Propädeutikum
Aufgaben zum C-Praktikum

Ausdruecke.dir

1. hallo.c (Einstieg)
Legen Sie ein neues Verzeichnis ***Ausdruecke.dir*** an und wechseln Sie dorthin. Schreiben Sie das Programm „Hallo, Welt!“ aus der Vorlesung und testen Sie dieses aus.

2. ausdruecke_[a | b | c | d].c (C-Ausdrücke)
Schreiben Sie je ein Programm, welches die angegebenen Ausdrücke berechnet:

a. $(a+b)(a-b)$, a^2-b^2 (mit $a^2 = a \cdot a$, b^2 analog), $\frac{1}{3}$, $\frac{a}{b}$ für $a=1$, $b=3$ und $-3x^2+2y^2-\frac{1}{2}z^2$ für $x=2$, $y=3$ und $z=4$.
Verwenden Sie die Standardfunktion `printf();` für die Bildschirmausgabe.

b. a^2, a^{10}, $\sqrt{a}$, $\sqrt[3]{a}$, $\frac{a^3-1}{1+\sin^2 x}$, $\cot x$ für $a=8$ und $x=\frac{\Pi}{2}$.
Verwenden Sie für die Quadratwurzelberechnung die Standardfunktion ***double sqrt (double);*** für Potenzen mit Exponenten > 2 ***double pow (double, double);*** und die üblichen Winkelfunktionen ***double sin (double); double tan (double);*** ... aus der mathematischen Bibliothek (***# include <math.h>***). Π steht dort als Konstante ***M_PI*** zur Verfügung. Beim Übersetzen müssen diese mit ***-lm*** als Option hinzugebunden werden:
$ gcc -ansi -Wall -O ausdruecke_b.c -lm -o ausdruecke_b

c. Verwenden Sie die Standardfunktion `scanf();` für die Tastatureingabe zur Berechnung der Umfangs $2\Pi r$ und Flächeninhalts Πr^2 eines Kreises mit einzugebenen Radius *r*.

d. Berechnen Sie die Lösung $x_{1,2} = -\frac{p}{2} \pm \sqrt{\left(\frac{p}{2}\right)^2 - q}$ der quadratischen Gleichung $x^2 + px + q = 0$ bei einzugebenen *p* und *q* (*p*, *q* konstant).

3. terms_[a | b | c | d | e].c (C-Ausdrücke)
Geben Sie durch Klammerung oder durch Aufschreiben die Reihenfolge der Auswertung folgender Ausdrücke entsprechend dem jeweilig ersten Muster an und bestimmen Sie deren Ergebnisse. Vergleichen Sie mit denen vom Rechner!

a. Arithmetische Ausdrücke

```
# include <stdio.h>
int main()
{ float a = 1e10, b = 1e10, c = 1e-10;

/*          Operationen mit ganzen Zahlen            */
  printf( "%d\n", - 3 + 4 * 5 - 6);
                                  /*(-3+(4*5))-6 => 11*/
  printf( "%d\n", 3 + 4 % 5 - 6);
  printf( "%d\n", - 3 * 4 % - 6 / 5);
  printf( "%d\n", ( 7 + 6 ) % 5 / 2);
/*          Operationen mit reellen Zahlen           */
  printf( "%f\n", 1 / 2 * a);
  printf( "%f\n", a * ( a - b + c));
  printf( "%f\n", a * ( a + c - b));
  return 0;
}
```

b. Wertzuweisungen

```
# include <stdio.h>
int main()
{ int x = 2, y, z; float a = 1e10, d;

  x *= 3 + 2; printf( "%d\n", x);
                                  /*x=x*(3+2)=2*5 => 10*/
  x *= y = z = 4; printf( "%d\n", x);
  y = a * x; printf( "%d\n", y);
  d = a * x; printf( "%g\n", d);
  return 0;
}
```

c. Inkrementierung und Dekrementierung

```
# include <stdio.h>
int main()
{ int x, y, z, i, summe = 0; x = y = 1;

  z = y * x++;          printf( "%d %d %d\n", x, y, z);
  /*z = y * x; x++;                          2  1  1*/
  z = y * ++x;          printf( "%d %d %d\n", x, y, z);

  z = x ++ - 1;         printf( "%d %d %d\n", x, y, z);

  z += - y ++ + ++ x;   printf( "%d %d %d\n", x, y, z);

  z = x / ++ x;         printf( "%d %d %d\n", x, y, z);

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

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

```
  return 0;
}
```

d. Logische Ausdrücke (Bedingungen)

```
# include <stdio.h>
int main()
{ int x = 2, y = 1, z = 0;

  printf( "%d\n", x == ( y = z )); /*y=0 => 2==0 => 0*/
  printf( "%d %d %d\n", x, y, z);           /*2  0  0*/

  printf( "%d\n", x = y == z);
  printf( "%d %d %d\n", x, y, z);

  printf( "%d\n", x && y || z);
  printf( "%d\n", x || y && z );

  printf( "%d\n", x < y && x++ < z++);
  printf( "%d %d %d\n", x, y, z);

  printf( "%d\n", y < x && x++ < z++);
  printf( "%d %d %d\n", x, y, z);

  printf( "%d\n", x <= y <= z);
  printf( "%d\n", x <= y && y <= z);
  return 0;
}
```

e. Bedingte Ausdrücke

