sichtbar. Da die Variablen dieselben Namen haben wie die globalen Variablen in Zeile 3, sind letztere nicht mehr sichtbar. Zeile 10 gibt entsprechend die Werte 7 und 5 der neuen Variablen aus.

Durch das vorangestellte Schlüsselwort **static** wird die lokale Variable a in demselben Speicherbereich angelegt wie die globalen Variablen in Zeile 3. Genau wie globale Variable wird diese sog. *statische Variable* nur bei Programmstart initialisiert und behält ihren Wert auch zwischen zwei Aufrufen der umgebenden Funktion.

Im Gegensatz dazu heißt die lokale Variable b eine *automatische Variable*. Sie wird zu Beginn des Blocks, also bei jedem Aufruf der Funktion main() erzeugt und initialisiert. Ohne den Initialisierer bliebe sie – im Gegensatz zu globalen oder statischen Variablen – uninitialisiert, hätte also einen zufälligen Anfangswert. Da dieser zufällige Anfangswert häufig Null ist, bleiben nicht initialisierte Variable oft lange unerkannt und können fatale Folgen haben. (Wenn man `gcc` mit den Optionen `-O` (Optimierung) und `-Wall` (alle Warnungen) aufruft, erkennt er derartige Situationen und erzeugt eine Warnung.)

In Zeile 11 erfolgt eine Zuweisung des Wertes des Ausdrucks b = 12 an die Variable a. Der Ausdruck b = 12 ist seinerseits eine Zuweisung des Wertes 12 an die Variable b.

Der *Zuweisungsoperator* = ist in C ein „normaler“ binärer Operator mit dem Seiteneffekt, daß er dem ersten Operanden den Wert des zweiten zuweist. Der Wert des Operator-Ausdrucks ist der zugewiesene Wert.

Neben = kennt C noch zahlreiche weitere Zuweisungsoperatoren:

= += −= ∗= /= %= &= |= ^= <<= >>=

Alle diese Operatoren sind binär und haben einen Seiteneffekt. Das dem = vorangestellte arithmetische Operatorsymbol wird jeweils auf die beiden Operanden angewendet. Als Seiteneffekt wird dann das Rechenergebnis dem ersten Operanden, der eine Variable sein muß, zugewiesen.

Beispiele:

a + 7	ergibt a + 7 (und hat keinen Seiteneffekt).
a = 7	ergibt 7 (und weist a den Wert 7 zu).
a += 7	ergibt a + 7 (und weist a diesen Wert zu).

1.6 Funktionen schreiben

Aus historischen Gründen kennt C mehrere Schreibweisen für die Deklaration von Funktionen.

Programmbeispiel: functions-1.c

Die moderne Schreibweise (ISO-C) lautet:

```
#include <stdio.h>

int answer (void)
{
  return 42;
}

void foo (void)
{
  printf ("%d\n", answer ());
}

void add_verbose (int a, int b)
{
  printf ("%d␣+␣%d␣=␣%d\n",
          a, b, a + b);
}


int main (void)
{
  foo ();
  add_verbose (2, 3);
  return 0;
}
```

Programmbeispiel: functions-2.c

Die ältere Schreibweise (K&R-C) lautet:

```
#include <stdio.h>

answer ()
{
  return 42;
}

void foo ()
{
  printf ("%d\n", answer ());
}

void add_verbose (a, b)
  int a, b;
{
  printf ("%d␣+␣%d␣=␣%d\n",
          a, b, a + b);
}

main ()
{
  foo ();
  add_verbose (2, 3);
  return 0;
}
```

Die wichtigsten Unterschiede sind:

- In der ISO-Schreibweise stehen die typisierten Funktionsparameter zwischen den Klammern. Diese Schreibweise einer Funktionsdeklaration nennt man einen *Prototypen*.

  In der K&R-Schreibweise stehen zwischen den Klammern nur die Namen der Parameter. Die Typen werden in nachfolgenden Deklarationen vor der öffnenden Klammer des Blocks ergänzt.

  Dieser Unterschied ist nicht nur optischer Natur. Bei Verwendung der K&R-Schreibweise führt der Compiler beim Funktionsaufruf *keine* Überprüfung der Anzahl oder der Typen der Parameter durch. Dies geschieht nur bei Verwendung von ISO-Prototypen.

  Die ISO-Schreibweise ist daher eindeutig vorzuziehen.

- Um Verwechselung mit einer unbestimmten K&R-Parameterliste zu vermeiden, wird eine leere Parameterliste in ISO-C durch das Schlüsselwort **void** spezifiziert.

- In der K&R-Schreibweise darf der Funktionsrückgabetyp **int** weggelassen werden; er wird dann implizit angenommen.

Gelegentlich findet man auch Kombinationen von K&R-C mit ISO-Prototypen (Datei: functions-3.c):

```
#include <stdio.h>

void add_verbose (int, int);
int main (void);

void add_verbose (a, b)
  int a, b;
{
  printf ("%d␣+␣%d␣=␣%d\n", a, b, a + b);
}

int main ()
{
  add_verbose (2, 3);
  return 0;
}
```

Wenn man eine Bibliothek schreibt, müssen die vorangestellten Prototypen (ohne Funktionsrumpf) ohnehin geschrieben werden. Diese werden dann in eine separate *Header-Datei* (.h-Datei) ausgelagert, wobei die K&R-Deklarationen in der .c-Datei häufig unverändert gelassen werden. Aus diesem Grunde ist die oben vorgestellte kombinierte K&R+ISO-Schreibweise trotz ihrer Veraltung in existierenden Programmen relativ häufig anzutreffen.

Bei **void** handelt es sich um einen Datentyp. Im Gegensatz zu z. B. **int**, das für ganze Zahlen steht, steht **void** für „nichts“.

Von Ausdrücken zurückgegebene **void**-Werte *müssen* ignoriert werden. (Von Ausdrücken zurückgegebene Werte anderer Typen *dürfen* ignoriert werden.)

1.7 Zeiger

In C können an Funktionen grundsätzlich nur Werte übergeben werden. Vom Funktionsrückgabewert abgesehen, hat eine C-Funktion keine Möglichkeit, dem Aufrufer Werte zurückzugeben.

