Systemnahe Programmierung

Was ist systemnahe Programmierung?

* Mit System ist das Betriebssystem, in unserem Fall Linux, gemeint
* Das Betriebssystem verwaltet die Hardware-Ressourcen und stellt Abstraktionen dafür bereit (Prozess, Dateisystem, ...)
* Systemnahe Software ist Software, die direkt auf die Schnittstellen des Betriebssystems zugreift.
* Systemnahe Software
  + Dienstprogramme (Editoren, Konverter, Statistiken...)
  + Werkzeuge
  + Services (Webserver, Datenbanken, ...)
  + Middleware ...
* Systemnahe Programmierung -> Entwicklung systemnaher Software

Betriebssystem-Schnittstelle

* Das Betriebssystem stellt eine Schnittstelle für Anwendungsprogramme zur Verfügung
* POSIX Standard - „Portable Operating System Interface for UniX“. „Für Unix entwickelte standardisierte Programmierschnittstelle, welche die Schnittstelle zwischen Anwendungssoftware und Betriebssystem darstellt.“
* Bestandteile: Definition der zugrundeliegenden Basiskonzepte C Systemaufrufen und dazugehörige Headerdateien Shell und Hilfsprogramme (sh, awk, vi ...)
* In POSIX kompatiblen Systemen ist die POSIX-Schnittstelle immer Teil der C Standardbibliothek „libc“ -> daher ist C immer die erste Sprache für systemnahe Programmierung.
* Auch praktisch alle anderen Sprach-Implementierungen verwenden letztendlich die libc.

Embedded Systems 🡪 behandeln wir hier nicht

* Sehr oft ohne Betriebssystem
* Direkte Programmierung der Hardware
* Keine einheitliche Schnittstelle zur Anwendungssoftware -> jedes System ist anders ...

1. Das Betriebssystem Linux

Hauptaufgaben

* Abstraktion (Linux verwaltet die Hardware und macht sie auf einer abstrakten Eben nutzbar)
* Virtualisierung (nicht direkt auf der Hardware)
* Geräte verwalten
* Interprozesskommunikation

Abstraktion

* Das Betriebssystem abstrahiert die Hardware eines Computers und stellt High- Level Konstrukte für Anwendungsprogramme zur Verfügung.
* Beispiele:
  + Festplatte (Zylinder, Sektoren, Blöcke) -> Filesystem (komfortabler als Abfrage über Blockzahlen etc.)
  + Geräte -> Zugriff darauf in Form von Dateien (/dev)
  + Druckerqueue
  + Benutzer und Zugriffsrechte

Prozesse - CPU Virtualisierung

Prozess - ist die Ablaufumgebung eines Programms auf einem Prozessor.

Das Betriebssystem ordnet jedem Prozess einen virtuellen Prozessor zu.

Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

Automatisch generierte Beschreibung

Der Betriebssystemkern (Kernel) läuft im privilegierten Modus der CPU und bildet den geschützten „Kernelspace“.

Ein Anwendungsprogramm (Prozess) läuft im User-Modus der CPU und hat keinen direkten Zugriff auf die Hardware. -> „Userspace“

Die Prozesse im Userspace sind voneinander komplett isoliert. Der Userspace kann auf die Geräte/CPU/Speicher ... nur über das Betriebssystem zugreifen

Prozesse - Scheduling

Da im Normalfall mehr Prozesse als reale Prozessoren vorhanden sind, ordnet das Betriebssystem virtuelle Prozessoren den realen Prozessoren für eine bestimmte Dauer (Timesclice) zu.

Nach dem Ablauf einer Timeslice:

* der Kernel übernimmt die Kontrolle (Kontext-Switch)
* der aktuell ausgeführte Prozess in die Warteschlange gestellt
* der nächste ausführbare Prozess wird bestimmt (Scheduling Algorithmus)
* Ein Bild, das Text, Screenshot, Reihe, Diagramm enthält.

  Automatisch generierte Beschreibungder nächste ausführbare Prozess wird aktiviert (Kontext-Switch)

Prozesshierarchie

* Regel in Linux: ein Prozess kann nur durch einen anderen Prozess erzeugt werden
* Beim Systemstart (boot) wird immer das Programm „init“ als erster Prozess gestartet.
* „init“ startet alle Programme deren Start für den Systemstart festgelegt ist. (Parent Prozess aller Prozesse)
* Diese Prozesse erzeugen weitere Prozesse - daraus ergibt sich eine Prozesshierarchie.

Ein Bild, das Text, Screenshot, Diagramm, Schrift enthält.

Automatisch generierte Beschreibung

PID ... Process Identifier - eindeutige Nummer die jedem Prozess zugeordnet ist.

PPID ... PID des erzeugenden Prozesses (Parent-PID)

Erste Übung

Öffnen sie ein Terminal und finden sie die PID des Shell-Prozesses, der gerade geöffnet ist heraus.

Verwenden sie den ps Befehl um folgende Fragen zu dem Prozess mit der gefundenen PID zu beantworten:

* Welches Programm wird ausgeführt?
* Welche PPID hat dieser Prozess?
* Welcher Prozess verbraucht aktuell am meisten CPU?
* Welcher Prozess hat seit dem Start am meisten CPU verbraucht?
* Welches Programm hat immer die PID 1?

Hinweise:

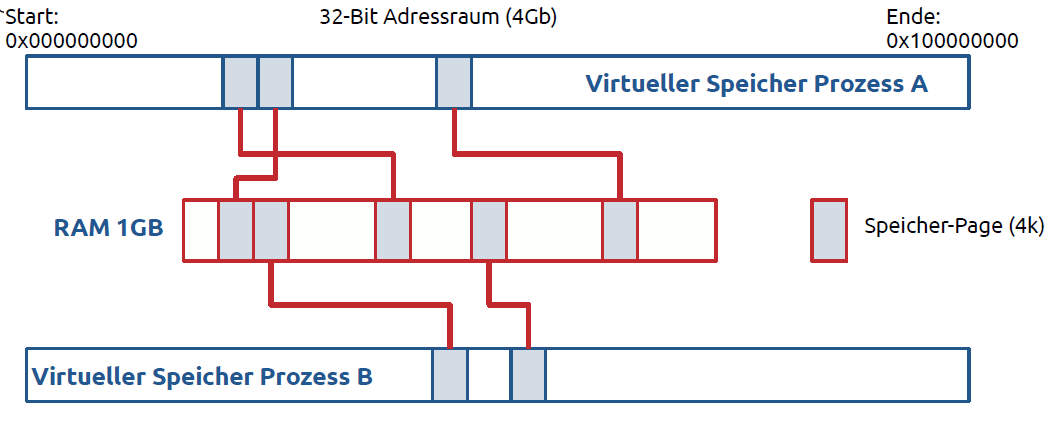
Der Befehl man zeigt die Manual-Seite eines Befehls an z.B.: man ps. Die Ausgabefelder von ps können sie mit den „Standard Format Specifiers“ anpassen. Die Shell-Variable $$ enthält die aktuelle PID.

ps -ply

Ps aux

Virtueller Adressraum

* Das Betriebssystem stellt jedem Anwendungsprozess einen eigenen virtuellen Adressraum zur Verfügung.
* Der reale Speicher wird vom Betriebssystem mit Hilfe einer MMU (Memory Management Unit) und einer Paging-Table in den virtuellen Adressraum gemappt.
* Der Speicher (virtuell und real) wird in Pages unterteilt.