```
# include <stdio.h>
int main()
{ int x, y, z;
  x = y = z = 1;
  x += y += z;

  printf( "%d\n", x < y ? y : x);    /*3<2 ? 2:3 => 3*/
  printf( "%d %d %d\n", x, y, z);           /*3  2  1*/

  printf( "%d\n", z += x < y ? x ++ : y ++);
  printf( "%d %d %d\n", x, y, z);

return 0;
}
```

Anweisungen.dir

4. quad_[a | b | c].c (Schleifenanweisungen)
Schreiben Sie in ein neues Verzeichnis ***Anweisungen.dir*** je ein Programm zur Berechnung der Quadratzahlen von 1 bis 100 unter Verwendung

a. einer **while** – Schleife,
b. einer **do** – Schleife und
c. einer **for** – Schleife.

5. condition.c (logische Ausdrücke, geschachtelte for - Schleife)
a. Schreiben Sie die Aussagenverknüpfung „Wenn die Aussage A oder die Aussage B gilt, so gilt auch die Aussage C“ als C-Bedingung auf.

b. Berechnen Sie in einem C-Programm ***condition.c*** den Wahrheitswert der Aussagenverknüpfung bei eingegebenen Wahrheitswerten für A, B und C!

c. Verändern Sie Ihr Programm so, dass alle möglichen Wahrheitswerte dieser Aussagenverknüpfung in Abhängigkeit der Wahrheitswerte der Aussagen A, B und C berechnet werden!

6. powerof2.c (Bitoperator <<, Datentypen, while - Schleife)
a. Schreiben Sie ein C-Programm, welches Zweierpotenzen mittel Linksverschieben der Zahl 1 berechnet. Verwenden Sie dabei zunächst `unsigned char` als Datentyp und anschließend nur `char`. Bis zu welchen $n \in \mathrm{N}^+$ lässt sich jeweils auf diese Weise die Potenz $f(n) = 2^n$ ermitteln?

b. Verwenden Sie jetzt unterschiedliche Datentypen für ganze Zahlen. Untersuchen Sie, welche Datentypen sinnvoll sind, indem Sie alle *n* ermitteln, welche ein korrektes Ergebnis liefern. Geben Sie den Definitionsbereich und Wertevorrat der Funktion $f(n)$ in Abhängigkeit des verwendeten Datentyps an!

7. quadratic.c (if - Anweisung)
Schreiben Sie ein Programm zum Lösen quadratischer Gleichungen in der Normalform $x^2 + p \cdot x + q = 0$ mit $x, p, q \in R$. Beachten Sie dabei alle drei Lösungsfälle! Verwenden Sie für die Quadratwurzelberechnung die Standardfunktion ***double sqrt (double);*** aus der mathematischen Bibliothek (***# include <math.h>***). Beim Übersetzen muss diese mit ***-lm*** als Option hinzugebunden werden:
$ gcc -ansi -Wall -O quadratic.c -lm
Testen Sie Ihr Programm mit geeigneten Testwerten aus!

8. fraction.c (arithmetische Ausdrücke, for - Schleife)

a. Berechnen Sie in einem Programm ***fraction.c*** Funktionswerte der folgenden Funktion

$$f(x) = \frac{x-1}{x+1} : \frac{x^2-1}{x^2+1} \text{ , } (x^2 = x \cdot x).$$

Wählen Sie geeignete Testbeispiele für das Argument *x* aus und geben Sie diese ein!

b. Vereinfachen Sie $f(x)$. Erweitern Sie Ihr Programm so, dass es sowohl die Funktion unter a. als auch die vereinfachte Funktion $g(x)$ berechnet.

c. Verändern Sie das Programm ***fraction.c*** so, dass die Funktionen $f(x)$ und $g(x)$ für $x \in [-10,10]$ mit der Schrittweite 0.5 berechnet werden. Vergleichen Sie die Funktionswerte und versuchen Sie die unterschiedlichen Ergebnisse zu interpretieren.

9. compare.c

(arithmetische Ausdrücke, Schleifen, Eingabeumlenkung, Formatbeschreiber)
Berechnen Sie die Funktionswerte der Funktionen $f_1(x) = x^3$ und $f_2(x) = (-4+x)(-3+x)(-2+x)(-1+x)x(1+x)(2+x)(3+x)(4+x)$ für die folgenden neun *x*-Werte:
$\pm 4.001582369265,\ \pm 2.994625591579,\ \pm 2.005572782045,\ \pm 0.998616004359,\ 0$

Welche Schlussfolgerungen können Sie aus dem Ergebnis ziehen?

- Um die langen Zahlen nicht jedes Mal einzulesen, ist die Verwendung von Eingabeumlenkung empfehlenswert! Die Zahlenwerte werden untereinander in eine Datei geschrieben, abgespeichert und beim Start des Programm durch Umlenkung eingelesen:

```
$ nedit compare.dat &
$ a.out < compare.dat
```

Spiele.dir

Für viele Spiele benötigt man Zufallszahlen.
Die Funktion **`int rand( void);`** aus **`<stdlib.h>`** erzeugt Pseudozufallszahlen zwischen **`0`** und **`RAND_MAX.`** Die Zufallsfolge, die durch aufeinanderfolgende Aufrufe von **`rand();`** geliefert wird, ist reproduzierbar. Mit der Funktion **`void srand( unsigned int Startwert);`** kann festgelegt werden, mit welchem Starwert (sonst: **`1`**) die Zufallsfolge berechnet werden soll. Um den Startwert variabel zu halten, empfiehlt es sich, diesen aus der aktuellen Zeit zu berechnen. Dazu ist der Header **`# include <time.h>`** einzubinden und mit **`srand(( int) time(( time_t) NULL));`** die Startzeit zu setzen.

10. raten.c

Richten Sie ein neues Verzeichnis Spiele.dir ein und wechseln Sie dort hin.
Zahlenraten: Der Rechner erzeugt eine Zufallszahl welche zu erraten ist. Auf jeden Tipp des Spielers wird mit „zu klein“ oder „zu groß“ reagiert, bis er die Zahl richtig erraten hat.

11. wuerfeln.c

Würfelspiel: Zwei Spieler vereinbaren eine Rundenanzahl und würfeln abwechselnd, ihre Augenzahlen werden addiert. Gewonnen hat der Spieler mit der höheren Augensumme.

12. bandit.c

Der einarmige Bandit: Der Spieler hat ein Anfangskapital von 300 Euro. In jeder Runde setzt er einen Teil des Kapitals ein. Es werden drei Zahlen zwischen 0 und 9 gezogen. Sind alle drei Zahlen gleich, so erhält der Spieler das Vierfache, sind nur zwei der Zahlen gleich, so das Doppelte des Einsatzes zurück. Sind alle Zahlen paarweise verschieden, so wird ihm der Einsatz vom Kapital abgezogen. Das Spiel bricht ab, wenn der Spieler Pleite ist oder über 500 Euro erspielt hat und somit der Bandit Pleite ist.

Funktionen.dir

13. potenz.c (Funktionen, Standardfunktionen, Rundungsfehler)

a. Schreiben Sie ein Programm zur Berechnung der Potenz $p = x^k$ mit $x \in R^+$ und $k \in \mathrm{N}$ durch k - mal Multiplizieren von x.
   Algorithmus
   (1) $p = 1$
   (2) solange $k > 0$: setze $p = p * x$, $k = k - 1$
b. Berechnen Sie die Potenzen von x = 1 000 000 für k = 1, 2, ..., 7. Vergleichen Sie die Ergebnisse mit den mathematisch zu erwartenden Werten.

14. exptab_verbessert.c (Funktionen, Standardfunktionen, Fehlerfortpflanzung)

Laden Sie in ein neues Verzeichnis ***Funktionen.dir*** das Programm ***exptab.c*** für die Berechnung der Exponentialfunktion vom Netz und testen Sie es für $x = 1$ und $x = -10$ aus.

Die Fehler treten für $x < 0$ auf. Verbessern Sie deshalb das Verfahren, indem Sie $e^{|x|}$ berechnen. Für $x < 0$ ist dann $e^x = \frac{1}{e^{|x|}}$.

15. sintab.c (Funktionen, Standardfunktionen, Fehlerfortpflanzung)

a. Schreiben Sie eine Funktion ***double mysin(double);*** zur Berechnung des Sinus mittels Reihenentwicklung auf Rechnergenauigkeit:

$$\sin(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

   Testen Sie die Funktion durch ein geeignetes Testprogramm ***sintab.c***, indem Sie $\sin\left(\left(\frac{1}{6} + i\right) \cdot \Pi\right) = \frac{1}{2}$ für $i \in [0,20)$ tabellieren. Vergleichen Sie in ***sintab.c*** dabei Ihre Ergebnisse mit denen der Standardfunktion ***double sin(double);*** aus der mathematischen Bibliothek (# ***include <math.h>***)!

b. Verbessern Sie Ihr Programm durch eine weitere Funktion ***double mysin_verbessert(double);***, welche zunächst die Eingabewerte auf das dichtere Intervall $x \in \left[0, \frac{\Pi}{2}\right)$ transformiert und anschließend ***double mysin(double);*** aufruft. Starten Sie Ihr Testprogramm ***sintab.c*** erneut.

16. wurzel.c (Funktionen, Standardfunktionen)

Das Problem $x = \sqrt[k]{a}$ ist mittels Nullstellenbestimmung für eine beliebige natürliche Zahl $k \in \mathrm{N}$ mit $k > 1$ und eine positive reelle Zahl $a \in R$ mit $a > 0$, $a \neq 1$ zu lösen: $x = \sqrt[k]{a} \Rightarrow f(x) = x^k - a = 0$

Programmieren Sie zwei Funktionen, die mittels Näherungsverfahren die Nullstellen berechnen!

a. ***double mypow(double, unsigned int);***: Schreiben Sie zunächst zum Berechnen von Potenzen mit natürlichen Exponenten eine Hilfsfunktion und verwenden Sie diese bei den folgenden Funktionen.

b. ***double newton_wurzel(double, unsigned int, double);***: Verwenden Sie zur Wurzelberechnung durch Nullstellenbestimmung das Newtonsche Iterationsverfahren, welches sich leicht aus der Punktrichtungsgleichung herleiten lässt, als Tangentennäherungsverfahren bekannt:

   Iterationsanfang: Setze $x_0 = a$

   Iterationsschritt: $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n}{k}\left(1 - \frac{a}{x_n^k}\right)$

   Iterationsabbruch: $|f(x_{n+1})| < \varepsilon, \qquad 0 < \varepsilon \in R$

c. ***double regulafalsi_wurzel(double, unsigned int, double, double);***: Verwenden Sie zur Wurzelberechnung durch Nullstellenbestimmung das Iterationsverfahren Regula falsi, welches analog aus der Zweipunktegleichung herleitbar und auch als Sekantennäherungsverfahren bekannt ist:

   Iterationsanfang: Setze x_0 und x_1, so dass $\mathrm{sgn}(f(x_0)) = -\mathrm{sgn}(f(x_1))$.

   Iterationsschritt: $x_{n+2} = x_n - \frac{x_{n+1} - x_n}{f(x_{n+1}) - f(x_n)} \cdot f(x_n)$.

   Falls $\mathrm{sgn}(f(x_{n+2})) = \mathrm{sgn}(f(x_{n+1}))$, dann setze $x_{n+1} = x_n$.

   Iterationsabbruch: $|f(x_{n+2})| < \varepsilon, \qquad 0 < \varepsilon \in R$

d. Berechnen Sie in einem Testprogramm ***wurzel.c*** mit beiden Verfahren $x = \sqrt[3]{27}$ mit $\varepsilon = 10^{-t}$ für $t \in [1,5] \subset \mathrm{N}$. Geben Sie jeweils die notwendige Schrittzahlen n an. Vergleichen Sie Ihre Testergebnisse mit dem Ergebnis der Standardfunktion ***double pow(double, double);*** aus ***<math.h>***.

17. makefile
Nehmen Sie das Programm ***potenz.c*** aus der Vorlesung in das Verzeichnis ***Funktionen.dir*** auf. Schreiben Sie ein ***makefile***, welches die Programme ***potenz.c***, ***exptab.c***, ***exptab_verbessert.c***, ***sintab.c*** und ***wurzel.c*** wartet.

---

ggTkgV.dir

---

18. ggTkgV.c (Funktionen, make)
Gegeben sei das folgende Modul:

ggTkgV.c

