B-BKSPP-PR (WS 2013/2014)
Einführung in die Systemprogrammierung mit C
Teil 1a: Grundlagen der Sprache C

13. Oktober 2013

> *"C is not a "very high level" language, nor a "big" one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages."*
> *— Brian W. Kernighan, Dennis M. Ritchie, "The C Programming Language" (2nd ed.)*

Im ersten Teil dieses Praktikums beschäftigen wir uns mit den grundlegenden Konzepten und Werkzeugen der Sprache C. Dieser erste Teil besteht aus einer kurzen Einführung in C (Teil 1a), gefolgt von einer Einführung in die wichtigsten Werkzeuge zur Verwendung von C (Teil 1b).

Um C zu verstehen, ist es hilfreich, die Ziele zu verstehen, die C verfolgt, und zugleich auch die Ziele zu verstehen, die die Sprache *nicht* verfolgt.

Ziele von C *sind* bzw. *waren*:

- Für die Entwicklung aller Arten von Programmen nützlich zu sein
- In enger Symbiose mit dem UNIX-System die Entwicklung auf und für dieses Systems zu ermöglichen[1]
- Nahe an der Hardware zu sein
- Nahe an der *Maschinensprache* zu sein (Abschnitt 1), mit der der Prozessor gesteuert wird, ohne sich auf die Maschinensprache eines bestimmten Prozessors festzulegen
- Portierung von Programmen, also die Übertragung eines Programmes, das für ein bestimmtes Computersystem entwickelt wurde, auf ein anderes Computersystem, zu vereinfachen
- Einfach verständlich und ausdrucksfähig zu sein
- Die Erzeugung von effizientem Maschinencode zu erlauben
- Schnell übersetzbar zu sein
- Keinen unnötig großen Sprachumfang zu haben

Ziele von C sind *nicht*:

- Plattformunabhängige Ausführung zu gewährleisten— ein C-Programm, das auf einem 32-Bit-System entwickelt wurde, wird u.U. nur mit Änderungen auf einem 64-Bit-System korrekt funktionieren

[1] C ist unabhängig von UNIX definiert, aber einige Konventionen und Bibliotheksfunktionen der Sprache ergeben einen verständlicheren Sinn im Kontext von UNIX.

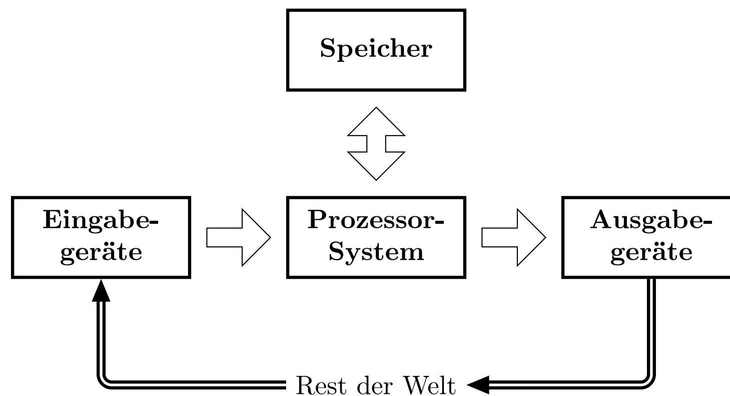


Abbildung 1: Grundlegende Struktur eines Computersystems (Analog zu einem Moore-Automaten)

- Starke Typisierung, also das garantierte Finden aller Typfehler— in C-Programmen können Speicherzellen explizit adressiert werden, so daß man einen Speicherblock absichtlich oder unabsichtlich sowohl z.B. als Zahl oder als Zeichenkette interpretieren kann.
- Hohe Abstraktionsfähigkeit zu erlauben— im Vergleich zu Hochsprachen wie SmallTalk oder Haskell sind intuitiv recht grundlegende Operationen wie Mengenvereinigung oder generische Algorithmen oft nur recht umständlich möglich.

Aus diesem Grund wird C daher manchmal auch informell als eine 'portable Assemblersprache' bezeichnet, ein Konzept, das wir im nächsten Abschnitt kurz erläutern.

1 Hintergrund: Maschinensprache und Assemblersprache

Ein Computersystem besteht neben Ein- und Ausgabesystemen (Abbildung 1) aus *Speicher* und einem *Prozessorsystem* (einem oder mehreren Prozessoren, die wiederum aus mehreren Recheneinheiten bestehen können). Während der Computer "läuft," liest der Prozessor kontinuierlich Befehle aus dem Speicher und führt diese aus. Aus Effizienzgründen werden die Befehle dazu in Bitfolgen kodiert, in ein sogenanntes *Maschinenprogramm*. Die Sprache, aus der Maschinenprogramme bestehen, nennt man *Maschinensprache*.

1.1 Speicher

Wir können uns den Speicher vereinfacht als eine Kette von Bits vorstellen, zum Beispiel:

```
10010000 01110010 01101111 01101110 01110010 01100001 01101101 01101101 00000000
```

Da für das menschliche Auge lange Folgen von `1` und `0` schwer zu erkennen sind, verwenden wir in der Praxis meist nicht Binärdarstellung, sondern (auch um Platz zu sparen) eine Darstellung in der Basis 16, als *Hexadezimalzahlen*. Wir verwenden hier dazu die C-Notation, die Hexadezimalzahlen mit dem Präfix `0x` notiert. Somit ist also z.B.

$$\text{(Hexadezimal)}\texttt{0x10} = 16\text{(Dezimal)}$$

Die Ziffern jenseits von 9 werden dabei als `a` bis `f` notiert:

$$\text{(Hexadezimal)}\texttt{0x1f} = 31\text{(Dezimal)}$$

Dies hat die angenehme Eigenschaft, daß alle Zahlen, die in einem (modernen 8-Bit) Byte ausgedrückt werden können, mit zwei Hexadezimalziffern, `00` bis `ff`, ausgedrückt werden können. Wir können also die obige Bitfolge hexadezimal wie folgt notieren:

10010000	01110010	01101111	01101110	01110010	01100001	01101101	01101101	00000000
50	72	6f	67	72	61	6d	6d	00

Wenn wir die Speicherinhalte eines beliebigen Rechners ausgeben würden, bekämen wir also genau solche Bitfolgen zu sehen. Der Speicher selbst hat kein Wissen darüber, was genau die Bedeutung dieser Bitfolgen sind; diese Bedeutung ergibt sich ausschließlich durch ihre Verwendung. Diese Verwendung kann in zwei Formen daherkommen:

- Durch das Prozessorsystem
- Durch Ausgabegeräte

Erstere Verwendung ergibt sich, wenn der Prozessor die Speicherinhalte als *Maschinenprogramm* oder als *Daten* verwertet[2]. Zum Beispiel könnten wir die ersten acht der neun obigen Bytes als Intel-Maschinenspracheprogramm interpretieren. Dieses Programm besteht dann aus den folgenden fünf individuellen Maschinenbefehlen:

```
50
72 6f
67 72 61
6d
6d
```

Um diese Befehle menschenlesbar zu machen, notiert man sie normalerweise nicht in Maschinensprache, sondern in *Assemblersprache*:

```
       push  rax
       jb    0x72
addr32 jb    0x67
       ins   DWORD PTR es:[rdi],dx
       ins   DWORD PTR es:[rdi],dx
```

Diese fünf Befehle weisen den Prozessor an, bestimmte Informationen zu kopieren, zwei Mal unter bestimmten Umständen zu springen, und zwei Mal Informationen von angeschlossenen Eingabegeräten auszulesen. Solcher Code könnte theoretisch im Betriebssystemkern vorkommen, auch, wenn das etwas ungewöhnlich wäre.

Diese 8 Bytes können aber auch noch auf einige andere Arten und Weisen interpretiert werden. Wenn ein Programm die Werte als Daten einliest, kann es sie z.B. als 32-Bit Zahlenwerte (`int` in C, Java, C++ auf der Intel-Architektur) interpretieren, was die Zahlen 1735357008 und 1835884914 ergeben würde, oder als 64-Bit-Fließkommazahl (`double`) die dann einen Wert von ungefähr $1.29642767 \times 10^{219}$ hätte. Der Grund für diese spezifischen Werte ist, daß der Prozessor selbst intern für bestimmte Zahlenkodierungen konstruiert ist; wenn man ihn anweist, eine Zahl aus dem Speicher zu laden oder sie dorthin zurückzuschreiben, dann geschieht dies in der entsprechenden Kodierung.

Wenn wir die obigen Bytes ein Ausgabegerät weitergeben, wird die Interpretation wiederum vom Gerät selbst abhängig sein. Ein textbasiertes LCD zum Beispiel würde, wenn wir ihm die obigen Bytes senden werden, die Zeichenkette „`Programm`“ darstellen. Grund dafür ist ein Standard zur Interpretation von Bytes als Zeichen, nämlich der ASCII[3]-Standard, der allen Zahlen von 32 bis 126 ein lesbares

[2]Es gibt auch Situationen, in denen bestimmte Speicherinhalte sowowhl als Programm als auch als Daten verwertet werden.

[3]“American Standard Code for Information Interchange”, also „Amerikanischer Code für Informationsaustausch.“

Zeichen zuordnet. So ist z.B. `0x41` der Großbuchstabe `A`, `0x42` der Großbuchstabe `B` usw.; und entsprechend auch `0x50` der Großbuchstabe `P`. Diese Zeichen beinhalten übrigens keine deutschen Umlaute, was der Grund dafür ist, warum einige (ältere und neuere) Systeme verschiedene Probleme mit eben diesen haben.

Wir haben also vier verschiedene mögliche Interpretationen der gleichen Bytes beobachtet:

- Maschinencode
- Zwei 32-Bit Ganzzahlenwerte
- Ein 64-Bit Fließkommawert
- Eine Zeichenkette

In der Praxis ist die Zahl der möglichen Interpretationen natürlich nur durch die Kreativität der Programmierer beschränkt.