Paging/Swapping

* Paging
  + Umsetzung der virtuellen Speicheradresse in ein physische Adresse.
    - Umrechnung Adresse -> Page
    - Nachschlagen der phys. Page in Pagetable
    - Berechnen der phys. Adresse
* Swapping
  + Wenn nicht genügend physischer Speicher vorhanden ist, können einzelne Pages oder alle belegten Pages eines Prozesses auf ein Swap-Device ausgelagert werden -> Swap Daemon kswapd
  + Bei Bedarf (pagefault) müssen diese wieder in den Speicher geladen werden.

Ein Bild, das Text, Katze, Diagramm, Säugetier enthält.

Automatisch generierte Beschreibung

Geräteverwaltung

* Korrekte Initialisierung der Geräte beim Start
* Geräte werden als Files in den Userspace gemapped. (für jedes Gerät wird eine Datei angelegt 🡪 kann direkt angesprochen werden)
  + z.B.: Festplatte /dev/sda1
* Schutz der Geräte vor fehlerhaften Anwendungsprogrammen
  + Nur kontrollierter high-level Zugriff aus dem Userspace (kein direkter Zugriff auf z.B.: Statusregister ...)
* Serialisierung von parallelen Zugriffen:
  + Prozesse die auf Geräte zugreifen wollen, müssen warten bis andere Zugriffe abgeschlossen sind -> Warteschlange
* Interprozesskommunikation
* Signale
* Pipes
* Message Queue
* Shared memory

Die Schnittstelle zum Betriebssystem

Die C Standardbibliothek libc stellt u.a. ein „API“ in Form von compilierten CFunktionen zur Verfügung und mappt diese auf das Kernel-Interface

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

Jeder Prozess stellt der Anwendung einen komplett separierten, virtuellen Adressraum zur Verfügung. Der Zugriff auf andere Prozesse ist nicht möglich, Kernel-Zugriff ist nur über syscall() möglich.

1. Programmiersprache C

Wird heute vor allem benutzt für:

* Entwicklung von Betriebssystemen
* Systemprogrammierung
* Embedded Systems
* Performance-kritische Anwendungen und Bibliotheken

Merkmale

* Wenige Schlüsselwörter - einfache Syntax
* Strukturen
* Zeiger
* Externe Standardbibliothek - libc
* Compiliert zu Maschinencode
* Makro-Präprozessor
* Moderne Derivate: C++, C# Objective C

Low-Level Sprache

* Explizite Speicherallokierung und Freigabe
* Keine Exceptions
* Keine objektorientierten Sprachkonstrukte
* Kein Range-checking
* Eingeschränkte Typ-Prüfung zur Compilezeit
* Keine Typ-Prüfung zur Laufzeit
* Kein Garbage Collector

Die C-Standardbibliothek

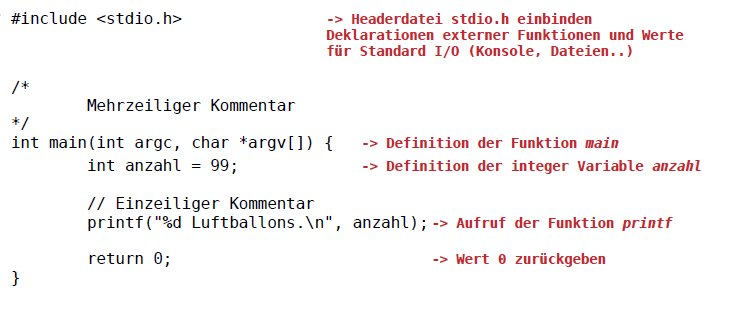
Die C-Standardbibliothek enthält ca. 200 Funktionen aus den Bereichen

* POSIX Systemschnittstelle (Prozesse, Speicherverwaltung ...)
* Ein-/Ausgabe Funktionen
* Stringfunktionen
* Mathematische Funktionen
* Erweiterte Typfunktionen
* Funktionen für Datum und Uhrzeit
* Sprach- und Gebietsfunktionen (locale)
* Konvertierungsfunktionen
* usw.

Die C-Standardbibliothek ist standardisiert und daher auf allen Systemen gleich, d.h. die Portierung von Programmen ist vergleichsweise einfach, abernatürlich nicht so einfach wie mit Java :-(

Die C-Standardbibliothek ist im Vergleich zu modernen Umgebung, wie der Java Klassenbibliothek sehr minimalistisch.

Struktur eines C-Programms



Kompilieren eines C-Programms

= Übersetzen in eine Maschinenlesbare Form

Ein Bild, das Text, Screenshot, Schrift, Diagramm enthält.

Automatisch generierte Beschreibung

Linker: nimmt object Datei nd verlinkt diese mit Bibliotheken 🡪 es entsteht eine ausführbare Datei (a.out)

Der C-Compiler ist standardmäßig als cc Befehl verfügbar:

cc ex1.c

Der Compiler ruft auch den Linker auf und erzeugt so gleich eine ausführbare Datei mit dem Default-Namen a.out.

Mit dem Flag -o wird der Name der ausführbaren Datei angegeben:

cc ex1.c -o luftballon

Nützliche Compiler-Flags

* -Wall ...alle Warnings anzeigen
* -O0 ... alle Optimierungen deaktivieren
* -g ... Debuginformationen erzeugen
* -l ... zusätzliche Bibliothek einbinden

Default Einstellungen des Compilers anzeigen:

cc -v -x c -E /dev/null

Header Dateien

stdio.h ist ein Teil der C Standard Bibliothek

weitere wichtige Headerdateien:

ctype.h, math.h, stdlib.h, string.h, time.h

Die Header-Dateien müssen sich im Include-Pfad befinden. Dieser zeigt normalerweise auf /usr/include und Unterverzeichnisse.

Mit dem Compiler-Flag -I<Pfad> können weitere Include-Verzeichnisse angegeben werden.