```
# include <stdio.h>
# include "euklid.h"
# include "uintio.h"

int main()
{ unsigned int m, n;

  m = uintEingabe( "Bitte nat. Zahl eingeben: ");
  n = uintEingabe( "Bitte nat. Zahl eingeben: ");

  uintAusgabe( "ggT = ", ggT( m, n));
  uintAusgabe( "kgV = ", kgV( m, n));

  return 0;
}
```

a. Richten Sie ein Verzeichnis ***ggTkgV.dir*** ein, speichern Sie das obige Modul in das Verzeichnis.

b. Fassen Sie die Deklarationen der notwendigen Funktionen im dem Header ***euklid.h*** und dem Header ***uintio.h*** zusammen und kommentieren Sie diese (Funktionalität, Eingabeparameter, Funktionswert):

uintio.h (Ein- und Ausgabe natürlicher Zahlen)
unsigned int uintEingabe(const char *);
void uintAusgabe(const char *, unsigned int);
euklid.h (Berechnung des ggT und des KgV)
unsigned int ggT(unsigned int, unsigned int);
unsigned int kgV(unsigned int, unsigned int);

c. Definieren Sie in den Modulen ***euklid.c*** und ***uintio.c*** die in dem entsprechenden Header deklarierten Funktionen.
Berechnen Sie den größten gemeinsamen Teiler und das kleinste gemeinsame Vielfache von 2 natürlichen Zahlen mittels Euklidschen Algorithmus unter Anwendung der ganzzahlige Division und Modulo(/, %):

ggT(r_0, r_1): $r_0 = r_1 * v_1 + r_2$
$r_1 = r_2 * v_2 + r_3$
$r_2 = r_3 * v_3 + r_4$

$$\ldots$$
$$r_n = r_{n+1} * v_{n+1} \qquad ggT(r_0, r_1) = r_{n+1}$$
$$kgV(r_0, r_1) = (r_0 * r_1) / r_{n+1}$$

d. Übersetzen Sie mit dem hier gegebenen ***makefile*** des Programmpaket und testen Sie Ihre Funktionen aus. Wozu ist die letzte Zeile nützlich?

makefile

```
# makefile fuer ggTkgV

CC = gcc
CFLAGS = –ansi -Wall -O -c
OBJ = uintio.o euklid.o ggTkgV.o

ggTkgV: $(OBJ)
   $(CC) $(OBJ) -o ggTkgV

ggTkgV.o: uintio.h euklid.h ggTkgV.c
   $(CC) $(CFLAGS) ggTkgV.c

euklid.o: euklid.h euklid.c
   $(CC) $(CFLAGS) euklid.c

uintio.o: uintio.h uintio.c
   $(CC) $(CFLAGS) uintio.c

clean:
   rm -f $(OBJ)
```

Geschichte.dir

---

19. Es_war_enmal.c (Rekursion)
Eine unendliche Geschichte

>>
Es war einmal ein Mann, der hatte sieben Söhne. Die sieben Söhne sprachen: „Vater erzähle uns eine Geschichte!“ Da fing der Vater an:

Es war einmal ein Mann, der hatte sieben Söhne. Die sieben Söhne sprachen: „Vater ...

Und wenn sie nicht gestorben ist, so erzählen sie noch heute.
<<