Der Speicher weist den gespeicherten Bytes fortlaufende *Adressen* zu. So ist das erste Byte im Speicher z.B. an Adresse `0` gespeichert, das zweite Byte an Adresse `1` etc. Das Prozessorsystem kommuniziert mit dem Speicher auf der Basis dieser Adressen. Die beiden möglichen Interaktionen sind dabei:

- Daten von einer bestimmten Adresse zu lesen
- Bestimmte Daten an eine bestimmte Adresse zu schreiben

Vereinfachtend gehen wir hier davon aus, daß diese 'Daten' immer genau ein Byte sind.

In unserer bisherigen Betrachtung haben wir uns auf den *Hauptspeicher* konzentriert. Ein moderner Rechner hat aber auch noch weitere Formen von Speicher. Für unsere Betrachtung ist dabei eine zweite Form von Speicher relevant, nämlich *Prozessorregister*. Diese Register sind kleien Speicherzellen, die direkt auf dem Prozessor untergebracht sind. Lese- und Schreibzugriff auf Register ist deutlich schneller als auf den Hauptspeicher, dafür ist aber in ihnen verfügbare Speicherplatz sehr beschränkt. Die folgende Tabelle vergleicht diese beiden Speicherformen:

	Adressierbar	**Größe**	**Geschwindigkeit**
Hauptspeicher	ja	erweiterbar: mehrere GiB	schnell
Register	nein	fest: 1 kiB oder weniger	sehr schnell

Beachten Sie insbesondere die gewaltigen Größenunterschiede zwischen den Speicherkapazitäten: selbst trivialste Programme sind normalerweise zu groß, um ohne Hauptspeicher auszukommen.

Die Spalte '**Adressierbar**' weist darauf hin, daß Register *keine* Adressen haben; wenn wir von einer Speicheradresse sprechen, beziehen wir uns also immer auf den Hauptspeicher.

1.2 Prozessorsystem

Jeder aktivierte Prozessor im Prozessorsystem und jeder Prozessorkern in diesem Prozessor führt seinen „eigenen" Satz an Maschinenbefehlen aus. Damit der Prozessor weiß, welcher Befehl als nächstes ausgeführt wird, verfügt er über ein sogenanntes *Programmzählerregister* (PC) oder *Instruktionszeigerregister* (IP).

Betrachten wir ein Beispiel für diesen Prozeß, um ein zumindest oberflächliches Verständnis für das Verhalten eines Prozessors zu erhalten. Wir nehmen dazu an, daß wir auf einem 32-Bit-Intel-Prozessor arbeiten. Das Programmzählerregister auf diesem Prozessor heißt EIP, und wir gehen davon aus, daß:

- EIP = `0x40041b` (eine typische Adresse unter Linux)

- Der Speicher an Adresse `0x4041b` enthält das Byte `bb`, und der Speicher an den unmittelbar folgenden Adressen enthält die Bytes `0a 00 00 00 bf c4 05 40 00 e8 b6 ff ff ff 83 eb 01 75 f1`.

Wie sie sehen, *zeigt* EIP auf das erste Byte `0x0a` (Adresse `0x4041b`) unserer Folge von Bytes. Solche Indirektion über Speicheradressen ist die wichtigste Form der Abstraktion auf der Prozessorebene: wenn wir nun EIP um eins erhöhen würden, würden wir auf das nächste Byte (`0xbf`, Adresse `0x4041c`) zeigen.

Die angegebene Bytefolge ist ein Maschinenprogramm. Wenn wir nun den Prozessor mit dieser Konfiguration starten, wird er dieses Programm ausführen. Zur Lesbarkeit haben wir das Programm unten in Assemblersprache dekodiert:

```
0x40041b:    mov    ebx, 0xa
0x400420:    mov    edi, 0x4005c4
0x400425:    call   0x4003e0 <puts@plt>
0x40042a:    sub    ebx, 0x1
0x40042d:    jne    0x400420
```

Hierbei sind links die Speicheradressen notiert und rechts die von Maschinensprache nach Assembler dekodierten Befehle.

Wenn wir also unseren Prozessor nun laufen lassen, wird er zuallernächst den Befehl, auf den EIP zeigt, aus dem Speicher lesen und ausführen:

```
0x40041b:    mov    ebx, 0xa
```

Dieser Befehl schreibt die Zahl `0xa` (also 10 dezimal) in ein Register namens EBX. Wie zuvor angedeutet, können wir uns dieses Register as eine Art globale Variable vorstellen, die außergewöhnlich schnell gelesen und geschrieben werden kann. Zusätzlich zum Ausführen des Befehls selbst erhöht der Prozessor auch EIP, um auf den nächsten Befehl zu zeigen:

```
0x400420:    mov    edi, 0x4005c4
```

Hier laden wir den Wert `0x4005c4` in das Register EDI. Beachten Sie, daß dieser Wert eine Speicheradresse in der Nähe des obigen Maschinencodes ist. Aus der Sicht des Prozessors existiert kein konzeptioneller Unterschied zwischen diesem Ladebefehl und dem vorherigen, aber als menschliche Beobachter können wir mutmaßen (aber zunächst nicht mit Sicherheit schließen), daß der hier geladene Wert als Adresse verwendet werden wird und nicht als Zahl.

Der nächste Befehl ruft eine Subroutine auf:

```
0x400425:    call   0x4003e0 <puts@plt>
```

Die Ausführung dieses Befehls setzt EIP auf die Adresse `0x4003e0` und ändert ein weiteres Register und eine Speicherzelle. Der Prozessor wird also im Anschluß ab Adresse `0x4003e0` weiterarbeiten, kehrt aber nach mehreren tausend ausgeführten Instruktionen wieder in unser Programm zurück. Für unsere Betrachtung hier reicht es, wenn wir uns auf den Effekt dieser Subroutine beschränken: sie gibt eine Zeichenkette an der Adresse, auf die EDI zeigt, aus. Wir wissen nun also, daß `0x4005c4` tatsächlich ein Zeiger in den Speicher ist, und daß dort eine Zeichenkette steht.[4]

Der nächste Befehl zieht 1 vom aktuellen Inhalt des Registers EBX ab:

```
0x40042a:    sub    ebx, 0x1
```

Zugleich vergleicht dieser der Befehl EBX mit 0. Falls EBX gleich ist, wird dieser Zustand in einem weiteren Register (das im Befehl nicht explizit genannt wird) gespeichert. Dies beeinflußt dann den nächsten Befehl:

[4] Natürlich werden diese Befehle auch ausgeführt, wenn EDI keine gültige Speicheradresse ist oder die Bytes an der betreffenden Adresse nur „Müll“ beinhalten. Dies kann zu wirren Programmausgaben oder Programmabstürzen führen.

```
0x40042d:    jne    0x400420
```

Dieser Befehl springt zurück auf die Adresse `0x400420`, also zu der Stelle im Programm, in der wir EDI laden, aber nur dann, wenn der letzte Vergleich (hier durch die Subtraktion ausgelöst) ein Ergebnis ungleich 0 hatte.

Dieses Programm wird also EBX von 10 bis 0 herunterzählen und für jeden Durchlauf eine Zeichenkette ausdrucken.

Wie erwähnt, ist dieses Programm spezifisch für die Intel-Architektur. Das gleiche Programm für den SPARC-Prozessor (der von einigen Hochleistungsservern verwendet wird) würde wie folgt aussehen:

```
clr    %i4
sethi  %hi(0x10400), %g1
add    %g1, 0x3a0, %i5
mov    %i5, %o0
call   0x20824 <puts>
inc    %i4
cmp    %i4, 9
ble,a  %icc, 0x106c8 <main+20>
```

Wie wir sehen, haben SPARC-Programme merklich andere Befehle, also eine andere Maschinen– und Assemblersprache. Es existieren noch sehr viel mehr Maschinensprachen, und selbst Prozessoren der gleichen Familie haben oft Detailunterschiede in den Befehlssätzen, Registergrößen etc. Allerdings existieren zahlreiche Gemeinsamkeiten. So verfügt jede moderne Assemblersprache über:

- Mehrere Register, die wie schnelle globale Variablen verwendet werden können
- Maschinenbefehle zum:
  - Laden vom Hauptspeicher in Register
  - Schreiben von Registern in den Haupspeicher
  - Direktladen von Zahlen in Register
  - Sprünge (Ändern des Programmzählers)
  - Berechnen arithmetischer Operationen (auf bestimmte Bitgrößen beschränkt)
  - Berechnen bestimmter Bitmusterverknüpfungen (bitweise-UND, bitweise-ODER etc.)
  - Vergleich von Registern
  - Bedingte Sprünge (Ändern des Programmzählers abhängig von einem Vergleich)

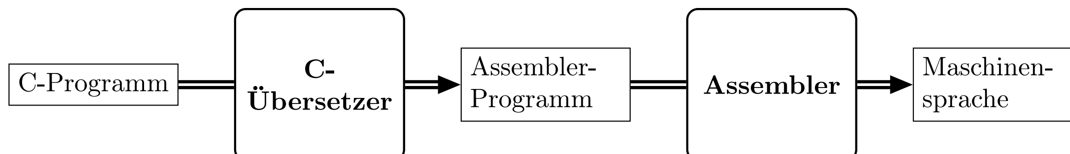
Aus diesen Gemeinsamkeiten leitet sich die Sprache C her. Um strukturierte Programmierung zu begünstigen, wurden Sprünge im Wesentlichen durch Kontrollstrukturen (Schleifen und Bedingungen) ersetzt, aber ansonsten orientiert sich C stark an Assembler– bzw. Maschinensprache. Zum Vergleich: das entsprechende C-Programmfragment sah wie folgt aus:

```
for (int i = 0; i < 10; i++) {
    puts("Hallo!");
}
```

Wir werden diese Konstrukte am Abschnitt 3 genauer betrachten.