Es ist dennoch möglich, eine C-Funktion aufzurufen, um eine Variable (oder mehrere) auf einen Wert zu setzen. Hierfür übergibt man der Funktion die *Speicheradresse* der Variablen als Wert. Der Wert ist ein *Zeiger* auf die Variable.

Wenn einem Zeiger der unäre Operator $*$ vorangestellt wird, ist der resultierende Ausdruck diejenige Variable, auf die der Zeiger zeigt. In Deklarationen wird dasselbe Symbol dem Namen vorangestellt, um anstelle einer Variablen des genannten Typs eine Variable vom Typ „Zeiger auf Variablen des genannten Typs“ zu deklarieren.

Umgekehrt wird der unäre Operator & einer Variablen vorangestellt, um einen Ausdruck vom Typ „Zeiger auf Variablen dieses Typs“ mit dem Wert „Speicheradresse dieser Variablen“ zu erhalten.

Beispielprogramm: pointers-1.c

```
#include <stdio.h>

void calc_answer (int *a)
{
  *a = 42;
}

int main (void)
{
  int answer;
  calc_answer (&answer);
  printf ("The answer is %d.\n", answer);
  return 0;
}
```

1.8 Arrays und Strings

1.8.1 Arrays und Zeiger – i. w. dasselbe

In C ist es möglich, mit einem Zeiger Arithmetik zu betreiben, so daß er nicht mehr auf die ursprüngliche Variable zeigt, sondern auf ihren Nachbarn im Speicher.

Solche Nachbarn gibt es dann, wenn mehrere Variablen gleichen Typs gemeinsam angelegt werden. Eine derartige Ansammlung Variabler gleichen Typs heißt *Array* (Feldvariable, Vektor).

Beispielprogramm: arrays-1.c

```
#include <stdio.h>

int main (void)
{
  int prime[5] = { 2, 3, 5, 7, 11 };
  int *p = prime, i = 0;
  while (i < 5)
    printf ("%d\n", *(p + i++));
  return 0;
}
```

Die initialisierte Variable prime ist ein Array von fünf ganzen Zahlen. Der Bezeichner prime des Arrays wird als Zeiger auf eine **int**-Variable verwendet. In diesem Sinne sind Arrays und Zeiger in C dasselbe.

p + i ist ein Zeiger auf den i-ten Nachbarn von *p. Durch Dereferenzieren *(p + i) erhalten wir den i-ten Nachbarn von *p selbst. Die von anderen Sprachen her bekannte Schreibweise p[i] für das i-te Element eines Arrays p ist in C eine Abkürzung für *(p + i), wobei man p gleichermaßen als Array wie als Zeiger auffassen kann.

Wenn wir uns dieser Schreibweise bedienen und anstelle des Zeigers p, der durchgehend den Wert prime hat, direkt prime verwenden, erhalten wir das Beispielprogramm arrays-2.c:

```
#include <stdio.h>

int main (void)
{
  int prime[5] = { 2, 3, 5, 7, 11 };
  int i = 0;
  while (i < 5)
    printf ("%d\n", prime[i++]);
  return 0;
}
```

(Zur Erinnerung: Das ++ in prime[i++] besagt, daß wir prime[i] an printf() übergeben und *danach* den Index i um 1 erhöhen.)

1.8.2 Strings

Ein wichtiger Spezialfall ist ein Array, dessen Komponenten den Datentyp **char** haben. In C ist **char** wie **int** eine ganze Zahl; der einzige Unterschied besteht darin, daß der Wertebereich von **char** daran angepaßt ist, ein Zeichen (Buchstabe, Ziffer, Satz- oder Sonderzeichen, engl. character) aufzunehmen. Ein typischer Wertebereich für den Datentyp **char** ist von −128 bis 127.

Ein Initialisierer für ein Array von **char**-Variablen kann direkt als Folge von Zeichen (Zeichenkette, engl. *String*) mit doppelten Anführungszeichen geschrieben werden. Jedes Zeichen initialisiert eine ganzzahlige Variable mit seinem ASCII-Wert. An das Ende eines in dieser Weise notierten Array-Initialisierers fügt der Compiler implizit einen Ganzzahl-Initialisierer für den Zahlenwert 0 an. Der Array-Initialisierer "Hello" ist also gleichbedeutend mit { 72, 101, 108, 108, 111, 0 }. (Die 72 steht für ein großes H, die 111 für ein kleines o. Man beachte die abschließende 0 am Ende!)

Ein String in C ist also ein Array von **char**s, also ein Zeiger auf **char**s, also ein Zeiger auf ganze Zahlen, deren Wertebereich daran angepaßt ist, Zeichen aufzunehmen.

Wenn bei der Deklaration eines Arrays die Länge aus dem Initialisierer hervorgeht, braucht diese nicht ausdrücklich angegeben zu werden. In diesem Fall folgt auf den Bezeichner nur das Paar eckiger Klammern und der Initialisierer.

Das Beispielprogramm arrays-3--.c zeigt, wie das Array durchlaufen werden kann, bis die Zahl 0 gefunden wird:

```
#include <stdio.h>

int main (void)
{
  char hello_world[] = "Hello,␣world!\n";
  int i = 0;
  while (hello_world[i] != 0)
    printf ("%d", hello_world[i++]);
  return 0;
}
```

Durch Verwendung von Pointer-Arithmetik und Weglassen der überflüssigen Abfrage != 0 erhalten wir äquivalente Beispielprogramm arrays-3.c:

```
#include <stdio.h>

int main (void)
{
  char hello_world[] = "Hello,␣world!\n";
  char *p = hello_world;
  while (*p)
    printf ("%d", *p++);
  return 0;
}
```

Durch die Formatangabe %d wird jedes Zeichen – korrektermaßen – als Dezimalzahl ausgegeben. Wenn wir stattdessen die Formatangabe %c verwenden (für *character*), wird für jedes Zeichen – ebenso korrektermaßen – sein Zeichenwert (Buchstabe, Ziffer, . . .) ausgegeben (Datei: arrays-3a.c):

```
#include <stdio.h>

int main (void)
{
  char hello_world[] = "Hello,␣world!\n";
  char *p = hello_world;
  while (*p)
    printf ("%c", *p++);
  return 0;
}
```

Dieses ist die in C übliche Art, eine Schleife zu schreiben, die nacheinander alle Zeichen in einem String bearbeitet.