```
void Geschichte()
{
  if( Mann_lebt_noch())
  { Erzaehlung(); Geschichte();
  }
  return;
}
```

Vollenden Sie das Programm ***Es_war_einmal.c***, welches die unendliche Geschichte beliebig oft erzählt, in dem sie die in der rekursiven Funktion ***Geschichte()*** angegebenen Funktionen ***Mann_lebt_noch()*** und ***Text()*** ergänzen und ein Hauptprogramm dazu schreiben.

Summe.dir

20. summe.c (iterative und rekursive Funktionen)
Deklarieren und definieren Sie in ***summe.***c

a. eine Funktion ***unsigned long it_sum(unsigned long);*** zur iterativen Berechnung der Summe s der ersten n natürlichen Zahlen ($n > 0$) durch Aufsummieren $s = \sum_{i=1}^{n} i$,

b. eine Funktion ***unsigned long rek_sum(unsigned long);*** zur rekursiven Berechnung der Summe s der ersten n natürlichen Zahlen ($n > 0$) mit $s = n + \sum_{i=1}^{n-1} i$

c. und eine Funktion ***unsigned long gauss_sum(unsigned long);*** zur Berechnung der Summe s der ersten n natürlichen Zahlen ($n > 0$) mittels der Formel $s = \frac{n \cdot (n+1)}{2}$ von **Carl Friedrich Gauß (1777-1855)**, die er als neunjähriger in der Anfängerschule dem erstaunten Lehrer Büttner lieferte.
Beweisen Sie die Gauß-Formel durch vollständige Induktion!

d. Testen Sie im Hauptprogramm von ***summe.c*** die drei Funktionen geeignet aus, indem Sie für eingegebene Werte $n \in \mathrm{N}^+$ die Ergebnisse der Funktionen miteinander vergleichen. Verwenden Sie zum Übersetzen ein ***makefile***.

e. Bis zu welchem $n \in \mathrm{N}^+$ sind mit den obigen Funktionen korrekte Lösungen zu erwarten? Schätzen Sie den maximalen Wert bei der iterativen Lösung und der Gauß-Lösung unter Verwendung der Gauß-Formel und **ULONG_MAX = $2^{32} - 1$** ab. Der Maximalwert für die rekursive Lösung liegt zwischen den beiden erhaltenen Grenzen und lässt sich durch logarithmisches Halbieren ermitteln.
Sichern Sie ab, dass die Funktionen nur zulässige Eingabewerte berechnen.

Kombinatorik.dir

21. dblFak.c (Rekursion, Datentypen)
Die Fakultätsberechnung aus der Vorlesung ist durch die Datentypen stark eingeschränkt. Mit dem in der Vorlesung gegebenen Beispielen ***unsigned long itFak (int);*** und ***unsigned long rekFak (int);*** kann man nur die Fakultäten bis zu der natürlichen Zahl 12 berechnen. Um dies zu verbessern, programmieren Sie die Fakultät als Funktionen vom Typ ***double itFak (int);*** und ***double rekFak (int);***. Bis zu welcher natürlichen Zahl ist ein korrektes Ergebnis zu erwarten? Beweisen Sie Ihre Vermutung unter Verwendung der Anzahl der gespeicherten Mantissenstellen **DBL_DIG = 52** des Datentyps ***double***!
Erweitern Sie dass ***makefile*** aus der Vorlesung entsprechend!

22. n_ueber_k.c
(Rekursion, Effektivität, Typumwandlung: Gleitkommazahlen in ganze Zahlen)
$\binom{n}{k} = \frac{n!}{k!(n-k)!}$ für $n,k \in \mathrm{N}$ mit $k \le n$ und $\binom{n}{0} = 1$ bezeichnet man als ***Binomialkoeffizienten***.

a. Schreiben Sie eine Funktion ***double it_bk(unsigned long, unsigned long);***, welche $\binom{n}{k}$ iterativ berechnen. Um vorzeitige Überläufe zu vermeiden, wird die Fakultätsfunktion hier nicht eingesetzt, sondern man multipliziert und dividiert im Wechsel. Achten Sie auf die Effektivität Ihrer Implementierung und berücksichtigen Sie für $k > n-k$ die Rechenregel $\binom{n}{k} = \binom{n}{n-k}$.

b. Für eine rekursive Implementierung ***double rek_bk(unsigned long, unsigned long);*** verwenden Sie die Formel $\binom{n+1}{k+1} = \binom{n}{k} \cdot \frac{n+1}{k+1}$.

c. Eine weitere rekursive Implementierung ***unsigned long rek_rek_bk(unsigned long, unsigned long);*** kann man aus dem Pascalschen Dreieck herleiten, welches die Beziehung $\binom{n+1}{k+1} = \binom{n}{k+1} + \binom{n}{k}$ nutzt.

d. Testen Sie die iterative und beide rekursiven Funktionen in einem geeigneten Testprogramm ***n_ueber_k.c*** aus, erweitern Sie das ***makefile*** entsprechend und vergleichen Sie die Effizienz der beiden rekursiven Implementierungen: Wie oft werden die Funktionen bei gegebenen *n* und *k* aufgerufen?

e. Berechnen Sie mit Ihrem Testprogramm $\binom{19}{17}$. Wandeln Sie die Ergebnisse der drei Funktionen in den Datentype ***unsigned long*** um und geben Sie diese aus! Wie erklären Sie sich den Fehler? Wie können Sie solche Fehler verhindern?

Ausdruecke.dir

---

23. terms_f.c (Ausdrücke mit Zeigern)
Wechsel Sie in das Verzeichnis ***Ausdruecke.dir***. Ermitteln Sie die Reihenfolge der Auswertung folgender Ausdrücke und bestimmen Sie deren Wert:

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

int main()
{ int i[ 3], *z;
  i[ 0]=10; i[ 1]=20; i[ 2]=30;
```

```
  for( z=i; z<i+3; z++) printf( " %d", *z);
  printf( "\n");

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

  for( z=i; z<i+3; z++) printf( " %d", *z);
  printf( "\n");

  return 0;
}
```