2 Hintergrund: Der C-Übersetzer

Wie im letzten Abschnitt erklärt, kann der Prozessor direkt nur Maschinensprachebefehle ausführen, und das auch nur, wenn diese Maschinensprachebefehle im Arbeitsspeicher des Rechners abgelegt sind (also z.B. nicht direkt von der Festplatte oder einem externen Datenträger). Um ein Programm einer beliebigen höheren Sprache (Python, Haskell, Java, C, . . .) ausführen zu können, muß dieses Programm also in irgendeiner Form dem Prozessor 'schmackhaft gemacht' werden. Eine der Möglichkeiten, um dies zu erreichen, ist die Übersetzung („Kompilierung“) des Programmes aus der höheren Sprache in Maschinensprache. In C findet diese Übersetzung in mehreren Phasen statt, die hier leicht vereinfacht dargestellt sind:



Wir werden in Abschnitt 3.26 eine Verfeinerung dieses Prozesses betrachten. Ein vom Benutzer angegebenes C-Programm wird also zunächst in ein Assemblerprogramm kompiliert, und von dort aus in Maschinensprache übersetzt. Tatsächlich bleiben diese Details jedoch meist verborgen. So können Sie ein C-Programm `programm.c` auf der Kommandozeile sehr direkt in ein ausführbares Programm `programm` übersetzten:

```
gcc --std=c99 programm.c -o programm
```

Sofern beim Übersetzen keine Fehler auftreten, wird der C-Übersetzer `gcc` das ausführbare Programm `programm` erzeugen. Eventuelle Fehler werden mit Zeilennummer und Erklärung angegeben.

Die Angabe von `-std=c99` weist den Übersetzer darauf hin, den C-Standard C99 zu verwenden, statt des älteren C89, das meist noch die Standardeinstellung ist. Dies vereinfacht einige Konstrukte. Um diesen Parameter nicht jedes Mal angeben zu müssen, können Sie der Shell sagen, daß sie ein neues Kommando `gcc99` anbieten soll:

```
alias gcc99=gcc --std=c99
```

Nun können Sie `gcc99` statt `gcc -std=c99` verwenden:

```
gcc99 programm.c -o programm
```

Beachten Sie dabei, daß diese Einstellung nur für ihre aktuelle Kommando-Shell gilt (mit der Shell befassen wir uns genauer in den Abschnitten 3.2 und 3.3). Falls sie das Alias immer automatisch einrichten möchten, können Sie es an die Datei `/.bashrc` anhängen, so daß es automatisch beim Start Ihrer Shell ausgeführt wird[5].

3 Aufgaben

3.1 RBI-Konto

Aufgabe. Falls Sie noch kein Benutzerkonto bei der RBI eingerichtet haben, sollten Sie dies nun tun. Selbst wenn Sie auf einem eigenen Laptop arbeiten, kann es nützlich sein, ein RBI-Konto als „Zweitzugang“ zu verwenden. RBI-Nutzeranträge können direkt bei der RBI (Kellergeschoß Robert-Mayer-Str. 11-15) gestellt werden.

[5]Diese Angaben, wie die meisten Angaben bezüglich der Shell, gehen davon aus, daß Sie die `bash` verwenden, die verbreitetste Shell. Für andere Shells wie die `tccsh` gelten analoge aber andere Regeln.

3.2 Grundlagen der UNIX-Kommandozeile (1)

Die RBI stellt unter `https://www-intern.rbi.informatik.uni-frankfurt.de/rbi/linux-tutorials/view?set_language=de` eine Einführung in die lokalen UNIX-Systeme, inklusive der Kommandozeilenumgebung, zur Verfügung. Lesen Sie zunächst diese Beschreibung durch, falls Sie noch nicht mit der UNIX-Kommandozeile vertraut sind. Wichtig sind insbesondere:

- Start und Interaktion mit einer Kommandozeilenumgebung
- Verzeichnisstrukturen
- Umgang mit der UNIX-Dokumentation über das Hilfsprogramm `man`

Wie wir im Laufe des Praktikums sehen werden, sind sowohl C als auch UNIX eng mit der Kommandozeile verbunden, und viele Eigenschaften des Systems sind in diesem Kontext leichter ersichtlich. Im Folgenden werden gelegentlich neue Programme eingeführt, die teilweise nur aus der Kommandozeile gestartet werden können. Es ist empfehlenswert, eine Liste der hier eingeführten Programme und deren Bedeutung zu führen, sofern Sie diese noch nicht kennen– Sie können jederzeit die Bedeutung und Bedienung eines Programmes über das Hilfsprogramm `man` nachschlagen.

Beachten Sie, daß auch C-Funktionen größtenteils mit `man` dokumentiert sind, allerdings in einem anderen „Kapitel“ (Abschnitt 3.6).

Aufgabe. Lesen Sie die RBI-Einführung zu den UNIX-Systemen durch, wenn Sie dies noch nicht getan haben.

3.3 Grundlagen der UNIX-Kommandozeile (2)

Aufgabe. Lesen Sie die (Kurz-)beschreibungen der folgenden Befehle nach, die Sie im Folgenden gelegentlich benötigen werden:

- `mkdir`
- `rmdir`
- `cp`
- `rm`

3.4 Editoren

Es existieren zahlreiche Editoren, die zur Arbeit mit C geeignet sind. Die traditionell beliebtesten Editoren sind

- `emacs` (`http://www-pool.math.tu-berlin.de/doc/emacs/`), und
- `vi` bzw. `vim` (`http://www.fh-schmalkalden.de/schmalkaldenmedia/Downloads/elektrotechnik/l/edv/vi.pdf`).

Die Lernkurve für beide Systeme kann jedoch recht steil sein. Eine einfache Alternative ist der `nano`-Editor, der im Rahmen dieses Praktikums ausreichend sein sollte.

Alternativ können Sie auch IDEs verwenden:

- Eclipse (`http://eclipse.org`)
- NetBeans (`http://netbeans.org`)
- Code::Blocks (`http://codeblocks.org`)

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv)
4 {
5     puts("Hallo, Welt!");
6     return 0;  // Rueckgabewert 0 signalisiert "kein Fehler"
7 }
```

Abbildung 2: Ein erstes C-Programm: `hallo.c`

Der Vorteil der neueren IDEs ist, daß sie auch Unterstützung für Programmtransformationen wie Refactoring anbieten und bei komplexeren Projekten den Entwicklern einige Verwaltungsarbeit abnehmen können. Der Nachteil der letztgenannten Vereinfachungen ist, daß solche Vereinfachungen meist spezifisch für die IDE sind und nur Sinn machen, wenn alle Entwickler die gleiche IDE verwenden. Im Rahmen zumindest von Teil 1 und 2 des Praktikums werden Sie die IDEs allerdings primär nur als Editoren verwenden, so daß viele der erweiterten Fähigkeiten unerheblich sein werden.

Aufgabe. Entscheiden Sie sich für einen Editor; Sie sind nicht an die obigen Vorschläge gebunden.

3.5 Übersetzung und Ausführung

Übernehmen Sie das Programm aus Abbildung 2. Schreiben Sie es in eine Datei (mit dem Editor Ihrer Wahl), übersetzen Sie es (wie unter Abschnitt 2 beschrieben) und führen Sie es aus. Das Programm sollte die Zeichenkette

```
Hallo, Welt!
```

in Ihr Terminal-Fenster ausgeben und ohne Fehler zurückkehren.

Beachten Sie zur Ausführung, daß UNIX-Shells aus Sicherheitsgründen üblicherweise so konfiguriert sind, daß sie Programme im aktuellen Verzeichnis nicht 'sehen' können– wenn Sie also Ihr Programm frisch übersetzt haben (z.B. in die Datei `hallo`), müssen Sie der Shell den vollständigen Pfad angeben (z.B. `./hallo`).

Dieses Programm gibt zwar nur eine Zeichenkette aus, hat aber bereits ein merkliches Maß an Komplexität. Wir werden später einige Details des Programmes erklären:

- Zeile 1 in Abschnitt 3.26
- Zeile 3 in Abschnitt 3.10
- Zeile 6 in Abschnitt 3.25

Für unsere Zwecke in dieser Teilaufgabe ist es ausreichend, wenn wir uns darauf verständigen, daß die Bedeutung der Zeilen folgende ist:

- Zeile 1 stellt einige Operationen, wie die Operation `puts`, zur Verfügung
- Zeile 3 sagt dem System, wo das Programm anfängt
- Zeile 5 druckt die Ausgabe aus
- Zeile 6 „muß so sein“, um mit dem Rest des Systems „korrekt“ zu interagieren
- Und die geschweiften Klammern in Zeilen 4 und 7 drücken aus, daß die in ihenn eingeschlossenen Anweisungen nacheinander augeführt werden sollen.

Aufgabe. Übersetzen Sie das Programm, und führen Sie es aus.

3.6 Numerische Ausgaben

Die Funktion `printf` kann zur formatierten Ausgabe verwendet werden:

```
printf("Zahl: %d,  Hexadezimal: %x,  Zeichenkette:  %s\n", 17, 254, "foo");
```

Dieser Befehl wird folgenden Text ausgeben:

```
Zahl: 17,  Hexadezimal: fe,  Zeichenkette:  foo
```

Hierbei werden in der als ersten Parameter angegebenen Zeichenkette alle *Konvertierungsspezifikationen* (conversion specification) (die mit einem Prozentzeichen % beginnen) durch die späteren Parameter der Funktion ersetzt; die korrekte Anzahl der Parameter ist also abhängig von der Anzahl der Konvertierungsspezifikationen. Die zusätzlichen Parameter werden dabei in der Reihenfolge „eingesetzt", in der auch die Konvertierungsspezifikationen auftauchen. `printf` hat mehrere Konvertierungsspezifikationen; die hier verwendeten sind:

- `%d`: Ganzzahl
- `%x`: Ganzzahl hexadezimal
- `%s`: Zeichenkette

Aufgaben.

1. Schlagen Sie die Dokumentation der Funktion `printf` nach:

   ```
   man 3 printf
   ```

   Die Zahl „3" hier sagt `man`, das `printf` in Abschnitt 3 des Systemhandbuchs dokumentiert ist. Das Systemhandbuch hat mehrere Abschnitte, von denen uns hier die ersten drei am Wichtigsten sind:

   - 1: Shell-Befehle
   - 2: Systemaufrufe (die von C aus zugänglich sind)
   - 3: C-Befehle, die keine Systemaufrufe sind

   Wir werden uns mit Systemaufrufen hauptsächlich in Teil 2 des Praktikums beschäftigen.

   `man` sucht immer nach dem ersten Vorkommen eines Begriffes; so kann es z.B. bei `printf` sein, daß es einen Kommandozeilenbefehl des gleichen Namens anzeigt, statt die C-Funktion zu dokumentieren.

2. Verwenden Sie die Funktion `printf`, um das Ergebnis von drei Berechnungen auszugeben:

   (a) $1 + 1$
   (b) 23 * 42
   (c) 65535 * 65535

   Notieren Sie das Ergebnis der letzten Berechnung. Entspricht es Ihren Erwartungen? Wenn ja, erklären Sie das Ergebnis.

3. Was passiert, wenn Sie in unserem Anfangsbeispiel zu `printf` eine der Zahlen mit der Zeichenkette `"foo"` vertauschen?

Operatoren (stärker bindende oben)	Arg.	Assoz.
(*Ausdruck*) [...] -> .	2	links
! ~ ++ - + - (typ) * & **sizeof**	1	rechts
* / %	2	links
+ -	2	links
<< >>	2	links
< <= >= >	2	links
== !=	2	links
&	2	links
^	2	links
|	2	links
&&	2	links
||	2	links
... ? ... : ...	3	rechts
= += -= *= /= %= <<= >>= |= &= ^=	2	rechts
,	2	links

Abbildung 3: Alle Operatoren der Sprache C, in Reihenfolge ihrer Präzedenz (Bindungsstärke).

3.7 Operatoren (1)

C hat eine beträchtliche Menge von Operatoren, die in Abbildung 3 zusammengefaßt sind. Für unsere Zwecke sind zunächst insbesondere die folgende Operatoren wichtig:

- `+`, `*`, `/`, `-`: Die vier arithmetischen „Grundoperationen“
- `%`: Der Modulus-Operator (Divisionsrest)
- >>, <<: Bitschiebeoperationen
- && und ||, die “und” und “oder”-Operatoren

Aufgaben.

1. Schreiben Sie ein Programm, das feststellt, ob `*` tatsächlich stärker bindet als `+`. Erklären Sie, welches Ergebnis Sie in welchem Fall erwarten.
2. Schreiben Sie ein Programm, daß die Auswirkung der Bitschiebeoperationen auf Zahlen beobachtet. Welcher arithmetischen Operation entspricht das Rechts- bzw. Linksschieben bei positiven Zahlen?
3. Lesen Sie Foliensatz 5, Folien 37ff (`http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/folien-05.pdf`) um einen Überblick über weitere wichtige Operatoren (insbesondere die Zuweisungsoperatoren) zu erlangen.

3.8 Operatoren (2)

Wie wir gesehen haben, speichern Computer Zahlen intern in der Basis 2, als Binärzahlen. Daher sind bestimmte Operationen, die sich durch einfache binäre Hardwareoperationen erklären lassen, besonders effizient:

- Bitverschiebung (<<, >>)
- Bitweise ‘und’ (&)

- Bitweise ‘oder’ (|)
- Bitweise ‘exklusiv-oder’ (^)

In der Systemprogrammierung bevorzugen wir daher diese Operatoren gegenüber anderen Operationen, wenn sie anwendbar sind.

Um diese Operationen in Aktion zu sehen, ist es natürlich hilfreich, wenn wir die einzelnen Bits direkt sehen können. Mit `printf` verwenden wir dazu meist die Formatierungsspezifikation `%x`.

Aufgaben.

1. Bestimmen Sie per `printf` die korrekte Hexadezimaldarstellung der Dezimalzahl 1000.
2. Wie weit können Sie die Zahl 1 durch Linksschieben vergrößern? Bei welcher Anzahl von Bits erreichen Sie also die Schranke des Möglichen?
3. Beschreiben Sie, was passiert, wenn Sie eine zwei-ziffrige Hexadezimalzahl mit `0xf` verknüpfen:
   (a) Per Bitweise-Und
   (b) Per Bitweise-Oder
   (c) Per Bitweise-Exklusiv-Oder
4. Was passiert, wenn Sie eine Zahl zwei Mal mit der gleichen Zahl Exklusiv-Oder verknüpfen?

3.9 Lokale Variablen

C erlaubt Ihnen, Zwischenergebnisse und Zählvariablen in Variablen abzulegen:

```
int x;          // Deklaration: Variable ‘x’ mit Typ ‘int’
x = 1 + 2;      // Zuweisung eines Wertes an ‘x’

printf("%d", x); // ‘x’ wird ausgelesen und geschrieben

int y = 1 + 2; // Kurzfassung: Deklaration + Zuweisung
```

Beachten Sie dabei, daß jeder Variable ein expliziter Typ zugewiesen werden muß. Der Typ dient nur in beschränktem Maße Vermeidung von Programmierfehlern, dafür aber insbesondere den folgenden Zwecken:

- Festsetzung der Menge an Speicher, der für die Variable reserviert werden muß; so wird z.B. für den Typ `char` nur ein Byte reserviert, während für den Typ `int` mindestens zwei Bytes an Speicher reserviert werden.
- Bestimmung der Operationen, die auf dem Typ ausgeführt werden können. So unterscheidet der Prozessor z.B. zwischen Additionsoperationen für Fließkommazahlen und Additionen für Ganzzahlen; C stellt bei der Übersetzung automatisch fest, welche dieser beiden Assembleroperationen benötigt wird.

Die numerischen Typen von C sind in Abbildung 4 und Abbildung 5 zusammengefaßt. Typen mit mehr Speicherplatz können dabei größere bzw. genauere Zahlen darstellen. Beachten Sie, daß alle in C verfügbaren Typen eine feste Größe haben und somit immer nur begrenzte Größe und Genauigkeit erlauben. Nicht-numerische Typen sind komplexer; wir betrachten sie später.

Die Typen in Abbildung 4 repräsentieren Ganzzahlen (ohne Nachkommastelle). Diese Typen existieren in zwei Varianten: `signed` (Vorzeichenbehaftet) und `unsigned` (nicht vorzeichenbehaftet):

Typ	Seit	sizeof gemäß C99	Direktnotation
char		1 oder mehr	'a', '\n', '\013'
short int		2 oder mehr	
int		2 oder mehr	42, 0x2a, 0
long int		4 oder mehr	42l 0x2al, 0L
long long int	**C99**	8 oder mehr	42ll, 0x2all, 0LL

Abbildung 4: Die Ganzzahl-Typen der Sprache C in der Übersicht. Die Bedeutung von **sizeof** wird in Abschnitt 3.22 behandelt.

Typ	Spezifikation	sizeof gemäß Spezif.	Beispielwerte
float	IEEE-754 **binary32**	4	1.3f, 1.2e7f, 1.2e-7F
double	IEEE-754 **binary64**	8	1.3, 22.0e+23, 0x2a.e7p8
long double	**binary64** oder besser	8 oder mehr	1.3l, 22E23l, .5L

Abbildung 5: Die numerischen C-Typen in der Übersicht.

```
signed int x = -1;
unsigned int x = 0xa0000;  // Hexadezimalzahl
```

Wenn keine Angabe gemacht wird, geht der C-Übersetzer von `signed` aus. So kann man auf heutzutage handelsüblichen Rechnern z.B. von folgendem ausgehen:

- `signed char` kann Werte von -128 bis 127 annehmen
- `unsigned char` kann Werte von 0 bis 255 annehmen

Wir werden in dieser Veranstaltung Fließkommazahlen nur als Komponenten einiger Projekte im dritten Teil der Veranstaltung behandeln.

In C99 können Sie lokale Variablen überall dort deklarieren, wo Sie eine Anweisung angeben können. In C89 sind solche Deklarationen nur am Beginn eines Blocks (also den geschweiften Klammern, `{ ... }`) erlaubt.

Aufgabe. Stellen Sie über die Definition einer lokalen Variable fest, was die größten bzw. kleinsten erlaubten Werte von `signed short` und `unsigned short` in Ihrer Implementierung sind.

3.10 If und Kommandozeilenargumente

Die `if`-Kontrollstruktur erlaubt die bedingte Ausführung eines Blocks, zum Beispiel:

```
if (x > 0) {
    // wird nur ausgefuehrt, wenn x > 0 ist:
    puts("x ist groesser als 0");
} else {
    puts("x ist kleiner oder gleich 0");
}
```

Der **else**-Teil wird ausgefuehrt, wenn die Bedingung `x > 0` nicht wahr ist. Dieser Teil ist optional:

```
if (x > 0) {
    puts("x ist groesser als 0");
}
```

Strenggenommen sind in diesem Fall auch die geschweiften Klammern, die zum Gruppieren von Anweisungen in einen Block verwendet werden, hier optional:

```
if (x > 0)
    puts("x ist groesser als 0");
```

Aber wir empfehlen Ihre Verwendung, um verwirrende Fälle wie den folgenden zu vermeiden:

```
if (x > 0)
    puts("x ist groesser als 0");
    puts("Dieser puts wird IMMER ausgefuehrt!");
```

Also besser:

```
if (x > 0) {
    puts("x ist groesser als 0");
    puts("Dieser puts wird nur noch bei x > 0 ausgefuehrt.");
}
```

Bedingungen können beliebige C-Ausdrücke sein. C kennt intern keine dedizierten booleschen Werte (true, false) sondern sieht eine Bedingung als „wahr“ oder „erfüllt“ an, wenn beim Ausrechnen der Bedingung das Ergebnis ungleich 0 ist. Für Vergleiche ist diese Eigenschaft implizit erfüllt, aber auch andere Ausdrücke sind möglich. So ist folgendes z.B. legales C:

```
if (0) {
    // wird nie passieren
}
```

Vorsicht: Aufgrund der obengennanten Eigenschaft sind einige C-Ausdrücke gültige Ausdrücke in `if`-Bedindungen und ähnlichen Strukturen, obwohl sie keine echte Überprüfung eines Wahrheitswertes sind, so z.B.:

```
// FALSCH:  x = 0 ist eine ZUWEISUNG, kein Vergleich!
if (x = 0) {
    printf("x ist 0");
}
```

Im obigen Programm ist ein Tippfehler: Statt des Vergleiches `x == 0` („ist x gleich 0?“) enthält dieses Programm eine *Zuweisung*, die `x` immer auf 0 setzt.