Eine weitere Formatangabe %s dient in printf() dazu, direkt einen kompletten String bis ausschließlich der abschließenden 0 auszugeben.

Beispielprogramm: arrays-4.c

```
#include <stdio.h>

int main (void)
{
  char hello_world[] = "Hello, world!\n";
  printf ("%s", hello_world);
  return 0;
}
```

Anstatt als Array können wir die Variable hello_world auch als Zeiger deklarieren (Datei: arrays-4a.c):

```
#include <stdio.h>

int main (void)
{
  char *hello_world = "Hello, world!\n";
  printf ("%s", hello_world);
  return 0;
}
```

Allein die Formatspezifikation entscheidet darüber, wie die Parameter von printf() bei der Ausgabe dargestellt werden:

%d	Der Parameter wird als Zahlenwert interpretiert und dezimal ausgegeben.
%x	Der Parameter wird als Zahlenwert interpretiert und hexadezimal ausgegegeben.
%c	Der Parameter wird als Zahlenwert interpretiert und als Zeichen ausgegeben.
%s	Der Parameter wird als Zeiger interpretiert und als Zeichenfolge ausgegeben.

1.8.3 Unterschiede zwischen Arrays und Zeigern

Bei allen Gemeinsamkeiten gibt es doch zumindest einen Unterschied zwischen Arrays und Zeigern in C. Dieser zeigt sich, sobald man die Adresse einer Array- oder Zeigervariablen bildet.

Beispielprogramm: arrays-5.c

```
#include <stdio.h>

void foo (int *x)
{
  static int counter = 1;
  printf ("foo(%d): %d\n", counter++, *x);
}

int main (void)
{
  int a[3] = { 13, 14, 15 };
  int *b = a;
  foo (a);   <-- Adresse des Arrays, also der 13
  foo (&a);  <-- dasselbe wie a: Adresse der 13
  foo (b);   <-- dasselbe wie a: Adresse der 13
  foo (&b);  <-- Adresse von b
  return 0;
}
```

Bei einer Zeigervariablen unterscheidet man zwischen der Adresse &b der Variablen b selbst und der Adresse, auf die der Zeiger zeigt, was dasselbe ist wie der Wert der Zeigervariablen b.

Bei einer Array-Variablen a gibt es keine Adresse außer der der Adresse &a der Array-Variablen selbst, gleichbedeutend mit der Adresse &a[0] des ersten Elements des Arrays (mit Index 0). Eine Besonderheit in C ist, daß der Name der Array-Variablen auch ohne & für dieselbe Adresse steht. Hier wäre es konsequent gewesen, die Konstruktion &a zu verbieten, da man die Adresse einer Adresse – also eines Wertes im Gegensatz zu einer Variablen – nicht bilden kann.

1.8.4 Parameter des Hauptprogramms

Bisher haben wir das Hauptprogramm immer mit **int** main (**void**) eingeleitet. Tatsächlich bekommt das Hauptprogramm vom Betriebssystem zwei Parameter übergeben, durch die es auf die Kommandozeile zugreifen kann, über die es aufgerufen wurde:

Beispielprogramm: args-1.c

```
#include <stdio.h>

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

Der **int**-Parameter argc steht für die Anzahl der übergebenen Parameter. Der Parameter argv ist ein Zeiger auf Zeiger auf **char**-Variable, also ein Array von Arrays von **char**-Variablen, also ein Array von Strings. Gemäß Standard stehen die Array-Elemente von 0 bis ausschließlich argc für die in der Kommandozeile übergebenen Argumente, wobei das Argument Nr. 0 der Name ist, unter dem das Progamm selbst aufgerufen wurde:

```
$ ./args-1 foo bar baz
argc = 4
argv[0] = "./args-1"
argv[1] = "foo"
argv[2] = "bar"
argv[3] = "baz"
```

Insbesondere enthält argv[0] den genauen Pfad, unter dem das Programm aufgerufen wurde. Wenn sich das Programm in einem Standard-Pfad befindet und nur über args−1 aufgerufen wird, enthält auch argv[0] nur den String args−1.

Wenn man die Kommandozeilenparameter nicht benötigt, ist die verkürzte Schreibweise **int** main (**void**) insofern korrekt, als daß es in C zulässig ist, Parameter zu ignorieren. Beim Linken des Programms muß lediglich das Symbol main zur Verfügung gestellt werden. Das Betriebssystem übergibt dann verschiedene Parameter und ruft main() als Funktion auf. Was die Funktion mit den übergebenen Parametern macht, spielt aus Sicht des Betriebssystems keine Rolle.

Aus demselben Grund ist es *nicht* korrekt, main() mit anderen Parameter-Typen als den im Standard vorgesehenen auszustatten. Wenn mein Programm z. B. einen Integer- und einen String-Parameter erwartet, kann ich das *nicht* auf folgende Weise erreichen (Beispielprogramm: args-2-wrong.c):

```
#include <stdio.h>

int main (int arg1, char *arg2)
{
  printf ("arg1 = %d\n", arg1);
  printf ("arg2 = \"%s\"\n", arg2);
  return 0;
}
```

Das Programm läßt sich zwar – mit Warnung – compilieren, das Betriebssystem übergibt aber trotzdem die oben beschriebenen Parameter **int** argc und **char** **argv, die nun vom Programm inkorrekt interpretiert werden:

```
$ gcc -std=c99 -Wall -O args-2-wrong.c -o args-2-wrong
args-2-wrong.c:3: warning: second argument of 'main' should be 'char **'
$ ./args-2-wrong 42 foobar
arg1 = 3
arg2 = "íÐ°¿öÐ°¿ùÐ°¿"
```

Die Namen der Parameter dürfen hingegen anders lauten als argc und argv. Hier z. B. wurde **int** arg1 ohne Warnung anstelle von **int** argc akzeptiert, und tatsächlich hat arg1 den Wert von argc bekommen.

Um nun wirklich numerische Parameter entgegenzunehmen, muß ein Programm die in argv übergebenen String-Parameter in Zahlen umwandeln, z. B. mit Hilfe der Funktion sscanf() (siehe `sscanf(3)`) (Beispielprogramm: args-2-correct.c):

```
#include <stdio.h>

int main (int argc, char **argv)
{
  int arg1;
  char *arg2 = argv[2];
  sscanf (argv[1], "%d", &arg1);
  printf ("arg1 = %d\n", arg1);
  printf ("arg2 = \"%s\"\n", arg2);
  return 0;
}
```

```
$ gcc -std=c99 -Wall -O args-2-correct.c -o args-2-correct
$ ./args-2-correct 42 foobar
arg1 = 42
arg2 = "foobar"
```

Zusätzlich zu argc gibt es noch einen zweiten Mechanismus, über den das Betriebssystem dem Programm die Anzahl der Parameter mitteilt. Auf die gleiche Weise, wie das Ende eines Strings durch den Zahlenwert 0 markiert wird (nicht zu verwechseln mit der Ziffer '0'), wird das Ende des Arrays argv durch einen Zeiger markiert, der auf *nichts* zeigt. Zu diesem Zweck wird die Speicheradresse mit dem Zahlenwert 0 reserviert; ein Zeiger zeigt genau dann auf etwas, wenn er „ungleich Null“ ist.

Zu dem Beispielprogramm args-1.c gibt es also die folgende, gleichwertige Alternative args-3.c:

```
#include <stdio.h>

int main (int argc, char **argv)
{
  int i = 0;
  char **p = argv;
  printf ("argc = %d\n", argc);
  while (*p)
    printf ("argv[%d] = \"%s\"\n", i++, *p++);
  return 0;
}
```

Für die Adresse mit dem Zahlenwert 0 gibt es in C die Konstante NULL.

1.9 Strukturen

In vielen Situationen ist es sinnvoll, mehrere Daten von nicht notwendigerweise gleichem Typ zu einer Einheit zusammenzufassen. In C heißt eine derartige Einheit **struct**.

Beispielprogramm: structs-1.c

```
#include <stdio.h>

typedef struct
{
  char day, month;
  int year;
}
date;

int main (void)
{
  date today = { 11, 4, 2012 };
  printf ("%d.%d.%d\n", today.day, today.month, today.year);
  return 0;
}
```

Der auf diese Weise definierte neue Datentyp – hier: date – kann genauso wie vordefinierte Datentypen **int**, **char** usw. verwendet werden.

Wenn wir eine Variable today vom Datentyp date vorliegen haben, können wir auf die Komponenten des **struct**s zugreifen, indem wir an die Variable einen Punkt und den Namen day, month bzw. year der Komponente anhängen. Die **struct**-Komponente today.day ist dann eine völlig normale Variable vom Typ **char**.

Die Benutzung von **struct**s lohnt sich dann, wenn zusammengehörende Daten gemeinsam an eine Funktion übergeben werden sollen (Beispielprogramm: structs-2.c):

```
#include <stdio.h>

typedef struct
{
  char day, month;
  int year;
}
date;

void get_date (date *d)
{
  (*d).day = 11;
  (*d).month = 4;
  (*d).year = 2012;
}

int main (void)
{
  date today;
  get_date (&today);
  printf ("%d.%d.%d\n", today.day, today.month, today.year);
  return 0;
}
```

Damit die Funktion get_date in die Datums-Variable schreiben kann, müssen wir ihr nur eine Adresse &today übergeben – anstelle von drei Adressen für die Variablen day, month und year.

Die Klammern bei (*d).day sind notwendig, weil der Komponentenzugriffsoperator . eine höhere Priorität besitzt als der unäre Dereferenzoperator *. Der Ausdruck *d.day stünde also für das, worauf eine in der **struct**-Variablen d enthaltene Komponente day, die ein Zeiger sein muß, zeigt. (In dieser Situation wäre das ein Fehler, weil d keine **struct**-Variable, sondern ein Zeiger auf eine **struct**-Variable ist und day keine Zeiger-Variable, sondern eine **char**-Variable.)

Weil die Schreibweise (∗foo).bar relativ umständlich ist und andererseits die Konstruktion „Komponente einer **struct**-Variablen, auf die der Zeiger foo zeigt“ sehr häufig auftritt, stellt C das Symbol foo−>bar als Abkürzung für (∗foo).bar zur Verfügung – analog zur Indexdereferenzierung foo[42] als Abkürzung für ∗(foo + 42).

Durch Verwendung dieser Abkürzung ändert sich das Beispielprogramm wie folgt (Datei: structs-3.c):

```
#include <stdio.h>

typedef struct
{
  char day, month;
  int year;
}
date;

void get_date (date *d)
{
  d->day = 11;
  d->month = 4;
  d->year = 2012;
}

int main (void)
{
  date today;
  get_date (&today);
  printf ("%d.%d.%d\n", today.day,
          today.month, today.year);
  return 0;
}
```

Wie bei Funktionen gibt es auch bei **struct**s deutliche Unterschiede zwischen den verschiedenen Varianten der Programmiersprache C.

Programmbeispiel: structs-1.c

Die moderne Schreibweise (ISO-C) lautet:

```
#include <stdio.h>

typedef struct
{
  char day, month;
  int year;
}
date;

int main (void)
{
  date today = { 11, 4, 2012 };
  printf ("%d.%d.%d\n", today.day,
          today.month, today.year);
  return 0;
}
```

Die ältere Schreibweise (K&R-C) lautet:

```
#include <stdio.h>

struct date
{
  char day, month;
  int year;
};

int main (void)
{
  struct date today = { 11, 4, 2012 };
  printf ("%d.%d.%d\n", today.day,
          today.month, today.year);
  return 0;
}
```

Die wichtigsten Unterschiede sind:

- In K&R-C gibt es einen eigenen Namensraum für **struct**-Deklarationen. Ein **struct**-Typ mit dem Namen date wird immer als **struct** date geschrieben. Die Verwendung von date ohne **struct** ist etwas anderes. Insbesondere ist es zulässig, den Bezeichner date noch für etwas anderes zu verwenden.

- K&R-C erlaubt es, Zeiger auf *noch unbekannte* **struct**-Typen zu verwenden. So ist z. B. die Deklaration einer Zeigervariablen

  ```
  struct date *d;
  ```

  bereits *vor* der Deklaration des Typs **struct** date zulässig, solange nicht auf die (noch unbekannten) Komponenten der **struct** zugegriffen wird.

Der letztgenannte Unterschied ist der Grund, weshalb auch in ISO-C-Programmen in bestimmten Situationen auf die K&R-Schreibweise zurückgegriffen wird – siehe Abschnitt 1.11.

Aufgabe

Schreiben Sie eine Funktion inc_date (date *d),
die ein gegebenes Datum d unter Beachtung von Schaltjahren
auf den nächsten Tag setzt.

Lösung

Wir lösen die Aufgabe über den sog. *Top-Down-Ansatz* („vom Allgemeinen zum Konkreten“). Als besonderen Trick approximieren wir unfertige Programmteile zunächst durch einfachere, fehlerbehaftete. Diese fehlerhaften Programmteile sind in den untenstehenden Beispielen rot markiert. (In der Praxis würde man diese Zeilen unmittelbar durch die richtigen ersetzen; die fehlerhaften „Platzhalter“ sollten also jeweils nur für Sekundenbruchteile im Programm stehen. Falls man einmal tatsächlich einen Platzhalter für mehrere Sekunden oder länger stehen lassen sollte – z. B., weil an mehreren Stellen Änderungen notwendig sind –, sollte man ihn durch etwas Uncompilierbares (z. B. @@@) markieren, damit man auf jeden Fall vermeidet, ein fehlerhaftes Programm auszuliefern.)

Zunächst kopieren wir das Beispielprogramm structs-3.c und ergänzen den Aufruf der – noch nicht existierenden – Funktion inc_date() (Datei: incdate-1.c):

```
#include <stdio.h>

typedef struct
{
  char day, month;
  int year;
}
date;

void get_date (date *d)
{
  d->day = 11;
  d->month = 4;
  d->year = 2012;
}

int main (void)
{
  date today;
  get_date (&today);
  inc_date (&today);
  printf ("%d.%d.%d\n", today.day, today.month, today.year);
  return 0;
}
```

Als nächstes kopieren wir innerhalb des Programms die Funktion get_date() als „Schablone“ für inc_date() (Datei: incdate-2.c):

```c
#include <stdio.h>

typedef struct
{
  char day, month;
  int year;
}
date;

void get_date (date *d)
{
  d->day = 11;
  d->month = 4;
  d->year = 2012;
}

void inc_date (date *d)
{
  d->day = 11;
  d->month = 4;
  d->year = 2012;
}

int main (void)
{
  date today;
  get_date (&today);
  inc_date (&today);
  printf ("%d.%d.%d\n", today.day, today.month, today.year);
  return 0;
}
```

Da die Funktion jetzt existiert, ist der Aufruf nicht mehr fehlerhaft. Stattdessen haben wir jetzt eine fehlerhafte Funktion inc_date().

Im nächsten Schritt ersetzen wir die fehlerhafte Funktion durch ein simples Hochzählen der day-Komponente (Datei: incdate-3.c)

```c
void inc_date (date *d)
{
  d->day += 1;
}
```

Diese naive Vorgehensweise versagt, sobald wir den Tag über das Ende des Monats hinauszählen (Beispielprogramm: incdate-4.c). Dies reparieren wir im nächsten Schritt, wobei wir für den Moment inkorrekterweise annehmen, daß alle Monate 30 Tage hätten (Datei: incdate-5.c):

```c
void inc_date (date *d)
{
  d->day += 1;
  if (d->day > 30)
    {
      d->day = 1;
      d->month += 1;
      if (d->month > 12)
        {
          d->month = 1;
          d->year += 1;
        }
    }
}
```

Diese Version versagt in Monaten mit mehr oder weniger als 30 Tagen (Beispielprogramm: incdate-6.c).

Die Reparatur nehmen wir wieder stufenweise vor. Zunächst lagern wir die Konstante 30 in eine Funktion aus (Datei: incdate-7.c)

```
int days_in_month (date *d)
{
  return 30;
}

void inc_date (date *d)
{
  d->day += 1;
  if (d->day > days_in_month (d))
    {
      d->day = 1;
      d->month += 1;
      if (d->month > 12)
        {
          d->month = 1;
          d->year += 1;
        }
    }
}
```

Anschließend „reparieren“ wir die fehlerhafte Funktion, wobei wir zunächst das Problem der Schaltjahre aussparen (Datei: incdate-8.c):

```
int days_in_month (date *d)
{
  if (d->month == 4)
    return 30;
  else if (d->month == 6)
    return 30;
  else if (d->month == 9)
    return 30;
  else if (d->month == 11)
    return 30;
  else if (d->month == 2)
    return 28;
  else
    return 31;
}
```

Dieses Problem (Datei: incdate-9.c) reparieren wir durch Auslagern des Problems „Schaltjahr oder nicht?“ in eine zunächst fehlende (Datei: incdate-10.c) und danach fehlerhafte Funktion (Datei: incdate-11.c):

```
int is_leap_year (date *d)
{
  if (d->year % 4 == 0)
    return 1;
  else
    return 0;
}

int days_in_month (date *d)
{
  if (d->month == 4)
    return 30;
  else if (d->month == 6)
    return 30;
  else if (d->month == 9)
    return 30;
  else if (d->month == 11)
    return 30;
```