24. string.c (Ausdrücke mit Zeigern, Zeiger und Felder)
Analysieren Sie das nachfolgende Programm ***string.c***. Welche Funktionalität liegt bei der Funktion ***void stringFunktion(char *, char *);*** vor?
Wie erklären Sie sich die Warnung beim Compilieren (***-Wall***)?

string.c

```
# include <stdio.h>
# define LAENGE 256
/* Funktionsdeklaration mit Zeiger als Parameter */
void stringFunktion( char *, char *);

int main()
{ char S1[LAENGE], S2[LAENGE];
  printf( "Stringeingabe (max. 255 Zeichen!):");

/* Funktionsaufrufe, Uebergabe der Feldadressen */
  scanf( "%255s", S1);
  stringFunktion( S2, S1);
  printf( "Stringausgabe: %s\n" , S2);
  return 0;
}

/* Funktionsdefinition mit Zeigerarithmetik */
void stringFunktion( char *Z, char *Q)
{while( *Z++ = *Q++);
  return;
}
```

Mittelwert.dir , Scalar.dir

25. scalar.c (Felder, make)

a. Holen Sie sich das Programmpaket zur Mittelwertberechnug (***mwert.c, vektorio.h, vektorio.c, makefile***) aus dem Netz und speichern Sie es in ein neues Verzeichnis ***Mittelwert.dir***. Übersetzen Sie es mit ***make*** und starten Sie es!

b. Stellen Sie analog ein Programmpaket in einem Verzeichnis ***Scalar.dir*** zur Berechnung des Skalarproduktes $\vec{a} \cdot \vec{b} = \sum_{i=0}^{n} a_i \cdot b_i$ zweier Vektoren $\vec{a} = (a_0 a_1 \cdots a_n)$ und $\vec{b} = (b_0 b_1 \cdots b_n)$ zusammen. Schreiben Sie in einem eigenen Modul ***scalar.c*** die notwendige Funktion zur Ermittlung des Skalarproduktes und das Hauptprogramm. Nutzen Sie für die Vektorein- und ausgabe die unter a. zur Verfügung stehenden Funktionen des Moduls ***../Mittelwert.dir/vektorio.c***. Binden Sie dazu das Modul in ***scalar.c*** und im dazugehörigen ***makefile*** ein.

Kombinatorik.dir

26. permutation.c (Felder, Rekursionen)

a. Schreiben Sie ein Programm ***permutation.c*** zur rekursiven Berechnung aller Permutationen ohne Wiederholung der Zahlen $z \in \{1,2,3,4,5,6\}$ und erweitern Sie das im Verzeichnis bereits angelegte ***makefile*** entsprechend.
Zum Speichern der Permutationen vereinbaren Sie einen Vektor ***int P[6];*** global. In ihm werden die einzelnen Permutationen entwickelt, nach Fertigstellung einer Permutation wird diese ausgegeben.

Rekursiver Algorithmus zur Berechnung der Permutationen:
(***i – 1***) Komponenten von ***P*** sind bereits besetzt, ***i*** - te Komponente wird behandelt:

1. ***i > 6***: Permutation wird ausgegeben.
2. ***i*** ≤ 6: Für alle $z \in \{1,2,3,4,5,6\}$:
   Ist z erlaubt, d.h. $z \notin \{P[0], P[1], \cdots, P[i-2]\}$, so
   a) $P[i-1] = z$, ***i*** Komponenten von ***P*** sind damit besetzt und
   b) (***i+1***) - te Komponente wird behandelt. (Rekursion)

Für diese Aufgabenstellung sind folgende Funktionen zu entwickeln:

void permutation(int i);
rekursive Funktion zum Erzeugen der Permutationen entsprechend dem Algorithmus.

void permutationAusgabe(void);
Ausgabe einer erzeugten Permutation.

int erlaubt(int z, int i);
überprüft, ob eine Zahl ***z*** in der Permutation bis zur (***i – 1***) - ten Komponente berücksichtigt wurde.
Rückgabewerte:
1, falls ***z*** noch nicht in der Permutation bis zur (***i – 1***) - ten Komponente enthalten; 0, sonst.

b. Verallgemeinern Sie Ihr Programm, indem Sie ein Makro einführen ***(# define PERMUTATION_SIZE 6***), so dass durch dessen Wertänderung und anschließender Übersetzung, die Permutationen für beliebige $n \in \mathrm{N}^+$ mit $z \in \{1,2,\cdots,n\}$ berechnet werden können.