```
// RICHTIG:  x == 0 ist ein Vergleich!
if (x == 0) {
    printf("x ist 0");
}
```

Wir werden diese Kontrollstruktur nun zum Auswerten der Kommandozeilenargumente verwenden. Bisher haben wir unser C-Programm immer mit der „magischen“ Zeile

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

begonnen. Diese Zeile markiert den Beginn einer bestimmten Funktion; ein Konzept, auf das wir allgemeiner noch im nächsten Abschnitt eingehen werden. Die hier angegebene Funktion heißt `main`, was gemäß Konvention des UNIX-Systems der Startpunkt eines jeden C-Programmes ist. Die weiteren Komponenten der Zeile haben folgende Bedeutung:

- Das `int` links von `main` signalisiert, daß `main` einen Wert vom Typ `int` als Ergebnis zurückliefert. Diesen Wert beschreiben wir etwas genauer in Abschnitt 3.25.
- `int argc` gibt an, wieviele Kommandozeilenparameter dem Programm übergeben wurden.

- `char** argv` gibt an, welche Kommandozeilenparameter genau dem Programm übergeben wurden. Der Typ `char**` ist etwas komplexer; wir berachten ihn genauer in Abschnitt 3.16. Für die Zwecke dieser Aufgabe ist es ausreichend, wenn wir wissen, daß `argv[0]`, `argv[1]` usw. Zeichenketten sind.

Aufgaben.

1. Was ist der Inhalt der Zeichenkette `argv[0]` ?
2. Schreiben Sie ein Begrüßungsprogramm, das feststellt, ob es mit mindestens einem Kommandozeilenparameter aufgerufen wurde. Wenn nicht, soll das Programm eine Zeichenkette ausgeben, die nach dem Namen des Benutzers fragt. Ansonsten soll es davon ausgehen, daß der Benutzer den Namen als ersten Parameter angegeben hat namentlich begrüßen (also z.B. „Hallo, creichen!“, wenn das Programm `h` heißt und von der Shell aus als `./h creichen` aufgerufen wurde).

3.11 Funktionen in C

Außer der `main`-Funktion können C-Programme fast beliebig viele weitere Funktionen definieren. Hier ist ein Beispiel einer *Funktionsdefinition*:

```
int add(int x, int y)
{
    return x + y;
}
```

Diese Funktion wird zwei Zahlen addieren. Wir können sie von unserer `main`-Funktion aus aufrufen:

```
int main(int argc, char **argv)
{
    printf("%d\n", add(1, 2));
    return 0;
}
```

Aber Vorsicht: dies ist nur gestattet, wenn die Definition von `add` vor der Definition von `main` erfolgt. Falls das nicht möglich ist (z.B. bei gegenseitiger Rekursion) kann man den *Prototypen* der Funktion *deklarieren*:

```
int add(int x, int y);
```

Eine solche Deklaration sagt dem Übersetzer, wie die Funktion aussehen wird, aber nicht, was sie genau tut. Ein Programm kann beliebig viele Deklarationen für eine Funktion haben, aber immer nur eine Definition.

Weitere Informationen finden Sie in Folien 24–30 des Foliensatzes 6 (`http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/folien-06.pdf`).

Aufgabe. Implementieren Sie den Euklidschen Algorithmus zur Bestimmung des kleinsten gemeinsamen Vielfachen durch rekursive Funktionsaufrufe. In dieser wie in allen folgenden Aufgaben gilt, daß Sie die Korrektheit Ihrer Funktionen durch Tests belegen (aber nicht beweisen) müssen.

3.12 For-Schleife

For-Schleifen in C sind ein sehr mächtiges Konstrukt zur Wiederholung von Operationen. Lesen Sie dazu zunächst Foliensatz 5, Folien 12–14 (`http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/folien-05.pdf`).

Implementieren Sie ein Programm, das die ersten 20 Fibonacci-Zahlen berechnet und ausgibt. Zur Erinnerung: Die n-te Fibonacci-Zahl $\mathcal{F}_n$ berechnet sich wie folgt:

n	$\mathcal{F}_n$
0	0
1	1
n	$\mathcal{F}_{n-1} + \mathcal{F}_{n-2}$

Verwenden Sie dazu keine rekursiven Funkionsaufrufe, sondern eine `for`-Schleife, und den `++`-Operator.

3.13 Arrays

Es ist in C möglich, einen Speicherblock mit mehreren konsekutiven Einträgen gleichzeitig zu allozieren, ein sogenanntes Array:

```
int a[10];  // 10 Elemente:  Index 0 bis 9
a[0] = 1;
a[1] = 2;
a[2] = 4;
for (int i = 3; i < 10; i++) {
    a[i] = (i+1) * (i+1);
}
```

Zugriff erfolgt, wie in diesem Beispiel, über Indexnotation mit eckigen Klammern. Ungleich anderen Sprachen wie Python oder Java merkt sich C allerdings nicht zur Laufzeit, wie groß das Array ist; diese Information müssen Sie ihn Ihrem C-Code separat speichern.

Beachten Sie, daß C die Arraygröße bei Zugriffen nicht überprüft. Wenn Sie also folgendes schreiben:

```
int a[10];
a[999] = 10;
a[-1] = 10;
```

dann wird ihr C-Programm wahrscheinlich zunächst ausgeführt werden, aber auf subtile Art und Weise den Speicher modifizieren. Dies kann später zu Programmabstürzen, falschen Ergebnissen, oder anderen Problemen führen.

Um Arrays als Parameter an Funktionen zu übergeben, können Sie die Notation

```
int f(int array[]);
```

nutzen (also Klammern ohne Größenangabe).

Aufgaben.

1. Schreiben Sie eine Funktion, die das größte Element eines `int`-Arrays bestimmt.
2. Testen Sie Ihre Funktion mit Arrays verschiedener Größen.

3.14 Typkonvertierungen

Nicht immer sind die Werte, die Sie erhalten, in dem Typ, den Sie sich wünschen. C erlaubt Ihnen, in begrenztem Maße zwischen Typen zu konvertieren:

```
int x = (int) 4.3;  // Von Fliesskomma nach int
```

Diese Konvertierungen können teilweise auch implizit stattfinden:

```
int x = 42;
double d = x;   // implizite Konvertierung
```

Beachten Sie auch Folie 35 in Foliensatz 5 (`http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/folien-05.pdf`).

Aufgaben.

1. Konvertieren Sie eine Fließkommazahl nach `int`. Wie wird gerundet? beachten Sie, daß dieses Detail bei verschiedneen C-Implementierungen unterschiedlich sein kann.

2. Konvertieren Sie die Zahl `255` in ein `signed char` und geben Sie dies aus (Konvertierungsspezifikation `%d` in `printf`). Was ist geschehen?

3. Können Sie auf diese Weise eine Zeichenkette (`char *`) nach `int` konvertieren?

4. Können Sie auf diese Weise eine `int`-Zahl in eine Zeichenkette konvertieren?

3.15 Zeiger und Adreßoperatoren

In C hat der Typ τ `*` (für einen beliebigen Typen τ) die Bedeutung „Zeiger auf ein (oder mehrere) τ". Ein solcher Zeiger ist intern nichts anderes als eine Speicheradresse; C bietet viele Operationen an, um mit solchen Zeigern effizient zu arbeiten.

```
int a = 0;
int *b;
b = &a;
```

In dem obigen Programm ist der Inhalt der Variable `b` nun die Speicheradresse der Variable `a`. Wir sagen auch, daß `b` auf `a` *zeigt*, oder daß `b` ein *Zeiger auf* `a` ist.

Der obige Operator `&` ist der sogenannte *Adreßoperator*. Er gibt die Speicheradresse des Objektes zu seiner Rechten zurück. Natürlich kann der Adreßoperator nur auf Objekte angewendet werden, die auch eine Speicherstelle haben (Variablen, Arrays, Arrayelemente, Funktionen, ...). Nicht erlaubt ist z.B. der Ausdruck `&10`, da die Zahl 10 ein Wert ist und keine eigene Speicheradresse hat. Beachten Sie allerdings:

- Es ist durchaus möglich, die Zahl 10 im Speicher abzulegen (insbesondere in einer Variablen). Diese *Variable* hat nun wiederum eine Speicheradresse.

- In einigen Systemen ist 10 selbst eine gültige Adresse.

Diese drei Fälle unterscheiden sich wesentlich voneinander: der erste Fall beschreibt einen *Wert*, der zweite Fall einen *Speicherinhalt*, und der dritte Fall eine *Speicheradresse* (wir beschreiben weiter unten, wie wir Speicheradressen, die uns C normalerweise automatisch zuweist, ausgeben können.)

Ein Zeiger kann den Inhalt der Speicherstelle, auf die er zeigt, manipulieren:

```
int a = 0;
int *b;
b = &a;

*b = 1;
printf("%d\n", a);  // gibt 1 aus
printf("%d\n", *b); // gibt ebenfalls 1 aus
```

Der Dereferenzierungsoperator `*` erlaubt uns direkten Zugriff auf den referenzierten Speicherinhalt; er ist das Gegenstück zum Adreßoperator:

```
int a = 42;
printf("%d\n", *(&a));  // gibt 42 aus
```

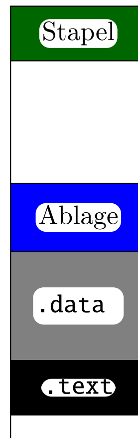


Abbildung 6: Adreßraum von Programmen der Sprache C auf den meisten modernen Rechnern. Die Adressen werden von unten nach oben größer. `text` bezeichnet ausführbaren Maschinencode. Der `data`-Bereich speichert statische Daten. Beide Bereiche haben eine beim Start des Programmes festgelegte Größe; sie stellen den *statischen Speicher* dar. *Ablage* stellt den Ablagespeicher (heap) dar, der nach oben hin wachsen kann; er dient als dynamischer Speicher. *Stapel* stellt den Aufrufstapel dar. Er wächst nach unten, schrumpft aber wieder, wenn Funktionen eine **`return`**-Anweisung ausführen; er dient als *stapeldynmamischer Speicher*.

Aufgaben.

1. Sie können die `printf`-Konvertierungsspezifikation `%p` verwenden, um die Adresse eines Zeigers auszugeben. Geben Sie (a) die Adresse einer Funktion, und (b) die Adresse einer lokalen Variable aus. Vergleichen Sie die Werte.

2. Schreiben Sie eine Funktion `swap`, die Zeiger auf zwei `int`-Variablen nimmt und die Inhalte dieser Variablen austauscht.

3.16 Speicher

Das C-Speichermodell erbt all die Stärken und Schwächen des Assembler-Modells: Es ist sehr flexibel, aber auch fehleranfällig.

C strukturiert den Hauptspeicher auf eine bestimmte Art und Weise, die den Stärken des verwendeten Prozessors Rechnung trägt. Dabei unterteilt es den Speicher in drei Kategorien:

- *Statischer Speicher*: Dieser Speicher merkt sich den Maschinencode des C-Programmes und die globalen Variablen (sofern solche existieren; in unseren bisherigen Beispielen haben wir auf globale Variablen verzichtet).

- *Stapeldynamischer Speicher*: Dieser Speicher merkt sich lokale Variablen, Funktionsparameter, und *Rücksprungadressen*, die nötig sind, um von einem Funktionsaufruf wieder zum korrekten Aufrufenden zurückkehren zu können. Der Speicher wächst, wenn das Programm Funktionen aufruft, und schrumpft, wenn diese Funktionen zurückkehren.

- *Dynamischer Speicher*: Diese Speicherform ist die flexiblelste Art von Speicher. Sie muß explizit angefordert und auch wieder freigegeben werden.

Abbildung 6 stellt dar, wo diese Speicherbereiche relativ zueinander liegen.

3.16.1 Statischer Speicher

Statischer Speicher wird automatisch für alles reserviert, was zu diesem Speicher gehört, insbesondere für den Maschinencode. Wenn wir eine Variable außerhalb einer Funktion definieren:

```
int i;  // globale Variable

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

oder als `static` definieren:

```
int f()
{
    static int counter = 0;
    return counter++;
}
```

dann wird diese Variable im statischen Speicher abgelegt. Das heißt insbesondere, daß diese Variable nur ein einziges Mal existiert; die oben angegebene Funktion `f` wird z.B. bei jedem Aufruf einen anderen Wert zurückliefern, da dieser statische Speicher bei jedem Aufruf bestehen bleibt und hochgezählt wird.

Beachten Sie im Gegensatz dazu die Funktion `g`:

```
int g()
{
    int counter = 0;
    return counter++;
}
```

Diese Funktion wird immer 0 zurückliefern, da bei jedem Aufruf eine neue Variable `counter` auf dem Stapel alloziert wird, wie wir als nächstes beschreiben.

3.16.2 Stapeldynamischer Speicher

Bei jedem Funktionsaufruf wächst der Ablagestapel nach unten[6]: Funktionsparameter und Rücksprungadresse werden auf den Stapel gelegt, und das Programm alloziert zusätzlichen Platz für lokale Variablen, soweit benötigt. Wenn wir also eine rekursive Funktion wie z.B. den Euklidschen Algorithmus nehmen, können wir beobachten, wie bei jedem rekursiven Aufruf die lokalen Daten weiter „unten" im Speicher abgelegt werden.

Abbildung 7 stellt eine solche rekursive Funktion dar. Hier sehen wir, wie bei jedem rekursiven Funktionsaufruf (gekennzeichnet durch unterschiedliche Farben) der Stapel weiter nach unten wächst, bis die Funktion schließlich vom Aufruf zurückkehrt. Die aktuelle Tiefe des Stapels wird dabei in einem *Stapelzeigerregister* auf dem Prozessor gespeichert.

3.16.3 Dynamischer Speicher

Die dritte und letzte vom C-Laufzeitsystem direkt angebotene Form des Speichers ist *dynamischer Speicher*. Dieser Speicher muß explizit mit einem der folgenden Befehle angefordert werden, bevor man ihn im C-Programm verwenden kann:[7]

[6] Einige Architekturen verwenden eine „umgekehrte" Architektur, bei der der Ablagespeicher nach unten und der Stapelspeicher nach oben wachsen; die Konzepte sind aber exakt analog.

[7] C++-Programmierer: Diese Operationen sind *nicht* austauschbar mit **new** und **new[]**. Letztere müssen zusammen mit **delete** verwendet werden, während `malloc`/`calloc`/`realloc` zusammen mit `free` verwendet werden müssen.

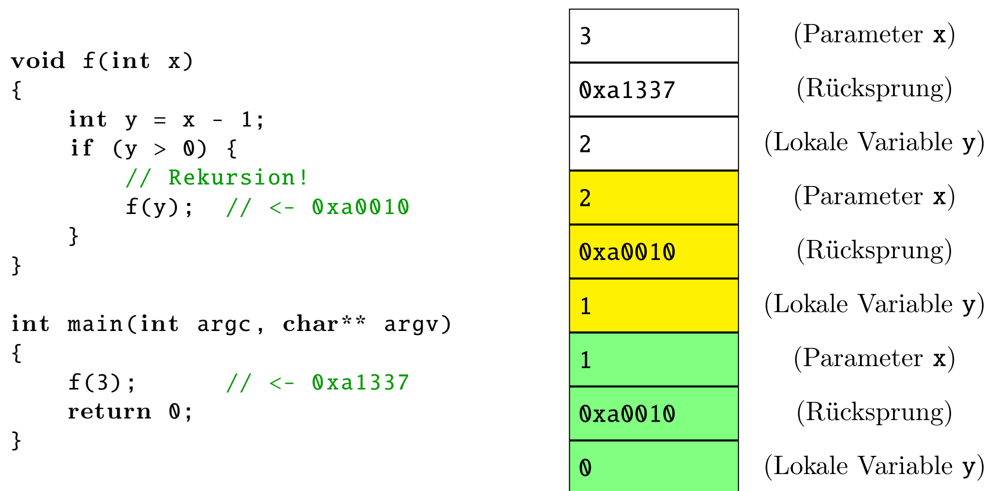


Abbildung 7: Dynamischer Speicher wächst während eines Funktionsaufrufes nach unten. Wenn die Funktion zurückspringt, wird der so allozierte Speicher automatisch freigegeben.

- `malloc(n)`
- `calloc(n, m)`

Diese Befehle allozieren n bzw. $n \times m$ Bytes. Der Unterschied zwischen ihnen ist, daß `calloc` den allozierten Speicher mit `0`-Bytes vorinitialisiert (tatsächlich bemüht sich die Funktion, Speicher zu finden, der bereits auf `0` gesetzt ist, da dies effizienter ist), während bei `malloc` der Inhalt des allozierten Speichers undefiniert ist.

Diese Funktionen liefern einen Wert vom Typ `void *` zurück. In C wird `void *` verwendet, um auszudrücken, daß wir einen Zeiger auf ein Objekt 'von unbekanntem Typ' haben.

Um nun in diesem Speicher einen bestimmten Wert ablegen zu können, müssen wir den Zeiger zunächst konvertieren, z.B. explizit durch::

```
int *zahlen = (int *) calloc(sizeof(int), 10);
```

Diese Allozierung (per `calloc`) reserviert genug Platz für 10 Integer-Zahlen. Betrachten wir genauer, was hier geschieht:

- `sizeof(int)` berechnet die Größe des Typs `int` in Bytes. Diese Berechnung findet bereits zur Übersetzungszeit statt; `sizeof` ist ein eingebauter Operator.
- `calloc` reserviert genug Platz für 10 `int`-Werte ($10 \times$ `sizeof(int)`) und gibt dies als `void *` zurück.
- `(int *) ...` konvertiert den Rückgabewert von `calloc` von `void *` nach `int *`, in einen Zeiger auf `int`-Zahlen.

Beachten Sie, daß *intern* die Konvertierung keinen Effekt hat– sowohl vorher als auch nachher ist der Zeiger eine Adresse auf eine bestimmte Stelle im Speicher. Die Konvertierung hilft allerdings dem C-Typsystem, falsche Warnungen und Fehlermeldungen zu vermeiden.

Sie können auf `zahlen` nun mit dem Dereferenzierungsoperator zugreifen, wobei dies nur den Zugriff auf das erste Element des Speicherbereiches erlaubt. Alternativ können Sie auch Arraynotation verwenden:

```
    *zahlen = 0;
    zahlen[0] = 0;  // gleichwertig zur obigen Operation
    zahlen[9] = 9;

    zahlen[-1] = -1;  // Speicherfehler, Vorsicht!
```

`malloc` und verwandte Operationen werden zur Verfügung gestellt, wenn Sie folgende Direktive zu Anfang Ihres Programmes angeben:

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

3.16.4 Speicherfreigabe

Mit `malloc` allozierter Speicher kann wieder freigegeben werden, wenn er nicht mehr benötigt wird. Dies müssen Sie *explizit* veranlassen, mit der Operation

```
free(void *p)
```

Wenn nicht mehr verwendeter Speicher nicht explizit freigegeben wird, bleibt er weiterhin reserviert; man spricht von einem *Speicherleck*. Wenn ein Speicherleck in einer Schleife oder Rekursion auftritt, kann dies schnell dazu führen, daß dem Programm der Speicher ausgeht, oder daß es anderen Programmen indirekt Speicher raubt und somit das System verlangsamt. Speicherlecks gehören zur Klasse der Speicherfehler und sind oft schwer zu finden; wir werden später Programme betrachten, die uns bei der Suche nach diesen Fehlern helfen können.