Dokumentation der GNU C Library: https://www.gnu.org/software/libc/manual/html\_node/index.html

Übersichtliche Funktionsreferenz: https://www.tutorialspoint.com/c\_standard\_library/index.htm

Allgemeines

Programmierumgebung und Tools

Eine virtuelle Maschine (Oracle VirtualBox)

Wir verwenden folgende Tools:

* gcc - GNU C-Compiler
* make
* Texteditor oder IDE (nach Vorliebe)

Tools installieren (ubuntu Linux)

Compiler und Tools installieren

sudo apt install build-essential

Splint – statische Checks

sudo apt install splint

Valgrind – Laufzeit Prüfung

sudo apt install valgrind

man Pages

Manpages installieren

sudo apt-get install manpages-dev

Wichtige Abschnitte:

* SYNOPSIS
  + Beschreibt die Funktionssignatur und notwendige Include-Dateien.
  + Beispiel:

#include <unistd.h>

ssize\_t write(int fd, const void \*buf, size\_t count);

* DESCRIPTION
* RETURN VALUE – wichtig, was wird zurückgegeben (Datentyp!)
* ERRORS – Wie wird errno im Fehlerfall gesetzt.
* SEE ALSO – Referenz zu ähnlichen Funktionen

Variablen und Datentypen

Variablen

Variable müssen vor der Nutzung deklariert werden

* int a;
* int a,b,c; -> Mehrfachdeklaration
* Initialisierung
* int a = 2;
* int a, b,c = 1, d=3;

Regeln für Variablennamen

Gültige Zeichen: Buchstaben, Zahlen

* Gültige Zeichen: Buchstaben, Zahlen, ‘\_‘
  + Dürfen nicht mit einer Zahl beginnen
  + Keine Schlüsselwörter
  + Sind case-sensitiv
* Gültig:
  + int \_Zahl
  + float s\_Weight3
* Ungültig:
  + char 1\_variable -> Zahl darf nicht erstes Zeichen sein

Speicherklassen auto und register

Variablen, die ohne zusätzliche Speicherklasse definiert werden, haben die Speicherklasse auto.

auto int i; = int i;

Die Speicherklasse register dient dazu, dem Compiler mitzuteilen, dass die Variable wenn möglich in einem Register gehalten werden soll (z.B.: Indexvaraible)

register int i;

Die Speicherklasse register sollte im Normalfall nicht verwendet werden, da mittlerweile die Compiler so gut optimierten Code erzeugen, dass eine manuelle Registerzuordnung die Optimierung eher schwächt.

Größe 🡪 char immer ein Byte, egal auf welchem System; andere können variieren!

Ein Bild, das Text, Screenshot, Zahl, Schrift enthält.

Automatisch generierte Beschreibung

Kein boolean Datentyp?

* Alle C-Ausdrücke expr evaluieren zu
  + false ... wenn expr == 0
  + true ... wenn expr <> 0
  + char c=0;
  + if (c) ... false
  + int i=-99;
  + if (i) ... true
* Seit C99 ist ein Boolean Datentyp mit <stdbool.h> verfügbar
  + Datentyp \_Bool
  + oder Macro bool
  + bool toBeOrNotToBe = true;

Literale

Numerische Literale

* Untertyp angeben
  + u ... unsigned unsigned int a = 30u;
  + l ... long long a = 30l;
  + unsigned long a = 30ul;
* Zahlensystem
  + dezimal 30
  + oktal 036
  + hexadezimal 0x1E
* Fliesskommazahlen
  + f ... float float f = 0.7f;
  + l ... long double long double ld = 0.7l;
* Entweder mit Komma oder Exponentialschreibweise
  + float f = 0.007;
  + float f = 7E-3;

Zeichen und Strings

* Zeichen (Typ char)
  + ‘A‘
  + ‘\x41‘ ... Hexadezimalcode
  + ‘\0101‘ ... Oktalcode
* Strings
  + “Hello World“ “Hello““World“
* Escape Sequenzen in Strings
  + \ ... Escape Character
  + \\ ... \
  + \‘ ... ‘
  + \“ ... “
  + \b ... Backspace
  + \t ... Tab
  + \r ... Carriage Return
  + \n ... Linefeed

Konstante & Enums

* Konstante deklarieren
  + const int = 1;
  + const char msg[] =“Hello world“;
* Enums
  + Enums bilden eine konstante Menge, deren Elementen ein Zahlenwert zugewiesen wird.
  + enum BOOL {no, yes } … no=0, yes=1
  + enum COLOR { red=1, blue, green, yellow=10 } … blue=2, green=3

Operatoren

Artithmetische Operatoren

+, -, \*, /, %

Relationale Operatoren

>, >=, <, <=, ==, !=

Logische Operatoren

&&, ||, !

Wenn der Wert eines Ausdrucks feststeht, dann wir der Rest nicht mehr ausgewertet!

Beispiel:

(1==1) || ((c=getchar())==‘y‘) -> der zweite Teil wird nicht ausgeführt

(0) && ((x=x+1)>0) -> der zweite Teil wird nicht ausgeführt

Sonstige Operatoren

sizeof(<expr>) -> berechnet den tatsächlichen Speicherverbrauch von expr

Increment/decrement Operatoren

++, --

Postfix

x++ y=x++ -> y=x; x=x+1; -> x wird zuerst zugewiesen, dann erhöht.

x-- y=x-- -> y=x; x=x-1;

Prefix

++x y=++x -> x=x+1;y=x; -> x wird zuerst erhöht, dann zugewiesen.

--y y=--x -> x=x-1;y=x;

Zuweisungs-Operatoren

Zuweisungsoperator: =

Shorthand Varianten

+=, -=, \*=, /=, %=

i += 8; -> i = i + 8;

Bitweise Operatoren

* & ... AND
* | ... OR
* ^ ... XOR
* ~ ... NOT (Einerkomplement)
* << ... shift left
* >> ... shift right

Beispiel

* 0x01<<4 = 0x10
* 0x10>>4 = 0x01
* Ein Leftshift um n Stellen entspricht einer Multiplikation mit 2n

Shorthand Varianten

* <<=, >>=, &=, |=, ^=

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte BeschreibungHäufige Bit Operationen

Bitoperationen werden häufig dazu verwendet einzelne Bits zu manipulieren.

* char c;
* Bit n setzen ( = 1 )
  + c = c | (1<<n)
* Bit n löschen ( = 0 )
  + c = c & ~(1<<n)
* Bit n invertieren
  + c = c ^ (1<<n)
* Abfragen, ob Bit n gesetzt ist
  + c & (1<<n)

Rangfolgen Operatoren

Ein Bild, das Text, Screenshot, Zahl, Schrift enthält.

Automatisch generierte Beschreibung

Einfache Ein/Ausgabe

stdin, stdout, stderr

* Die drei Streams stdin, stdout und stderr sind per default in jedem Programm definiert und zum Schreiben bzw. Lesen bereit.
* stdio.h bietet einige Funktionen, mit denen einfach auf diese Streams zugegriffen werden kann.
* EOF ... Rückgabewert wenn End-Of-File erreicht wurde (Stream geschlossen)
* int getchar(void);
  + Ein Zeichen von stdin lesen, Rückgabewert ist das gelesene Zeichen oder EOF (deswegen int). Abschluss m<Enter> Taste notwendig.
* int putchar(int ch);
  + Ein Zeichen auf stdout ausgeben, Rückgabewert ist das Zeichen das ausgegeben wurde oder EOF.