```
  else if (d->month == 2)
    {
      if (is_leap_year (d))
        return 29;
      else
        return 28;
    }
  else
    return 31;
}
```

Das nun vorliegende Programm arbeitet bereits für den julianischen Kalender sowie für alle Jahre von 1901 bis 2099 korrekt. Um durch 100 teilbare Jahre korrekt als Nicht-Schaltjahre zu erfassen (Datei: incdate-12.c) ergänzen wir nun die 100-Jahre-Regel des gregorianischen Kalenders (Datei: incdate-13.c):

```
int is_leap_year (date *d)
{
  if (d->year % 4 == 0)
    {
      if (d->year % 100 == 0)
        return 0;
      else
        return 1;
    }
  else
    return 0;
}
```

Um schließlich auch durch 400 teilbare Jahre korrekt als Schaltjahre zu erfassen (Datei: incdate-14.c) ergänzen wir nun als letztes die 400-Jahre-Regel des gregorianischen Kalenders. Damit ist die Aufgabe gelöst. Der vollständige Quelltext der Lösung (Datei: incdate-15.c) lautet:

```
#include <stdio.h>

typedef struct
{
  char day, month;
  int year;
}
date;

void get_date (date *d)
{
  d->day = 28;
  d->month = 2;
  d->year = 2000;
}

int is_leap_year (date *d)
{
  if (d->year % 4 == 0)
    {
      if (d->year % 100 == 0)
        {
          if (d->year % 400 == 0)
            return 1;
          else
            return 0;
        }
      else
        return 1;
    }
  else
```

```
    return 0;
}

int days_in_month (date *d)
{
  if (d->month == 4)
    return 30;
  else if (d->month == 6)
    return 30;
  else if (d->month == 9)
    return 30;
  else if (d->month == 11)
    return 30;
  else if (d->month == 2)
    {
      if (is_leap_year (d))
        return 29;
      else
        return 28;
    }
  else
    return 31;
}

void inc_date (date *d)
{
  d->day += 1;
  if (d->day > days_in_month (d))
    {
      d->day = 1;
      d->month += 1;
      if (d->month > 12)
        {
          d->month = 1;
          d->year += 1;
        }
    }
}

int main (void)
{
  date today;
  get_date (&today);
  inc_date (&today);
  printf ("%d.%d.%d\n", today.day, today.month, today.year);
  return 0;
}
```

Bemerkungen:

- Der Top-Down-Ansatz ist eine bewährte Methode, um eine zunächst komplexe Aufgabe in handhabbare Teilaufgaben zu zerlegen. Dies hilft ungemein, in längeren Programmen (mehrere Zehntausend bis Millionen Zeilen) die Übersicht zu behalten.
- Der Trick mit dem zunächst fehlerhaften Code hat den Vorteil, daß man jeden Zwischenstand des Programms compilieren und somit austesten kann. Er birgt andererseits die Gefahr in sich, die Übersicht über den fehlerhaften Code zu verlieren, so daß es dieser bis in die Endfassung schafft. Neben dem bereits erwähnten Trick uncompilierbarer Symbole haben sich hier Kommentare wie */*FIXME */* bewährt, auf die man seinen Code vor der Auslieferung der Endfassung noch einmal automatisch durchsuchen läßt.
- Allen an der Berechnung beteiligten Funktionen wurde hier ein Zeiger d auf die vollständige date-Struktur übergeben. Dies ist ein *objektorientierter Ansatz*, bei dem man die Funktionen als *Methoden*

der *Klasse* date auffaßt. (Von sich aus unterstützt die Sprache C – im Gegensatz zu z. B. C++ – keine Klassen und Methoden, sondern man muß diese bei Bedarf in der oben beschrieben Weise selbst basteln. Für eine fertige Lösung siehe z. B. die GObject-Bibliothek – http://gtk.org.)

Alternativ könnte man sich mit den zu übergebenden Parametern auf diejenigen beschränken, die in der Funktion tatsächlich benötigt werden, also z. B. **int** days_in_month (**int** month, **int** year) und **int** is_leap_year (**int** year). Damit wären die Funktionen allgemeiner verwendbar.

Welcher dieser beiden Ansätze der bessere ist, hängt von der Situation und von persönlichen Vorlieben ab.

Aufgabe

Schreiben Sie eine Funktion add_date (date *d, **int** days),
die auf ein gegebenes Datum d days Tage addiert.

(ohne Musterlösung)

Programmiertips

Einrückung. Ordnen Sie Ihren Quelltext möglichst übersichtlich an. Eine korrekte Einrückung hilft enorm, den Überblick über das Programm zu behalten, z. B. wenn es notwendig wird, eine vor mehreren Jahren geschriebene Stelle noch einmal zu überarbeiten.

Dies gilt in noch stärkerem Maße, wenn mehrere Programmierer gemeinsam an einem Programm arbeiten.

Top-Down-Ansatz. Der oben beschriebene Top-Down-Ansatz hilft dabei, übersichtliche Programme zu schreiben. Durch fortwährende Unterteilung werden komplizierte Sachverhalte einfach und handhabbar.

Variable und Funktionen kosten nichts. Wenn Sie durch Einführen einer zusätzlichen Variablen oder Funktion die Übersicht erhöhen können, sollten Sie dies tun. Der zunächst offensichtliche Verlust an Speicherplatz oder Rechenzeit wird in den allermeisten Fällen durch die Optimierung des Compilers aufgefangen. Seien Sie daher großzügig mit der Verwendung von Variablen und Funktionen.

Code-Verdopplung vermeiden! Sobald ein Code-Fragment an mehreren Stellen im Programm benötigt wird, sollten Sie dafür eine Funktion schreiben und eventuelle Unterschiede in der Verwendung durch Funktionsparameter kompensieren.

Das mehrfache Kopieren von Code mit anschließendem Verändern von Details („Cut-and-paste-Programmierung“) ist fehleranfällig: Bei einer nachträglichen Änderung übersieht man leicht, wirklich *alle* Kopien eines Code-Fragments anzupassen. Hierdurch entstehen regelmäßig schwer zu lokalisierende Fehler in Software-Produkten.

„Sprechende“ Namen. Es lohnt sich, Zeit in eine sinnvolle Benennung von Variablen und Funktionen zu investieren. Eine sinnvoll benannte Funktion (z. B. **int** is_leap_year (**int** year)) dokumentiert das Programms besser als ein Kommentar (z. B. **int** a (**int** b) /**returns 1 if b is a leap year*/). Ein kurzer Name wie z. B. i ist für einen lokalen Schleifenindex angebracht, nicht jedoch für eine globale Zustandsvariable oder Bibliotheksfunktion.

Globale Variable mit Bedacht einsetzen. In der Literatur wird oft empfohlen, die Verwendung von globalen Variablen grundsätzlich zu vermeiden und stattdessen Funktionsparameter zu verwenden. Hiervon sollte man nur in begründeten Ausnahmefällen abweichen (z. B. beim Speichern eines globalen Zustands des Programms).

Erst konservativ, dann erst ambitioniert programmieren. In vielen Fällen lohnt es sich, nicht sofort den elegantesten Ansatz zu verfolgen, sondern leichter verständliche Zwischenstufen einzuschieben.

Beispiel: Die Aufgabe, auf ein gegebenes Datum days Tage zu addieren, läßt sich durch eine Schleife lösen, die days-mal 1 Tag addiert. Dieser Ansatz ist ineffizient, aber narrensicher. Wenn man anschließend einen eleganteren Ansatz ausarbeitet, kann der narrensichere Ansatz beim Testen sehr hilfreich sein.

Bemerkung

Am 3. 1. 2009 meldete *heise online*:

> Kunden des ersten mobilen Media-Players von Microsoft erlebten zum Jahresende eine böse Überraschung: Am 31. Dezember 2008 fielen weltweit alle Zune-Geräte der ersten Generation aus. Ursache war ein interner Fehler bei der Handhabung von Schaltjahren.
>
> http://heise.de/-193332,

Der Artikel verweist auf ein Quelltextfragment, das für einen gegebenen Wert days das Jahr und den Tag innerhalb des Jahres für den days-ten Tag nach dem 1. 1. 1980 berechnen soll:

```
year = ORIGINYEAR; /* = 1980 */

while (days > 365)
  {
    if (IsLeapYear (year))
      {
        if (days > 366)
          {
            days -= 366;
            year += 1;
          }
      }
    else
      {
        days -= 365;
        year += 1;
      }
  }
```

Dieses Quelltextfragment weist mehrere Code-Verdopplungen auf:

- Die Anweisung year += 1 taucht an zwei Stellen auf.
- Es gibt zwei unabhängige Abfragen days > 365 und days > 366:
  eine in einer **while**- und die andere in einer **if**-Bedingung.
- Die Länge eines Jahres wird nicht durch eine Funktion berechnet oder in einer Variablen gespeichert; stattdessen werden an mehreren Stellen die expliziten numerischen Konstanten 365 und 366 verwendet.

Diese Probleme führten am 31. Dezember 2008 zu einer Endlosschleife, die sich – z. B. durch eine Funktion DaysInYear() – leicht hätte vermeiden lassen.

Gut hingegen ist die Verwendung einer Präprozessor-Konstanten ORIGINYEAR anstelle der Zahl 1980 sowie die Kapselung der Berechnung der Schaltjahr-Bedingung in einer Funktion IsLeapYear().

1.10 Dynamischer Speicher

In vielen Situationen ist während der Erstellung des Programms nicht bekannt, wieviele Daten von welchem Typ anfallen werden. So ist es beispielsweise bei einem Textverarbeitungsprogramm weder akzeptabel, die damit schreibbaren Texte auf 1000 Zeilen zu begrenzen, noch ist es sinnvoll, bei jedem Aufruf Platz für eine Million Zeilen zu reservieren.

Die Ähnlichkeiten zwischen Arrays und Zeigern machen es in C sehr einfach, mit Arrays von vorher nicht festgelegter Länge zu arbeiten: Anstelle eines Arrays verwendet man einen Zeiger auf den entsprechenden Datentyp. Sobald die Größe bekannt ist, reserviert man mit Hilfe der Funktion malloc() Speicherplatz für das Array. Danach kann man es über den Zeiger wie jedes „normale“ Array ansprechen. Wenn der Speicherplatz nicht mehr benötigt wird, kann man ihn mit Hilfe der Funktion free() wieder freigeben.

Das Beispielprogramm dynmem-1.c illustriert dies anhand eines dynamischen Arrays ganzer Zahlen:

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

int main (void)
{
  int *buffer = NULL;
  int count = 7;
  buffer = malloc (count * sizeof (int));
  for (int i = 0; i < count; i++)
    buffer[i] = i;
  for (int i = 0; i < count; i++)
    printf ("%d ", buffer[i]);
  printf ("\n");
  free (buffer);
  return 0;
}
```

Die Zeigervariable buffer zeigt zunächst auf „nichts“ (NULL). Die Größe des Arrays wird einer Variablen count entnommen. Im malloc()-Aufruf wird übergeben, wieviel Speicherplatz angefordert wird, nämlich count-mal die Größe eines Array-Elements. Anschließend kann mit dem Array „normal“ gearbeitet werden.

Das Freigeben des Speicherplatzes erfolgt in C *nur bei Programmende* automatisch. (Einen *garbage collector*, wie es ihn in vielen anderen Sprachen gibt, kennt C von Hause aus nicht.) Angeforderter Speicher, der nicht mehr benötigt wird, kann u. U. weitere Speicheranforderungen blockieren und sollte daher freigegeben werden.

C sieht keine Möglichkeit vor, der Zeigervariablen buffer anzusehen, wie groß das Array ist. Da wir diese Information in aller Regel benötigen, müssen wir sie separat speichern, hier z. B. in der Variablen count. Für viele Anwendungen ist es sinnvoll, den Zeiger auf die eigentlichen Daten gemeinsam mit der Größe des Arrays in einem **struct** – dem C-Äquivalent einer *Container-Klasse* – zu speichern.

Aufgabe

Schreiben Sie eine Bibliothek zur Nutzung eines dynamischen Arrays
für selbstgewählte Daten (z. B. Integer, String).