Brueche.dir

---

27. brueche_[1 | 2].c (Strukturen)
Schreiben Sie ein Programmpaket zum Rechnen mit gemeinen Brüchen.

a. Richten Sie dazu in einem neuen Verzeichnis ***Brueche.dir*** ein Modul ***bruch.c*** und dessen Header ***bruch.h*** ein und deklarieren Sie in ihm die Struktur ***struct bruch***, welche einen ganzzahligen ***Zaehler*** und einen ganzzahligen ***Nenner*** als Elemente besitzt.

b. Programmieren Sie Funktionen zur Eingabe ***struct bruch bruchEingabe(void);*** und Ausgabe ***void bruchAusgabe(struct bruch);*** eines Bruches. Beachten Sie, dass es keinen Bruch mit dem Nenner 0 gibt. In diesem Fall soll **NaN** für **Not a Number** am Bildschirm erscheinen. Eine Funktion ***unsigned int isvalid(struct bruch);*** soll eine **1** liefern, falls ein Bruch eine gültige Zahl darstellt, sonst eine **0**.

c. Übernehmen Sie den Euklidschen Algorithmus ***unsigned int ggT(unsigned int, unsigned int);*** aus einer der vorherigen Aufgaben (12. c.) zum Kürzen eines Bruches ***struct bruch kuerzen(struct bruch);***. Da der Algorithmus nicht für ganze Zahlen geeignet ist, müssen Sie mit dem Absolutbeträgen des Zählers und des Nenners arbeiten und das Vorzeichen aus den Vorzeichen des Zählers und des Nenners neu berechnen. Dazu ist es zweckmäβig, diese durch entsprechende Funktionen bereit zu stellen:

int signum(int);
liefert eine **1**, falls eine ganze Zahl positiv, eine **0**, falls sie Null und ein **–1**, falls sie negativ ist.
unsigned int intAbs(int);
liefert den Absolutbetrag einer ganzen Zahl.

d. Schreiben Sie zusätzlich die unären Operationen ***struct bruch inverse(struct bruch);*** zum Erzeugen des Kehrwertes und ***struct bruch neg(struct bruch);*** zum Erzeugen des negativen Wertes eines Bruches.

e. Realisieren Sie schlieβlich die binären Operationen ***struct bruch add(struct bruch, struct bruch);*** zum Addieren, ***struct bruch sub(struct bruch, struct bruch);*** zum Subtrahieren, ***struct bruch multt(struct bruch, struct bruch);*** zum Multiplizieren und ***struct bruch div(struct bruch, struct bruch);*** zum Dividieren zweier Brüche, wobei die Ergebnisse gekürzt zurückgegeben werden sollen.

f. Verwenden Sie zum Testen der einzelnen Komponenten das mitgelieferte ***makefile*** und das Testprogramm ***brueche_1.c***.

makefile
```
# makefile fuer brueche_1
```

```
CC = gcc
CFLAGS = -ansi -Wall -O -c
OBJ = brueche_1.o bruch.o

brueche_1: $(OBJ)
  $(CC) $(OBJ) -o brueche_1

brueche_1.o: brueche_1.c bruch.h
  $(CC) $(CFLAGS) brueche_1.c

bruch.o: bruch.c bruch.h
  $(CC) $(CFLAGS) bruch.c

clean:
  rm -f $(OBJ)
```

brueche_1.c

```
/********* Testen der Funktionen aus bruch.h *********/

#include <stdio.h>
#include "bruch.h"

int main()
{ struct bruch t1[] =
  {{12,8},{0,8},{8,0},{0,0},{-12,8},{12,-8},{-12,-8}};
  struct bruch t2[] =
  {{12,8},{3,4},{2,1},{4,2},{0,2},{1,3},{2,3},{1,0},
  {1,4},{0,-7},{1,0},{2,3}};
  int i;

  printf( "Test 1: kuerzen\n");
  for( i = 0; i < sizeof( t1) / sizeof( t1[ 0]); i++)
  { printf( "%d/%d => ",
            t1[ i].Zaehler, t1[ i].Nenner);
    bruchAusgabe( kuerzen( t1[ i])); printf( "\n");
  }

  printf( "\nTest 2: inverse\n");
  for( i = 0; i < sizeof( t1) / sizeof( t1[ 0]); i++)
  { printf( "%d/%d => ",
            t1[ i].Zaehler, t1[ i].Nenner);
    bruchAusgabe( inverse( t1[ i])); printf( "\n");
  }

  printf( "\nTest 3: neg\n");
  for( i = 0; i < sizeof( t1) / sizeof( t1[ 0]); i++)
  { printf( "%d/%d => ",
            t1[ i].Zaehler, t1[ i].Nenner);
    bruchAusgabe( neg( t1[ i])); printf( "\n");
  }

  printf( "\nTest 4: add\n");
```

```
  for( i = 0; i < sizeof( t2) / sizeof( t2[ 0]); i +=2)
  { printf( "%d/%d + %d/%d => ",
    t2[ i].Zaehler, t2[ i].Nenner,
    t2[ i+1].Zaehler, t2[ i+1].Nenner);
    bruchAusgabe( add( t2[ i], t2[ i+1]));
    printf( "\n");
  }

  printf( "\nTest 5: sub\n");
  for( i = 0; i < sizeof( t2) / sizeof( t2[ 0]); i +=2)
  { printf( "%d/%d - %d/%d => ",
    t2[ i].Zaehler, t2[ i].Nenner,
    t2[ i+1].Zaehler, t2[ i+1].Nenner);
    bruchAusgabe( sub( t2[ i], t2[ i+1]));
    printf( "\n");
  }

  printf( "\nTest 6: mult\n");
  for( i = 0; i < sizeof( t2) / sizeof( t2[ 0]); i +=2)
  { printf( "%d/%d * %d/%d => ",
    t2[ i].Zaehler, t2[ i].Nenner,
    t2[ i+1].Zaehler, t2[ i+1].Nenner);
    bruchAusgabe( mult( t2[ i], t2[ i+1]));
    printf( "\n");
  }

  printf( "\nTest 7: div\n");
  for( i = 0; i < sizeof( t2) / sizeof( t2[ 0]); i +=2)
  { printf( "%d/%d / %d/%d => ",
    t2[ i].Zaehler, t2[ i].Nenner,
    t2[ i+1].Zaehler, t2[ i+1].Nenner);
    bruchAusgabe( div( t2[ i], t2[ i+1]));
    printf( "\n");
  }

  return 0;
}
```

g. Berechnen Sie in einem zweiten Testprogramm ***brueche_2.c*** folgende gemeinen Brüche und geben Sie deren Ergebnisse am Bildschirm aus. Erweitern Sie das ***makefile*** entsprechend.