Scanf (Kommandozeileninput) und printf

* int scanf(const char \* restrict format,...);
  + Liest die Parameter formatiert von stdin ein. Rückgabewert ist die Anzahl der gelesenen Objekte.
    - int zahl1, zahl2;
    - int gelesen = scanf(“%d %d“,&zahl1, &zahl2); *🡪 & Macht einen Zeiger aus int*
* int printf(const char \* restrict format,...);
  + Gibt die Parameter formatiert auf stdout aus
    - char c; int d;
    - printf(“char: %c int: %d\n“,c,d);

printf Formate

* Verwendung
  + Der Formatstring von printf kann folgende Elemente enthalten:
    - Formatausdrücke
    - Text
    - Escape Sequenzen
  + printf (“Spalte 1 %d\tSpalte 2 %.2f\t Spalte 3 %s\n“,wert1,wert2,wert3);
  + Bei scanf dürfen nur Formatausdrücke verwendet werden. Soll vor einer Eingabe ein Text stehen, so muss dieser vorher mit printf oder einer anderen Funktion ausgegeben werden.
  + Beispiel:
    - printf (“Bitte geben Sie 2 Zahlen ein:“);
    - anzahl = scanf(“%d %d“, &zahl1, &zahl2);

Aufbau: %[flags][width][.precision][modifier]<type>

type:

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

Aufbau: %[flags][width][.precision][modifier]<type>

flags:-

* -... linksbündig ausrichten (default rechtsbündig)
* + ... Vorzeichen immer ausgeben
* # ... Oktal/Hexadezimalzahlen mit 0, 0x ausgeben
* 0, Leerzeichen ... Links mit 0 oder Leerzeichen auffüllen

width: Mindestanzahl an Zeichen die ausgegeben werden (kein Abschneiden)

Beispiele:

* printf(“%3d“,2); -> \_\_2 ( \_\_ = 2 Leerzeichen)
* printf(“%7s“,“hello“); -> \_\_hello 🡪 String der Länge 7 wird vorne aufgefüllt
* printf(“%-7s“,“hello“); -> hello\_\_
* printf(“%+3d“,21); -> +21 🡪 Vorzeichen mit anführen
* printf(“%#8x“,21); -> 0x15 🡪 Umwandeln in Hexadezimal
* printf(“%04d“,21); -> 0021

Aufbau: %[flags][width][.precision][modifier]<type>

precision:

* Anzahl der Zeichen für %d, %i, %o, %u und %x
* Anzahl der Nachkommastellen für %a, %e, %f
* printf(“%.6d“,314); -> \_\_\_314
* printf(“%.2f“,3.14159); -> 3.14
* printf(“%05.2f“,3.14159); -> 03.14 🡪 Führende 0 und auf 2 beschränken)

modifier:

* Datensubtyp anpassen
* h ... als short interpretieren (mit i,d,u,x)
* l .. als long interpretieren (mit i,d,u,x)
* L .. als double interpretieren (mit e,f)
* printf(“%Lf“,314159.0l); -> 314159.000000 🡪 Float wird als Double interpretiert

Typumwandlung

Automatische Typkonvertierungen

* char wird bei Ausdrücken immer in int umgewandelt.
  + char c; (c++) -> int
* float wird bei Ausdrücken immer in double umgewandelt.
* In Ausdrücken mit gleichem Grundtyp wird das Ergebnis immer in den größten vorkommenden Datentyp konvertiert.
* Ganzzahlige Typen:
  + char < short < int < long < long long
  + Beispiel: int \* long -> long
* Gleitkomma-Typen:
  + float < double < long double
  + Beispiel: double / long double -> long double
* Die Konvertierung von längeren Datentypen in kürzere kann zu Datenverlust führen.

Gemischte Ausdrücke

Bei der Auswertung von Ausdrücken wird solange wie möglich nach int konvertiert.

Sobald ein Gleitkommatyp im Ausdruck vorkommt, wird der gesamte Ausdruck in Gleitkomma umgewandelt.

Übung: Wie lautet das korrekte Ergebnis der Berechnung (c+i)\*f ?

int i=2;

char c=‘a‘;

float f = 3.14159;

printf (?);

Explizite Typkonvertierungen - Cast

Syntax: (Typ) <Ausdruck>

c = (char)65; // ’A’

double pi\_i = 3.14159;

float some\_num = (float) c \* pi\_i; // 3.0

Dies macht eig. Nur bei Zeigern Sinn, da durch die automatische Typkonvertierung sowieso verändert

Blöcke

Statement

i=0;

printf(“Out: %d\n“, i);

Compound Expressions

Durch Komma getrennte Ausdrücke:

i=0, j=0, puts(“Test“);

Block

Alles zwischen { und }

Beispiel

int main (int arvc, char \*argv) { ... }

Anweisungen, denen kein Anweisungsblock folgt, werden mit einem Semikolon abgeschlossen.

Conditionals

If-else

if (expr) -> expr == 0 -> false, sonst true!

Block

else if (expr)

Block

else

Block

Verschachtelung

if (expr)

if (expr)...

else - > zuerst wird immer das innerste if abgeschlossen

...

if (expr){

if (expr) ...

} else - > so gehört das else zum äusseren if

...

Ternärer Operator

expr ’?’ expr1 ’:’ expr2

Entspricht:

if (expr)

expr1;

else

expr2;

Beispiel:

x>100 ? “alt“ : “jung“;

switch

switch (expr) { -> expr muss zu einer ganzen Zahl auswertbar sein! (int oder char)

case 1 :

/\* block or statement \*/

break;

case 2 :

/\* block or statement \*/

break;

default:

/\* block or statement \*/

break;

}

Switch ist wie eine „Sprungtabelle“

Der Code wird vom ersten „case“ Match ausgeführt bis ein „break“ auftritt.

switch (num) {

case 1 :

case 2 :

/\* do something if num == 1 or 2 \*/

break;

case 3 :

/\* do something else if num == 3 \*/

break;

case 4 :

/\* if num == 4: do this ... \*/

/\* da hier kein break wird bei num = 4 sowohl case 4 als auch case 5 ausgeführt\*/

case 5 :

/\* and do that if num == 4 or 5 \*/

break;

default:

/\* block or statement \*/

break;

}

Schleifen

while

Der Body wird solange ausgeführt, solange die Auswertung von (expr) true ergibt.

while (expr)

/\* body \*/

Wenn expr beim ersten Aufruf false liefert, wird der Body nie durchlaufen.

Beispiel: while ( i< 10 ) i++; oder kürzer while(++i<10);

do while

Der Body wird solange ausgeführt, solange die Auswertung von (expr) true ergibt.

do /\* body \*/

while (expr);

Der Body wird mindestens einmal durchlaufen.