(ohne Musterlösung)

1.11 Rekursive Datenstrukturen

Die in Abschnitt 1.10 eingeführten dynamischen Arrays haben den Vorteil, daß man über den Index unmittelbar auf jedes Array-Element zugreifen kann. Diesem Vorteil steht der Nachteil gegenüber, daß für jedes Einfügen eines Elements in das Array ein größerer Speicherbereich mit Hilfe einer Schleife verschoben werden muß.

Die *verkettete Liste* ist eine Datenstruktur, bei der es sich genau umgekehrt verhält: Man kann an jeder Stelle ein neues Element unmittelbar einfügen, aber nur der Reihe nach auf die Elemente zugreifen. Um insbesondere auf ein konkretes Element zuzugreifen, benötigt man eine Schleife.

Eine verkettete Liste besteht aus **struct**-Variablen, die neben ihrer eigentlichen „Nutzlast“ noch einen Zeiger enthalten, der auf eine **struct**-Variable desselben Typs zeigt, nämlich „die nächste in der Liste“.

Das Beispielprogramm string-list-1.c illustriert eine verkettete Liste, die insgesamt drei Strings enthält:

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

typedef struct string_list
{
  char *content;
  struct string_list *next;
}
string_list;
```

```
int main (void)
{
  string_list *first = malloc (sizeof (string_list));
  first->content = "Dies ist ein Test.";
  first->next = malloc (sizeof (string_list));
  first->next->content = "Was ist das?";
  first->next->next = malloc (sizeof (string_list));
  first->next->next->content = "Ein Test.";
  first->next->next->next = NULL;
  for (string_list *p = first; p; p = p->next)
    printf ("%s\n", p->content);
  while (first)
    {
      string_list *next = first->next;
      free (first);
      first = next;
    }
}
```

- Die Zeiger-Variable first zeigt auf das erste Element der Liste.
- Jedes Listenelement enthält einen Zeiger next auf das nächste Element. Insbesondere ist first->next das zweite Element der Liste, first->next->next das dritte usw.
- Jedes Listenelement wird dynamisch mit malloc() angelegt.
- Die Nutzlast besteht in diesem Beispiel aus String-Konstanten: Der in jedem Listenelement enthaltene content-Zeiger wird so gesetzt, daß er auf eine irgendwo im Speicher befindliche String-Konstante verweist.
- Das Ende der Liste wird dadurch markiert, daß der next-Zeiger auf NULL („nichts“) zeigt.
- Um alle Listenelemente der Reihe nach durchzugehen, läßt man einen Zeiger p zunächst auf das erste Listenelement first zeigen. Danach arbeitet man mit den Komponenten des Elements, auf das p zeigt. Um mit dem nächsten Element weiterzumachen, geht man von p zu p->next über. Die Schleife ist beendet, sobald p den Wert NULL hat.
- Um die Liste wieder freizugeben, geht man sie auf ähnliche Weise durch, muß dabei allerdings beachten, daß der Inhalt eines wieder freigegebenen Speicherbereichs undefiniert ist. Es ist daher notwendig, den Zeiger auf das nächste Listenelement zwischenzuspeichern.

Um die Liste in der umgekehrten Reihenfolge durchzugehen, ist zusätzliche Arbeit erforderlich. Eine mögliche Vorgehensweise ist eine *rekursive* Funktion, d. h. eine Funktion, die sich selbst aufruft (Beispielprogramm: string-list-2.c):

```
void print_reverse (string_list *p)
{
  if (p)
    {
      print_reverse (p->next);
      printf ("%s\n", p->content);
    }
}
```

Beim Aufruf belegt diese Funktion zwei Zeiger auf dem CPU-Stack: die Rücksprungadresse und den Parameter p. Normalerweise stellt dies kein Problem dar, da der Speicher wieder freigegeben wird, sobald die Funktion durchgelaufen ist. Wenn die Funktion sich jedoch selbst aufruft, ist sie noch nicht durchgelaufen und belegt demzufolge diesen Speicher ein weiteres Mal.

Die Anzahl der Selbstaufrufe, die sog. *Rekursionstiefe*, entspricht hier der Länge der Liste. Für sehr lange Listen und/oder einen stark begrenzten CPU-Stack (z. B. auf Mikro-Controllern) kann diese Vorgehensweise daher zu einem Überlauf des CPU-Stacks, also zu einem Absturz führen.

Eine alternative Vorgehensweise, um eine verkettete Liste rückwärts durchzugehen, ist eine Doppelschleife (Beispielprogramm: string-list-3.c):

```
void print_reverse (string_list *first)
{
  string_list *p = first;
  while (p && p->next)
    p = p->next;
  while (p)
    {
      printf ("%s\n", p->content);
      string_list *q = first;
      while (q && q->next != p)
        q = q->next;
      p = q;
    }
}
```

Diese Vorgehensweise ist umständlicher und langsamer als eine rekursive Funktion, belegt aber nur begrenzten Platz auf dem CPU-Stack.

Eine weitere Herangehensweise besteht darin, zusätzlich zu einem Zeiger auf das nächste Listenelement auch einen Zeiger auf das vorherige Listenelement in der Struktur zu speichern. Eine derartige sog. *doppelt verkettete Liste* vermeidet die Nachteile der beiden genannten Vorgehensweisen, hat dafür aber den Nachteil, daß jedes einzelne Listenelement einen zusätzlichen Zeiger enthält, was mehr Speicherplatz (und mehr Verwaltungsarbeit) erfordert.

Welche der genannten Implementationen einer dynamischen Datenstruktur (dynamisches Array, verkettete Liste mit Rekursion oder mit Doppelschleife, doppelt verkettete Liste) die beste ist, hängt stark von den jeweiligen Anforderungen ab.

Ein gutes Programm zeichnet sich dadurch aus, daß es alle wichtigen Operationen in Funktionen kapselt, so daß ein nachträglicher Austausch der Implementation mit wenig Aufwand und – vor allem – geringer Fehleranfälligkeit möglich ist.

Aufgabe

Schreiben Sie eine Bibliothek zur Nutzung einer verketteten Liste
für selbstgewählte Daten (z. B. Integer, String).