Hier nun ein Beispiel mit `free`:

```
#include<stdlib.h>

int main(int argc, char **argv)
{
    int *zahlen = (int *) calloc(sizeof(int), 10);

    for (int i = 0; i < 9; i++) {
        zahlen[i] = i*i;
    }

    free(zahlen);
    return 0;
}
```

Lesen Sie dazu zunächst Foliensatz 6, Folien 14–21 (`http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/folien-06.pdf`); die Betrachtungen zu **`struct`** können Sie zunächst überspringen.

Aufgaben.

1. Lesen Sie die `man`-Seite zu `realloc`. Schreiben Sie ein Programm, das Allozierungen durchführt und die von `realloc` zurückgegebenen Adressen ausgibt.

   Schreiben Sie ein Programm, in dem `realloc` die gleiche Adresse zurückgibt, die Sie als Parameter gegeben haben, und eines in dem `realloc` eine andere Adresse zurückgibt.

2. Implementieren Sie den *Sieb des Eratosthenes* in C. Schreiben Sie dazu eine Funktion, die die größte nach Primzahlen zu durchsuchende Zahl als Parameter nimmt und per `printf` alle Primzahlen bis maximal der angegebenen Zahl ausdruckt. Allozieren Sie den Sieb per `malloc`, und vergessen Sie nicht, ihn wieder freizugeben!

3.17 Zeichenketten

Zeichenketten in C haben den Typ `char *`, sind also Zeiger auf ASCII-Zeichen bzw. Arrays von Zeichen. Zeichenketten können also per Indexnotation ausgelesen und beschrieben werden. Eine Besonderheit dabei ist das letzte Zeichen der Zeichenkette: dies hat den numerischen Wert 0, ein Sonderzeichen, dem keine ASCII-Bedeutung zugeordnet ist. Dieser *Terminator* markiert also das Ende der Zeichenkette und kann somit z.B. zur Bestimmung der Länge dieser Zeichenkette verwendet werden.

Entsprechend ist jeder Zeichenkette ein fester Bereich im Speicher zugeordnet: Änderungen in diesem Bereich sind legitim, aber das Schreiben über das Ende der Zeichenkette hinaus

Aufgaben.

1. Lesen Sie die `man`-Seiten der folgenden Befehle:
   - `strlen`
   - `strcat`
   - `strcpy`
2. Welche Länge hat die Zeichenkette "ABC"? Wieviele Bytes nimmt sie im Speicher ein?
3. Nehmen Sie das folgende Programm:

```
char a[] = "Anfang";
char b[] = "aba:";
char c[] = ":bab";
char z[] = "Ende";
```

   und erweitern Sie es, um folgendes auszuführen:
   - Nehmen Sie den ersten (vom Benutzer angegebenen) Kommandozeilenparameter, und hängen Sie ihn zwischen die Zeichenketten `b` und `c`.
   - Ersetzen Sie der Ergebniszeichenkette jedes Vorkommen eines `'a'` durch ein `'_'`.
   - Geben Sie die Zeichenkette `a` aus.
   - Geben Sie die Ihre modifizierte Zeichenkette aus.
   - Geben Sie die Zeichenkette `z` aus.
   - Beschreiben Sie, wo Sie dynamischen Speicher alloziert haben, falls dies nötig war. Haben Sie auch allen allozierten Speicher wieder mit `free` freigegeben?

3.18 `const`-Typen

C unterstützt einige *Typqualifizierer*, die die Bedeutung von Typen verändern; beachten Sie dazu insbesondere Folien 34 und 35 in Foliensatz 7 (`http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/folien-07.pdf`).

Aufgaben.

1. Schreiben Sie eine Funktion, die Parameter `p` des Typs `const int *` nimmt. Überprüfen Sie, ob der Übersetzer Ihnen folgendes erlaubt:
   (a) Zuweisung auf `p`, so daß `p` auf eine andere Speicherzelle zeigt
   (b) Änderung der Speicherzelle, auf die `p` zeigt
2. Versuchen Sie gleiches für den Type `int const *`.
3. Versuchen Sie gleiches für den Type `int * const`.

3.19 **switch**

C verfügt über ein Konstrukt, das eine Unterscheidung nach mehr als zwei Fällen durchführen kann, das sogenannte **switch**-Konstrukt. Betrachten Sie das folgende Beispiel:

```
switch (x) {
    case 0:
      printf("null\n");
    case 1:
      printf("null oder eins\n");
      break;
    case 2:
      printf("zwei\n");
      break;
    default:
      printf("Andere Zahl\n");
}
```

Hier wird der Inhalt der Variable `x` untersucht. Abhängig vom Inhalt der Variable springt **switch** in die mit **case** `0`, **case** `1`, oder **case** `2` markierte Zeile, oder in die mit **default** markierte Zeile, falls `x` weder 0 noch 1 noch 2 ist. C führt dann die unmittelbar folgenden Befehle aus:

- Falls `x == 0` ist, wird „`null`" ausgeben, und danach „`null oder eins`"— denn nach Beendung eines **case**-Falles führt C die Ausführung in der nächsten Zeile weiter, auch wenn dazwischen ein anderes **case**-Konstrukt liegt (!). Erst beim Erreichen von **break** wird die Bearbeitung abgebrochen.
- Falls `x == 1` ist, wird entsprechend nur „`null oder eins`" ausgegeben.
- Falls `x == 2` ist, wird „`zwei`" ausgegeben.
- Bei allen anderen Werten wird „`Andere Zahl`" ausgegeben.

Als **case**-Werte sind nur Zahlen oder Buchstaben (`char`-Werte) erlaubt.

Aufgabe. Iterieren Sie über alle Zeichen des ersten vom Benutzer übergebenen Kommandozeilenparameters, und ersetzen Sie alle 'x' durch 'u' und 'u' durch 'x', bevor Sie diesen wieder ausgeben. Verwenden Sie dazu **switch**.

3.20 Wiederholungen mit **while**

Zusätzlich zu **for**-Schleifen unterstützt C zwei weitere Schleifenarten:

```
while (bedingung) {
    ... // Schleifenkoerper
}

do {
    ... // Schleifenkoerper
} while (bedingung);
```

Diese Schleifen wiederholen den Schleifenkörper so lange, bis die Bedingung nicht mehr wahr ist (Bedindungen sind analog zu Abschnitt 3.10). Der Unterschied dabei ist, daß eine **while** `(b) { ...}`-Schleife den Körper nicht ausführt, wenn `b` zu Beginn nicht wahr ist, während eine `Ckwdo { ...} Ckwwhile (b);`-Schleife den Körper immer mindestens ein Mal ausführt.

Beachten Sie zu Kontrollstrukturen auch Folie 23–25 in Foliensatz 5 (`http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/folien-05.pdf`), insbesondere bezüglich der Schlüsselwörter **break** und **continue**.

Eine Übung zu `while` finden Sie im nächsten Abschnitt.

3.21 Eingabe mit `scanf`

Analog zur formatierten Eingabe mit `printf` kann man auch formatiert einlesen; die korrespondierende Funktion heißt `scanf`.

Probieren Sie das folgende Programm aus:

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int v;
    scanf("%d", &v);
    printf("%d\n", v * v);
    return 0;
}
```

Es liest einen Zahlenwert ein und druckt dessen Quadrat aus. Beachten Sie, daß Sie an die `scanf`-Funktion keine Zahlen oder Berechnungen übergeben können, sondern nur Adressen: die `scanf`-Funktion muß ihre Ergebnisse an eine geeignete Stelle im Speicher schreiben, und genau diese Speicherstelle geben Sie durch die Parameteradresse an.

Aufgabe. Implementieren Sie mittels `scanf` und **`while`** eine einfache Variante des Mondlandespiels. In diesem Spiel müssen Sie eine Kapsel auf dem Mond landen, ohne an diesem zu zerschellen. Erschwert wird die Landung dadurch, daß Sie nur begrenzten Treibstoff an Bord haben. Das Spiel verwendet drei Variablen:

- Höhe
- Vertikale Geschwindigkeit
- Verbleibender Treibstoff

Das Spiel läuft, solange die Höhe größer als 0 ist. Wenn die Höhe 0 (oder niedriger) erreicht wird, wird das Spiel beendet; wenn zu diesem Zeitpunkt die Geschwindigkeit größer als 3 ist, zerschellt die Kapsel, ansonsten war die Landung erfolgreich.

- Jede Runde (Schleifendurchlauf) wird die aktuelle Höhe zusammen mit dem verbleibenden Treibstoffreserve ausgegeben.
- Der Benutzer gibt danach einen Schub-Wert zwischen 0 und 4 (inklusive) ein.
- Dieser Schub wird von der Treibstoffreserve abgezogen, sofern dies den Treibstoffstand nicht unter 0 bringen würde.
- Falls Treibstoff verbraucht wurde, wird der Schubwert von der Geschwindigkeit abgezogen.
- Unabhängig davon wird die Geschwindigkeit um 2 erhöht und von der Höhe abgezogen.

Das Spiel kann durch zufällige Auswahl einiger der Parameter, Echtzeitausführung, oder Graphik natürlich noch interessanter gestaltet werden, aber die betreffenden Techniken gehen über den Rahmen dieses Praktikums hinaus.

3.22 Beschränkungen der eingebauten Datentypen im Speicher

Der schon zuvor erwähnte eingebaute Operator **`sizeof`** nimmt als Parameter sowohl Typen (z.B. `int`) als auch Variablen. Er liefert eine Zahl zurück, die der Byte-Größe des entsprechenden Typs entspricht.

Aufgabe. Bestimmen Sie: Welche Größe in Bytes haben auf Ihrem Rechner:

- `int`
- `long int`
- `long long int`
- `char *`

3.23 struct und union

C erlaubt die Gruppierung von zusammenhängenden Informationen in *Strukturen*, ausgedrückt durch **struct**s[8]; lesen Sie dazu zunächst Folien 9–12 und 18–20 in Foliensatz 06 (`http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/folien-06.pdf`).

Ein **struct** erlaubt also, mehrere hintereinanderfolgende Speicherstellen für verschiedene Felder zu reservieren, um auf diese dann einheitlich zugreifen zu können. Betrachten Sie z.B. die folgende Struktur:

```
struct st {
    int x;
    char *y;
};

struct st s;
```

Hier haben `s.x` und `s.y` eine jeweils eigene Speicheradresse; C garantiert dabei, daß `s.x` vor `s.y` im Speicher liegt.

Eine **union** ist syntaktisch sehr ähnlich wie ein **struct**:

```
union u {
    int ux;
    char *uy;
};

union u ut;
```

Der Unterschied dabei ist, daß alle Felder einer **union** die *gleiche* Speicheradresse haben. Insbesondere werden die Inhalte eines Feldes wie z.B. `ut.ux` überschrieben, wenn in ein anderes Feld der gleichen **union**, z.B. `ut.uy`, geschrieben wird. Der Nutzen einer **union** erschließt sich in Situationen, in denen wir *entweder* den einen Eintrag *oder* den anderen Eintrag benötigen, aber nie beide gleichzeitig.

Aufgaben.

1. Betrachten Sie die obige Beispiel-Struktur `struct st` und Variable `s`. Wie weit im Speicher sind die Felder `x` und `y` voneinander entfernt?
2. Erstellen Sie ein **struct** mit zwei `int`-Einträgen und zwei `char`-Einträgen. Wie groß ist das **struct** in Bytes? Ändert sich die Größe, wenn Sie die Einträge umsortieren?
3. Dynamische Programmiersprachen wie Python erlauben Variablen oft, Werte beliebiger Typen anzunehmen. So ist folgendes in Python völlig legitim:

[8] **struct** in C und C++ unterscheiden sich wesentlich: Eine C-**struct**-Definition (z.B. `struct n { ...}` führt einen neuen **struct**-Namen (z.B. `n`) ein; um eine entsprechende **struct**-Variable zu deklarieren, muß man dies ausschreiben als `struct n`. In C++ ist **struct** eine Alternative zu **class** mit Unterschieden bezüglich der Sichtbarkeit von Feldern (ein Konzept, das in C so nicht existiert). In C++ führt **struct** also nicht einen **struct**-Namen ein, sondern einen Typnamen.

```
if x > 0:
    v = "string"
else:
    v = 42
```

`v` kann also entweder eine Zeichenkette oder eine `int`-Zahl sein. Die Zahl kann danach mit sich selbst verknüpft werden:

```
v = v + v
```

Dabei wird die +-Operation als Addition interpretiert, wenn `v` eine Zahl ist, und als eine Zeichenketten-Konkatenierung, wenn `v` eine Zeichenkette ist.

Implementieren Sie einen Datentyp, der entweder `int` oder `char *` ist, und sich zugleich merkt, welches der beiden er ist. Implementieren Sie dann eine Additionsoperation für diesen Typen, der das Verhalten der Python-Bibliothek simuliert (sie können dabei das Problem der Speicherlecks ignorieren). Ihre Additionsoperation soll einen neuen Wert dieses Typs mit dem entsprechenden Ergebnis zurückliefern. Falls die beiden Eingabetypen ungleich sind, soll er `NULL` zurückliefern.

3.24 Generische Operationen

C unterstützt Zeiger auf alle Dinge, die im Hauptspeicher liegen können. Da der Maschinencode von Funktionen ebenfalls im Hauptspeicher liegt, erlaubt C auch Zeiger auf diesen:

```
int f(unsigned, char);

int main(int argc, char **argv) {
    int (*p)(unsigned, char);
    p = &f;
    (*p)(0, 'x');  // Funktionsaufruf
}
```

Aufgabe. Implementieren Sie den Bubblesort-Algorithmus. Dieser Algorithmus führt eine *Austauschphase* aus, die jedes Element eines Arrays mit dessen Nachfolger vergleicht. Wenn der Nachfolger kleiner ist, werden die beiden Elemente vertauscht. Der Algorithmus wiederholt die Austauschphase so lange, bis das Array sich nicht mehr verändert.

Ihre Implementierung des Algorithmus sollte folgendem C-Prototypen entsprechen:

```
void bubblesort(int *base,
                int size,
                int (*compar)(int, int));
```

Hierbei ist `base` der Zeiger auf das erste Element des Arrays, `size` die Anzahl der Elemente, und `compar` ein Zeiger auf eine Funktion, die zwei Elemente a, b vergleicht und folgendes zurückliefert:

- Einen Wert < 0, wenn $a < b$
- 0, wenn $a = b$
- Einen Wert > 0, wenn $a > b$

Indem wir verschiedene `compar`-Funktionen einsetzen, können wir also ein Array auf verschiedene Arten und Weisen sortieren (z.B. in numerisch aufsteigender oder absteigender Reihenfolge, sortiert nach Anzahl der gesetzten Bits, sortiert nach Distanz zu einem bestimmten Wert usw.)

Demonstrieren Sie, daß Ihre bubblesort-Implementierung mit numerisch aufsteigender und numerisch absteigender Sortierung umgehen kann.

3.25 Fehlerbehandlung

C hat keinen Ausnahmebehandlungsmechanismus wie Python oder Java. Stattdessen werden Fehler üblicherweise über Rückgabewerte der Funktionen und Programme angegeben. Die speziellen Rückgabekonventionen dabei variieren stark zwischen verschiedenen Operationen. Der Grund dafür ist, daß C sich sehr um Effizienz bemüht, und effiziente Kodierung von der Problemdomäne abhängig ist.

Für die (benutzerdefinierte) `main`-Funktion gilt die Konvention, daß:

- `0` bei erfolgreicher Ausführung zurückgegeben wird
- `1` bei einem Fehler während der Ausführung zurückgegeben wird.

Die Ausführung des Programmes kann auch direkt mit dem Befehl `exit` abgebrochen werden; diese Funktion kehrt nicht zurück und hat aus Sicht der Shell (oder eines anderen aufrufenden Programmes) den gleichen Effekt wie ein **`return`** von der `main`-Funktion aus. Insbesondere die Shell verwendet diese Rückgabewerte, um Verknüpfung von Befehlen mit anderen Befehlen zu erlauben und Fehler in solchen Verknüpfungen zu behandeln.

Aufgabe. Lesen Sie die `man`-pages der angegebenen Funktionen und untersuchen Sie, wie diese Funktionen Fehler signalisieren:

- `printf`
- `scanf`
- `malloc`

3.26 Der Präprozessor

Der Präprozessor ist ein Instrument zur Vorverarbeitung des Programmes, bevor es an den Übersetzter weitergegeben wird; der Präprozessor behandelt insbesondere `#include`-Direktiven, indem er die dort genannten Dateien als Text in das Programm integriert.

Lesen Sie sich zunächst Foliensatz 7 bis einschließlich Folie 12 durch (`http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/folien-07.pdf`).

Sie können die Ausgabe des C-Präprozessors `cpp` direkt ausgeben lassen, indem Sie `gcc -E <programm>` aufrufen.

Aufgaben.

1. Lassen Sie sich die Ausgabe des Präprozessors zu einem Ihrer bereits geschriebenen Programme ausgeben. Messen Sie, um wie viel Ihr Programm länger wird.

   Sie können die Länge einer Datei `d` auf der Shell wie folgt bestimmen:

   ```
   cat d | wc -L
   ```

   `cat` gibt hierbei das Programm aus, `wc -L` zählt die Zeilen, die es als Eingabe erhält, und der vertikale Strich | (das sogenannte *pipe*-Symbol) verbindet die Ausgabe des links davon stehenden Programmes (`cat`) mit der Eingabe des rechts davon stehenden Programmes. Versuchen Sie, auf diese Weise auch die Länge der Ausgabe von `gcc -E` zu bestimmen.

2. Betrachten Sie das folgende C-Programm:

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

Schicken Sie dieses Programm durch den C-Präprozessor. Geben Sie nun zusätzlich den Kommandozeilenparameter `-DFOO=0` an. Wie verändert sich die Ausgabe?

3. Nachdem Sie die obigen Experimente abgeschlossen haben, verwenden Sie den Präprozessor, um ein Programm zu schreiben, das zur Übersetzungszeit mit den folgenden Präprozessormakros konfiguriert werden kann:

   - `NAME`: Ein Name. Das Programm soll entweder „`Hallo, NAME`“ oder „`Auf Wiedersehen, NAME`“ ausgeben, wobei `NAME` durch den entsprechenden Namen ersetzt werden soll. Beachten Sie: Wenn Sie auf der Kommandozeile doppelte Anführungszeichen verwenden (z.B. `-DNAME="NAME"`), dann werden diese von der Shell selbst interpretiert. Um dies zu verhindern, müssen Sie das Fluchtsymbol „\“ verwenden (also z.B. `-DNAME=\"NAME\"`).
   - `HALLO`: Falls gesetzt (`-DHALLO`), wird die „`Hallo`“-Variante verwendet, sonst wird die „`Auf Wiedersehen`“-Variante verwendet.

4 Literatur

- “The C Programming Language” (Brian W. Kernighan, Dennis M. Ritchie) (auch auf Deutsch in der Bibliothek)
- C99–Spezifikation (draft) `http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/C99.pdf`
- “The Linux Programming Interface: A Linux and UNIX System Programming Handbook” (Michael Kerrisk)
- „Linux– Unix- Systemprogrammierung“ (Helmut Herold)
- Foliensätze 05-07 der Veranstaltung B-SYSP vom Sommersemester 2013 `http://sepl.cs.uni-frankfurt.de/2013-ss/b-sysp/veranstaltungen.de.html`