Beispiel: do i++; while ( i< 10 );

for

for (initialization; condition; increment)

/\* body \*/

Beispiel:

for (i=0; i<1000; i++)

printf(“%d“,i);

Die einzelnen Ausdrücke können auch leer sein, „increment“ wird dann als true angenommen.

break

Mit break kann jede Schleife vorzeitig verlassen werden.

while ( i < 15 ){

if ( i%2 == 0 )

break;

/\* do something ...\*/

}

continue

Mit continue wird der Rest der Schleife übersprungen und mit der Loop-Condition fortgesetzt.

for (i=0, j=1000; i<1000; i++, j--) {

if (j == i) // skip if i==j

continue;

printf(“%d“,i);

}

Funktionen

Definition

Syntax:

<returntype> function\_name(args) { statements; }

Beispiel:

float flaecheEllipse(float a, float b) {

float pi=3.14159; 🡪 Lokale Variabe

return a\*b\*pi;

}

Returntype = void -> Es wird kein Wert zurückgegeben.

Wenn kein Returntype angegeben wird, dann wird int angenommen.

Ein Funktion muss vor ihrer Verwendung deklariert oder definiert werden.

Parameter

Die Parameterübergabe findet immer By-Value statt

Unterschied Definition - Deklaration

* Deklaration
  + Die Deklaration beschreibt den Namen, Rückgabewert und Parameter einer Funktion = Signatur der Funktion.
  + Sie wird benötigt, wenn die Funktion in anderen Compilation-Units (.c Dateien) verwendet wird oder wenn sie vor ihrer Definition verwendet wird.

int add(int a, int b);

* + Die Deklaration sollte immer in einer Headerdatei (.h) angegeben werden.
* Definition
  + Die Definition legt das Verhalten der Funktion fest.

int add(int a, int b) {

return a+b;

}

C-Standard Bibliotheken

Mathematische Funktionen

#include <math.h>

double sqrt(double x);

double pow(double x, double y)

double exp(double x);

double log(double x);

double sin(double x);

double cos(double x);

...

Die Mathematikfunktionen befinden sich in einer eigenen Bibliothek. Diese muss zum Executable dazu gelinkt werden.

In diesem Fall kann mit dem Flag

-l<Bibliotheksname>

die gewünschte Bibliothek angegeben werden. Die Mathematikbibliothek hat den Namen m. ( Finden mit: locate libm.a)

Also:

cc ex1.c -o luftballon -lm

Funktionen für Zeichen

#include <ctype.h>

int isalnum(int c) testen auf alphanumerisches Zeichen (a-z, A-Z, 0-9)

int isalpha(int c) testen auf Buchstabe (a-z, A-Z)

int iscntrl(int c) testen auf Steuerzeichen (\f, \n, \t ...)

int isdigit(int c) testen auf Dezimalziffer (0-9)

int isgraph(int c) testen auf druckbare Zeichen ohne Leerzeichen

int islower(int c) testen auf Kleinbuchstaben (a-z)

int isprint(int c) testen auf druckbare Zeichen mit Leerzeichen

int ispunct(int c) testen auf druckbare Interpunktionszeichen

int isspace(int c) testen auf Whitespace

int isupper(int c) testen auf Großbuchstaben (A-Z)

int isxdigit(int c) testen auf hexadezimale Ziffern (0-9, a-f, A-F)

Returnwert <> 0 : Bedingung erfüllt

int tolower(int c) wandelt Groß- in Kleinbuchstaben um

int toupper(int c) wandelt Klein- in Großbuchstaben um

Scope

Scope von Variablen

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

Was lokaler ist ist quasi sichtbarer

Hier ist x int und nicht float von globaler

Im Gelben wird z.B. lokales y genommen

Speicherklasse static

Die Speicherklasse static hat im Kontext einer globalen und einer lokalen Variable eine unterschiedliche Bedeutung.

Lokale static Variable

Eine lokal zu einer Funktion mit „static“ definierte Variable bleibt für die gesamte Ausführung des Programms erhalten.

int add(int a) {

static int x=0; *-> der Wert von x bleibt über jeden Aufruf von add() erhalten.*

x += a;

return x;

}

Beispiel:

* add(3); -> 3
* add(5); -> 8
* add(87); -> 95

Static wird einmal initialisiert und dann „übersprungen“

static - globale Variable und Funktionen

Ist eine globale Variable oder eine Funktion mit „static“ deklariert, dann ist ihre Sichtbarkeit auf die beinhaltende Datei beschränkt.

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

Arrays und Zeichenketten

Arrays

Arrays in C bestehen aus aufeinander folgenden Elementen gleichen Typs im Speicher. Sie haben eine feste Dimension, die bei der Deklaration festgelegt wird.

Ein Bild, das Schrift, Text, Zahl, Reihe enthält.

Automatisch generierte Beschreibungint arr[20];

float matrix[4][4]; -> Mehrdimensional Zeilen/Spalten

Der Arrayindex beginnt bei 0 :

Initialisierung

int arr[] = {1,2,3,4,5}; → Compiler setzt die Länge automatisch

float matrix[3][4] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};

Verwendung

a = arr[2];

matrix[0][i+1] = elem;

Verwendung

Die Elemente eines Arrays werden mit Ihrem Index angesprochen.

Beispiel: Initialisieren eines Arrays mit 0

int arr[10];

for (int i=0; i<10; i++)

arr[i] = 0;

Das Array selbst ist eine Konstante, Zuweisung funktioniert nicht:

int arr1[2]={1,2};

int arr2[2]={3,4};

arr2 = arr1; → Geht nicht!

Größe eines Arrays

Für Arrays gibt es in C keine len() Methode oder ähnliches.

Um die Anzahl der Elemente in einem Array zu berechen wird der sizeof() Operator verwendet.

int arr[20];

Anzahl der Elemente = sizeof(arr) ?

→ nein, berechnet den belegten Speicherplatz des Arrays

sizeof(arr) = Anzahl der Elemente \* sizeof(int)

→ Anzahl der Elemente = sizeof(arr)/sizeof(int)

oder sizeof(arr)/sizeof(arr[0])

Achtung: Anzahl Elemente bedeutet Anzahl der reservierten Elemente.

Verwendung 2

Arrays können als Parameter an eine Funktion übergeben werden:

void funktion(int arr[]){ ... }

ABER: ein Array als Funktionsparameter wird in einen Zeiger umgewandelt und verliert seine Längeninformation und sizeof(arr) liefert dann nicht mehr den belegten Speicherplatz des Arrays!

Also: Die Länge als Parameter mit übergeben:

void funktion(int arr[], int laenge){ ... }

Arrays können nicht als Rückgabetyp von Funktionen verwendet werden.

int arr[] funktion(){ ... } -> geht nicht!

Arrays mit variable Länge

Seit C99 können Arrays mit variabler Größe angelegt werden.

int y=10;

int arr[y];

Es gelten jedoch folgende Einschränkungen:

* Die Größe kann nur einmal bei der Definition des Arrays festgelegt werden.
* Arrays mit variabler Länge können nur innerhalb von Funktionen angelegt werden.

Arraygrenzen

* Die Grenzen von Arrays werden zur Laufzeit nicht überprüft!

char arr[10]=“Luftballon“;

char arr1[5]=“Hallo“;

char c = arr[12];

char c = arr[-3];

* Beide Statements werden vom Compiler akzeptiert und liefern zur Laufzeit völlig undefinierte Werte zurück.
* Wenn der Index auf eine ungültige Speicheradresse verweist, dann wird das Programm abgebrochen (Segmentation fault).
* Die ungeprüften Arraygrenzen in C werden oft für sog. buffer overflow attacks ausgenützt.