$$\left(\frac{3}{1}\cdot\frac{3}{2}+\left(\frac{5}{12}\cdot\left(-\left(\frac{1}{4}+\frac{1}{8}\right)+\left(\frac{7}{2}+\frac{7}{8}\right)\right)\right)\right)\div\frac{5}{2}$$

$$\left(\left(\left(\frac{9}{2}\cdot\frac{1}{3}+\frac{7}{3}\right)+\frac{7}{1}\div\frac{3}{2}\right)-\left(\left(\frac{7}{9}\cdot\frac{3}{1}+\frac{11}{3}\right)+\frac{5}{2}\right)\right)^{-1}$$

Integration.dir

28. integration.c

(Zeiger auf Felder und Funktionen, dynamische Speicherplatzverwaltung)

a. Laden Sie das Programm zur numerischen Integration ***integration.c*** aus der Vorlesung in ein neues Verzeichnis ***Integration.dir*** und erweitern Sie das Programm, indem Sie im Fall 3 beliebige Polynome $P_n(x) = \sum_{i=0}^{n} a_i x^i$ mit $a_i \in R$ vom Grade $n \in N^+$ zulassen. Führen Sie dazu ein neues Modul ***polynom.c*** (***polynom.h***) ein.

b. Deklarieren Sie in ihm global eine Variable ***int Grad = 0;*** für den Polynomgrad und einen Zeiger ***double *Polynom;*** auf das Feld der Komponenten des Polynoms.

c. Deklarieren und definieren Sie eine Funktion ***void polynomEingabe(void);*** zum Einlesen eines Polynoms, indem Sie, je nach Grad des Polynoms, über ***void *malloc(size_t);*** entsprechenden Speicher dynamisch zur Verfügung stellen und das Polynom komponentenweise einlesen. Um den dynamisch bereitgestellten Speicher wieder freizugeben, ist außerdem eine Funktion ***void polynomLoeschen(void);*** aufzunehmen.

d. Löschen Sie die Deklaration und Definition der alten Funktion ***double polynom(double);*** in ***integration.c*** und ersetzen Sie diese im neuen Modul durch eine entsprechende Funktion ***double polynom(double);*** zur Berechnung des Funktionswertes eines Polynoms mittels HORNER-Schema:

$$P_n(x) = \sum_{i=0}^{n} a_i x^i = \left(\left(\cdots\left(\left(\left(a_n \cdot x + a_{n-1}\right)\cdot x\right) + a_{n-2}\right)\cdot x + \cdots + a_2\right)\cdot x + a_1\right)\cdot x + a_0 .$$

e. Entwickeln Sie parallel dazu ein ***makefile*** und ergänzen Sie das Hauptprogramm ***integration.c*** so, dass ein Polynom bei der Wahl von Fall 3 eingelesen und berechnet werden kann.

Liste.dir

---

29. punktListe.c (einfach verkettete Listen)

a. Übernehmen Sie das Programmpaket zur Arbeit mit Listen (***punktListe.c***, ***liste.h***, ***liste.c***, ***makefile***) in ein neues Verzeichnis ***Liste.dir***. Schreiben Sie eine Funktion ***void printPunkt(struct Punkt);*** zur Ausgabe der Koordinaten eines Punktes und, unter Verwendung dieser, die Funktion ***void printListe(struct Punkt *);*** zur Ausgabe der sortierten Liste in das Modul ***liste.c*** (***liste.h***) und aktualisieren Sie ***punktListe.c*** so, dass die Polarkoordinaten **[1, 0°]** bis **[1, 360°]** im Abstand von **45°** in eine Liste aufgenommen werden. Drucken Sie diese Liste aus und kommentieren Sie die Ergebnisse!

b. Vervollständigen Sie die Listenoperationen im Modul ***liste.c*** (***liste.h***), indem Sie die Funktion ***struct Punkt *kar_suchen(struct Punkt *, struct Punkt);*** aus der Vorlesung übernehmen, die Funktionen ***struct Punkt *pol_suchen(struct Punkt *, struct Punkt);***, ***struct Punkt *kar_entfernen(struct Punkt **, struct Punkt);*** und ***struct Punkt *pol_entfernen(struct Punkt **, struct Punkt);*** unter Berücksichtigung der in der Vorlesung betrachteten Effektivität neu schreiben.
Welche Risiken bringt die direkte Nutzung der Funktion ***struct Punkt *kar_suchen(struct Punkt *, struct Punkt);*** bzw. ***struct Punkt *kar_entfernen(struct Punkt **, struct Punkt);*** durch einen Anwender mit sich?

c. Schreiben Sie eine weitere Funktion ***void inverse(struct Punkt **);*** zum Umkehren einer Liste, d.h. das Ende wird der Anfang der Liste.

d. Erweitern Sie das Hauptprogramm ***punktListe.c*** so, das auch die neuen Funktionen ausgetestet werden können.