Zeichenketten/Strings

Strings sind Arrays vom Datentyp char, deren Ende mit \0 markiert wird (ASCII Code für die Zahl0).

char text[24]=“FH JOANNEUM“;

Ein Bild, das Text, Reihe, Schrift, Screenshot enthält.

Automatisch generierte Beschreibung

char text[24];

text =“FH JOANNEUM“; → das geht nicht, weil Arrays ja konstant sind.

Strings können ansonsten wie Arrays verwendet werden.

z.B.: text[10]=‘m‘ oder if (text[j] == ‘H‘)

Wichtige Stringfunktionen

#include <string.h>

strlen(s) -> Länge von s, der Terminator wird nicht mitgerechnet!

strcpy(s1,s2) -> kopiert s1 in den String s2

strcat(s1,s2) -> fügt s2 am Ende von s1 an

strcmp(s1,s2) -> vergleicht s1 mit s2, liefert:

0 ... s1 == s2

<0 ... s1 < s2

>0 ... s1 > s2

Umwandlungsfunktionen

#include<stdlib.h>

atoi(string) ... string -> int

atof(string) ... string -> float

atol(string) ... string -> long

Beispiele

int i = atoi(“334“);

float f = atof(“334.14159“);

Zeiger

Zeiger

Zeiger sind Variable, die virtuelle Adressen enthalten, also auf eine andere Variable/Speicherzelle zeigen.

oder Allgemein: Ein Zeiger ist ein Objekt, das die Speicheradresse eines anderen Objektes enthält.

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte BeschreibungDeklaration: <datentyp> \*variable;

int \*p1;

char \* p2;

p1 ist ein Zeiger auf einen Speicherplatz vom Typ int

Operatoren

Der Adressoperator & liefert die virtuelle Adresse einer Variable

Der Dereferenzierungsoperator \* ermöglicht den Zugriff auf den Inhalt einer Zeigervariable (auch Inhaltsoperator)

Ein Bild, das Text, Schrift, Screenshot, Reihe enthält.

Automatisch generierte Beschreibung

int i=4711;

int \*p;

Ein Bild, das Text, Schrift, Screenshot, Reihe enthält.

Automatisch generierte Beschreibung

p = &i;

p = Adresse von i

Ein Bild, das Text, Schrift, Screenshot, Reihe enthält.

Automatisch generierte Beschreibung

\*p =4712;

Setze den Inhalt des Speicherplatzes , auf den p zeigt, auf den Wert 4712.

Zeiger verwenden

Zeiger können wie „normale“ Variable verwendet werden

* Einen Wert zuweisen – aber nur ein Zeiger vom gleichen Typ
* Dereferenzieren um den Wert zu lesen
* Dereferenzieren um den Wert zu schreiben

Beispiel

int main() {

int i = 123;

int \*p;

p = &i; *// Adresse zuweisen*

\*p = 345; *// Dereferenzierung, Wert zuweisen*

printf("i=%d\n", i);

printf("\*p=%d\n", \*p); *// Dereferenzierung, Wert lesen*

}

Zeiger verwenden

Zeiger ermöglichen Funktionsaufrufe mit Call By Reference, indem die Adresse einer Variable als Parameter übergeben wird.

Beispiel

void inc(int \*zahl) {

(\*zahl)++;

}

int main() {

int i = 1;

inc(&i);

printf("zahl=%d\n", i);

}

Zeiger sind Typ-gebunden

int \*p;

int \*\*pp; // *Zeiger auf einen Zeiger der auf einen int Speicherplatz zeigt*

int i;

float f;

p=&i;

p=&f; *-> geht nicht, weil int \* != float \**

pp=&p;

pp=p; *-> geht nicht, weil int \*\* != int \**

Casting von Zeigern

char c='A';

char \*p;

int \*i;

p = &c;

i =(int \*)p; *-> Ok char \* – int \**

float f = (float \*)p; *-> geht, aber nicht sinnvoll!*

Typlose Zeiger

void Zeiger sind typlos und können von/zu jedem anderem Zeigertypen konvertiert werden. void Zeiger selbst können (ohne cast) nicht dereferenziert werden.

Deklaration: void \*p;

int main() {

int i=3;

void \*p;

int \*p1,\*p2;

p = &i;

Richtig:

printf("Inhalt von p: %d\n", \*(int \*)p);

p1 = p;

p2 = &i;

printf(“Inhalt von p: %d\n“,\*p);

}

Funktioniert das oben angeführte Programm? Warum nicht? Sind p,p1,p2 gleich?

Ungültige Verwendung von Zeigern

1. Einen nicht initialisierten Zeiger dereferenzieren

int \*p;

\*p = 4711; 🡪 *geht nicht, weil dieser Pointer auf nichts zeigt*

2. Zeiger zu Variablen die Out-of-Scope sind

int \*verweis(int i) {

int j = i;

return &j;

}

🡪Geht nicht, da Adresse von j nur in Funktion Verweis vorhanden (Funktion wurde da bereits verlassen und Speicher existiert nicht mehr)

int main() {

int \*p = verweis(100);

\*p = 4711;

}

3. Einem Zeiger einen ungültigen Wert zuweisen

int \*p;

p = 4711; 🡪 Ein Zeiger ist kein Integer den man zuweisen kann

Null-Zeiger

Wenn ein Zeiger auf kein Objekt verweist, kann man ihm den Wert NULL zuweisen.

NULL ist ein Integer Literal vom Wert 0 und ist als Makro definiert:

#define NULL 0L

#define NULL (void \*)0

Man könnte auch schreiben

int \*p = 0;

Es empfiehlt sich aber immer das Makro zu verwenden:

int \*p = NULL;

Viele Funktionen, deren Rückgabewert ein Zeiger ist, geben NULL zurück wenn ein Fehler aufgetreten ist und kennzeichnen so den ungültigen Rückgabewert.

const Zeiger

Wenn ein Zeiger als const deklariert wird, dann ist der Inhalt, auf den der Zeiger verweist schreibgeschützt, der Zeiger selbst nicht.

Beispiel:

void printString(const char \*s) {

while (\*s != '\0') {

putchar(\*s);

++s; // *-> Zeiger wird verändert*

}

}

Zeiger - Array - String

Die Elemente von Arrays und Strings werden sequentiell hintereinander im Speicher abgelegt. Daher können auch Zeiger dazu verwendet werden, um auf Arrays und Strings zuzugreifen.

Der Variablenname eines Arrays ist ein unveränderbarer Zeiger, der auf das erste Elements das Arrays zeigt.

int arr[8]={1,2,3,4,5,6,7,8};

int ∗p;

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibungp = arr ; <=> p = &arr [0];

arr = p → geht nicht!

Eine Array-Variable ist eine vom Compiler generierte, unveränderliche Beschreibung des den durch das Array belegten Speicherplatzes.

Wird ein Array durch einen Zeiger angesprochen ist z.B.: die Information über den belegten Speicherplatz nicht mehr verfügbar!

p = arr ;

sizeof(arr) = 8 \* sizeof(int)

sizeof(p) → nicht definiert!

Arrays als Parameter einer Funktionen werden implizit in einen Zeiger umgewandelt. Die Längeninformation geht verloren!

void funktion ( int arr[] ){

int len = sizeof(arr)/sizeof(int);

printf("Länge int Funktion: %d\n", len); *->nicht die Array Länge!*

}

Länge des Arrays als Parameter mitgeben:

void funktion ( int arr[], int arrLen ){

for ( i=0; i<arrLen;i++)

arr[i]+=1;

}

Jeder Zeiger kann wie ein Array verwendet werden:

char st[11]=“Hallo Welt!“;

char \*p;

p=st;

for(int i=0; i<11; i++)

printf(“Zeichen: %c\n“,p[i]);

Oder mit Zeigerarithmetik:

for(p=st; \*p!='\0'; p++)

printf("Zeichen: %c\n",\*p);

Achtung: Die Länge von Speicherbereichen auf die ein Zeiger zeigt, muss immer selbst verwaltet werden.

Zeigerarithmetik

Zeiger sind Adressen. Zeigerarithmetik erlaubt es mit Adressen zu rechnen:

Zeiger: char \*p;

* Zeiger auf nächstes Element setzen: ++p oder p++
* Zeiger auf vorheriges Element setzen: --p oder p--
* Zeiger n Elemente nach vor setzen: p=p+n oder p+=n
* Zeiger n Elemente nach zurück setzen: p=p-n oder p-=n

Beispiele:

c = \*p++ -> c=\*p; p=p+1

\*str + i != \*(str+i)

p ist int \* : p+2 -> p + (2 \* sizeof(int))

s ist char \* : s+2 -> s +2 weil sizeof(char) == 1

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte BeschreibungÄquivalente:

strlen() Funktion mit Zeigerarithmetik

int my\_strlen(char \*string) {

char \*s;

for (s = string; \*s; ++s);  *🡪 Solange nicht 0 geht Schleife weiter*

return(s - string); *🡪 s zeigt auf Null Terminator und Differenz zum Start ist die Länge*

}

Zeiger auf Zeiger und Arrays

char \*laender[] = {"\xd6sterreich", "Deutschland", "Schweiz",

"Tschechien", "Slowenien"};

char \*\*pp = laender;

printf("%c\n", \*\*laender); 🡪 Ö

printf("%c\n", \*laender[0]); 🡪 Ö (Erstes Element vom Ersten String)

printf("%c\n", \*laender[4]); 🡪 S (1. Element vom 5. Element)

printf("%c\n", \*(laender[0]+2)); 🡪 t

printf("%c\n", \*(laender[0]+3)); 🡪 e

printf("%s\n", laender[0]); 🡪 Österreich (String Ausgabe)

printf("%s\n", \*laender); 🡪 Österreich

printf("%s\n", (laender[0]+1)); 🡪 sterreich (Beginnt erst bei [1])

printf("%s\n", (laender[0]+2)); 🡪 terreich

printf("%c\n", \*\*pp); 🡪 Ö

printf("%c\n", \*(\*pp+2)); 🡪t

printf("%c\n", \*\*(pp+2)); 🡪 S

printf("%c\n", \*(\*(pp+2)+3)); 🡪w

printf("%s\n", \*(pp+2)+3); 🡪 weiz

printf("%s\n", \*pp+2); 🡪 terreich

Zeiger auf Funktionen

f ist ein Zeiger auf eine Funktion, die zwei int Werte als Parameter hat und int zurückliefert.

int (\*f)(int, int);

f ist ein Zeiger auf eine Funktion, die zwei int Werte als Parameter hat und int \* zurückliefert.

int \*((\*f)(int, int));

Zuweisung:

f = add; // add ist eine Funktion die vorher definiert wurde zB

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

Aufruf:

int res = f(2,3);

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

1. Übungen

Übung Quadratwurzel

Schreiben sie ein Programm, das die Quadratwurzel der Zahl 3638 berechnet und ausgibt.

Verwenden sie dazu die Bibliotheksfunktion sqrt().

Die Definition von sqrt() können sie mit man sqrt nachschlagen.

Verwenden sie als Ausgabeformat (in der Funktion printf): %.2lf

Die Ausgabe soll so aussehen: Die Wurzel von 3638 ist <Ergebnis>

Hinweis: #include Dateien nicht vergessen.

#include <stdio.h>

#include <math.h>

int main() {

int startnumber = 3638;

double squareroot = sqrt(startnumber);

printf("Die Wurzel von %d ist %.2lf\n", startnumber, squareroot);

return 0;

}

Übung 1 Kleinbuchstaben

Definieren sie eine char Variable und initialisieren sie diese mit dem Kleinbuchstaben 'q'.

Wandeln sie den Inhalt der Variable unter Benutzung arithmetischer Operationen in einen Großbuchstaben um.

Das Programm sollte folgende Ausgabe erzeugen:

Kleinbuchstabe: q Großbuchstabe: Q

Hinweis

Die Buchstaben A-Z und a-z sind im ASCII Code sequenziell abgelegt.

A-Z entspricht dezimal 65-90

a-z entspricht dezimal 97-122

#include <stdio.h>

int main(){

char kleinbuchstabe = 'q';

char grossbuchstabe;

grossbuchstabe = kleinbuchstabe-32;

printf("Kleinbuchstabe: %c Großbuchstabe: %c\n", kleinbuchstabe, grossbuchstabe);

return 0;

}

Übung 2 ASCII

Lesen sie mit der getchar() Funktion eine char Variable ein und geben sie deren ASCII-Wert als Dezimalzahl und als Hexadezimalzahl aus.

Beispiel:

Zeichen eingeben: w

Der ASCII Wert von w ist 119/0x77

Testen sie das Programm mit den Zeichen w, U, 1 und ?

#include <stdio.h>

int main(){

char c;

printf ("Bitte ein Zeichen eingeben: ");

c = getchar();

printf("Der ASCII Wert von %c ist %d/%#x", c,c,c);

return 0;

}

Übung 3 Schaltjahr

Schreiben sie ein Programm, das prüft, ob ein Jahr ein Schaltjahr ist. Das Jahr soll als int mit scanf() eingelesen werden.

Folgende Regeln gelten für Schaltjahre:

* Ist die Jahreszahl durch 400 teilbar, ist es ein Schaltjahr
* Ist die Jahreszahl durch 100 teilbar aber nicht durch 400, ist es ein Schaltjahr.
* Ist die Jahreszahl durch 4 teilbar aber nicht durch 100, dann ist es ein Schaltjahr.
* In allen anderen Fällen ist das Jahr kein Schaltjahr.

#include <stdio.h>

int main(){

int year;

printf("Bitte ein Jahr eingeben: ");

scanf("%d", &year);

if (year % 400 == 0 ){

printf("%d ist ein Schaltjahr.\n", year);

}else {

if (year % 100 == 0 ){

printf("%d ist ein Schaltjahr.\n", year);

} else {

if (year % 4 == 0){

printf("%d ist ein Schaltjahr.\n", year);

} else{

printf("%d ist kein Schaltjahr.\n", year);

}

}

}

return 0;

}

Übung 5 Fakultät

Schreiben sie ein Programm, das eine ganze Zahl einliest und deren Fakultät n! berechnet.

Ein Bild, das Schrift, Text, weiß, Diagramm enthält.

Automatisch generierte Beschreibung

Verwenden sie eine for Schleife und scanf() dazu.

#include <stdio.h>

int main(){

int zahl;

unsigned long fakultaet = 1;

printf("Bitte eine Zahl eingeben: ");

scanf("%d", &zahl);

for (int i=1; i<= zahl; i++){

fakultaet \*= i;

}

printf("%lu", fakultaet);

return 0;

}

Unsigned long 🡪 Da wenn Zahl zu groß wird, diese sonst nicht mehr angezeigt werden kann

Übung 6 Dreieck

Schreiben sie eine Funktion flaecheDreieck(...), die den Flächeninhalt eines Dreiecks nach der folgenden Formel von Heron berechnet und zurückgibt.

Ein Bild, das Schrift, Handschrift, Reihe, Diagramm enthält.

Automatisch generierte Beschreibung

Übergeben sie der Funktion die Werte von a,b und c als Variable vom Typ double.

Lesen sie die Werte von a,b,c mit scanf() ein.

#include <stdio.h>

#include <math.h>

int main(){

double a;

double b;

double c;

double area;

double s;

printf("Bitte Seitenlänge a eingeben: ");

scanf("%lf", &a);

printf("Bitte Seitenlänge b eingeben: ");

scanf("%lf", &b);

printf("Bitte Seitenlänge c eingeben: ");

scanf("%lf", &c);

s = (a+b+c)/2;

area = sqrt(s \* (s - a) \* (s - b) \* (s - c));

printf("Area: %lf", area);

return 0;

}

Übung 01 Arrays

* Berechnen sie die Anzahl der Elemente in einem Array und geben sie jedes Element innerhalb einer Schleife einzeln aus.
* Berechnen sie weiters auch den Minimal- und Maximalwert des Arrays.

Verwenden sie dafür folgendes float Array: float weights[5]={1.34,4.567,56.55,33.44,782.00};

Damit die Validierung korrekt erfolgen kann, bitte folgendes Output Format verwenden:

Anzahl: 5

1.34

4.57

...

Min: 1.34

Max: 782.00

#include <stdio.h>

int main(){

float weights[5]={1.34,4.567,56.55,33.44,782.00};

int arraySize = sizeof(weights)/sizeof(float);

printf("Anzahl: %d\n",arraySize);

for (int i=0; i<arraySize; i++){

printf("%.2f\n", weights[i]);

}

float min = weights[0];

for (int i = 1; i < arraySize; i++) {

if (weights[i] < min) {

min = weights[i];

}

}

printf("Min: %.2f \n", min);

float max = weights[0];

for (int i = 1; i < arraySize; i++) {

if (weights[i] > max) {

max = weights[i];

}

}

printf("Max: %.2f", max);

return 0;

}

Übung 02 Länge eines Strings:

Schreiben sie eine Funktion my\_strlen(char string[]), welche die Länge eines Strings ohne den sizeof() Operator zu verwenden, berechnet.

Verwenden sie dazu eine Schleife und testen sie die Funktion mit folgender Zeichenkette:

char text[100]="FH JOANNEUM";

Geben sie den String und die Länge so aus:

String: “FH JOANNEUM“ Länge:11

#include <stdio.h>

int my\_strlen(char string[]){

int length = 0;

while (string[length] != '\0') {

length++;

}

return length;

}

int main(){

char text[100]="FH JOANNEUM";

int strlen = my\_strlen(text);

printf("String: \"%s\" Länge: %d", text, strlen);

}

Übung 03 Swap

Schreiben sie eine Funktion swap, die zwei int Werte vertauscht.

Die Signatur sollte so aussehen: void swap(int \*a, int \*b);

Testen sie die Funktion für folgende Wertepaare und geben sie die Werte vor und nach der Vertauschung aus.

Wertepaare:

1 8

98172232 77

34 31092

Ausgabe:

a:1 b:8

a:8 b:1

...

#include <stdio.h>

void swap(int \*a, int \*b){

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int main(){

int wertePaare[][2] = {

{1, 8},

{98172232, 77},

{34, 31092}

};

int sizeWertePaare = sizeof(wertePaare)/sizeof(wertePaare[0]);

//printf("Größe von wertePaare: %d\n", sizeWertePaare);

for(int i=0;i<sizeWertePaare; i++){

int a = wertePaare[i][0];

int b = wertePaare[i][1];

printf("a:%d b:%d\n", a, b);

swap(&a, &b);

printf("a:%d b:%d\n", a, b);

}

return 0;

}

Übung 4 Quicksort

Die Funktion qsort() ist in der Standardlibrary vorhanden, um Arrays oder Speicherbereiche zu sortieren.

Schlagen sie die Definition von qsort() in der entsprechenden man-Page nach.

Schreiben sie ein Programm, das ein int-Array mit 10 Werten sortiert.

Verwenden sie dieses Array:

int arr [] ={10,286,888,104,22,399,58,38,1,478,6,99};

Verwenden sie die Funktion qsort() um das Array zuerst aufsteigend und dann absteigend zu sortieren. Definieren sie dazu zwei Vergleichsfunktionen asc() und desc() die dem Funktionsprototypen oben entsprechen und jeweils eine aufsteigende oder absteigende Sortierreihenfolge implementieren.

Geben sie die sortierten Arrays auf stdout aus.

Ausgabeformat:

Aufsteigend:

...

Absteigend:

...

#include <stdio.h>

int asc(const void \*a, const void \*b) {

return (\*(int\*)a - \*(int\*)b);

}

int desc(const void \*a, const void \*b) {

return (\*(int\*)b - \*(int\*)a);

}

int main(){

int arr [] ={10,286,888,104,22,399,58,38,1,478,6,99};

int arrLength = sizeof(arr) / sizeof(arr[0]);

printf("Vor dem Sortieren: \n");

for (int i = 0; i < arrLength; i++) {

printf("%d ", arr[i]);

}

qsort(arr, arrLength, sizeof(int),asc);

printf("\nAufsteigend: \n");

for (int i = 0; i < arrLength; i++) {

printf("%d ", arr[i]);

}

qsort(arr, arrLength, sizeof(int),desc);

printf("\nAbsteigend: \n");

for (int i = 0; i < arrLength; i++) {

printf("%d ", arr[i]);

}

}