

# Schleifen und bedingte Ausführung

```
/* Datei: beispiel-2.2.c      Datum: 12.4.2016 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>          /* mathematische Funktionen */

double f(double x)      /* Funktion f gibt double-Wert zurueck und hat ein Argument "x" mit dem Typ double */
{
    return sin(x);       /* es wird der sin des Argument zurueckgeben */ } /* Ende f */

int main()
{ int i,n=10;           /* i,n sind int Variablen, n wird mit 100 vorbelegt */
    double y[100];        /* Speicher fuer 100 Gleitkommazahlen */

/* "for" Schleife */
    for(i=0;i<n;i++)
        /* 1) Startwert fuer i=0  2) ausfuehren solange i<n ist  3) am Ende i um 1 erhoehen (i++) und wieder zu 2 */
    {
        /* Block mit Anweisungen */
        y[i]=f(1.5*i);
    }

/* while-Schleife */
    i=0;
    while(i!=n)           /* solange Ausfuehren wie i!=n */
    {
        y[i]=f(1.5*i);
        i=i+1;             /* entspricht der i++ Anweisung von oben */
    }

/* do-while-Schleife
   Bedingung wird am Ende geprueft <=> Schleife wird mindestens 1x ausgefuehrt */

    i=0;
    do
    {
        y[i]=f(1.5*i);
        i=i+1;             /* entspricht der i++ Anweisung von oben */
    }
    while(!(i>=n));        /* solange Ausfuehren wie !(i>=n) */ } /* Ende main */
```

**Schleifen ermöglichen auf einfache Weise ähnliche Anweisung vielfach auszuführen.**

## 1) “for” Schleife

## 2) “while” Schleife

## 3) “do-while” Schleife

**Format des Codes ist fragwürdig!**

```
/* Datei: beispiel-2.2.c      Datum: 12.4.2016 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>          /* mathematische Funktionen */

double f(double x)      /* Funktion f gibt double-Wert zurueck und hat ein Argument "x" mit dem Typ double */
{
    return sin(x);       /* es wird der sin des Argument zurueckgeben */ } /* Ende f */

int main()
{ int i,n=10;           /* i,n sind int Variablen, n wird mit 100 vorbelegt */
    double y[100];        /* Speicher fuer 100 Gleitkommazahlen */

/* "for" Schleife */
    for(i=0;i<n;i++)
        /* 1) Startwert fuer i=0  2) ausfuehrer */
    { /* Block mit Anweisungen */
        y[i]=f(1.5*i);
    }

/* while-Schleife */
    i=0;
    while(i!=n)           /* solange Ausfuehren wie i!=n */
    {
        y[i]=f(1.5*i);
        i=i+1;            /* entspricht der i++ Anweisung von oben */
    }

/* do-while-Schleife
   Bedingung wird am Ende geprueft <=> Schleife wird mindestens 1x ausgefuehrt */

    i=0;
    do
    {
        y[i]=f(1.5*i);
        i=i+1;            /* entspricht der i++ Anweisung von oben */
    }
    while(!(i>=n));       /* solange Ausfuehren wie !(i>=n) */ } /* Ende main */
```

**Beachte:**

**Definition der zweiten Funktion "f"**  
**zusätzlich math.h**

**for-Schleife: typisch für Durchlaufen eines Index**  
**erster Parameter: setzen des Startwertes**  
**zweiter Parameter: Bedingung die Schleife beendet**  
**dritter Parameter: Operation zum Erhöhen des Index**

```
/* Datei: beispiel-2.2.c      Datum: 12.4.2016 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>          /* mathematische Funktionen */

double f(double x)      /* Funktion f gibt double-Wert zurueck und hat ein Argument "x" mit dem Typ double */
{
    return sin(x);       /* es wird der sin des Argument zurueckgeben */ } /* Ende f */

int main()
{ int i,n=10;           /* i,n sind int Variablen, n wird mit 100 vorbelegt */
    double y[100];        /* Speicher fuer 100 Gleitkommazahlen */

/* "for" Schleife */
    for(i=0;i<n;i++)
        /* 1) Startwert fuer i=0  2) ausfuehren solange i<n ist  3) am Ende i um 1 erhoehen (i++) und wieder zu 2 */
    {
        /* Block mit Anweisungen */
        y[i]=f(1.5*i);
    }

/* while-Schleife */
    i=0;
    while(i!=n)           /* solange Ausfuehren wie i!=n */
    {
        y[i]=f(1.5*i);
        i=i+1;             /* entspricht der i++ Anweisung von oben */
    }

/* do-while-Schleife
   Bedingung wird am Ende geprueft <=> Schleife wird mindestens 1x ausgefuehrt */

    i=0;
    do
    {
        y[i]=f(1.5*i);
        i=i+1;             /* entspricht der i++ Anweisung von oben */
    }
    while(!(i>=n));        /* solange Ausfuehren wie !(i>=n) */ } /* Ende main */
```

**while-Schleife:**  
**Parameter ist eine Bedingung, die Schleife beendet**

```
/* Datei: beispiel-2.2.c      Datum: 12.4.2016 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>          /* mathematische Funktionen */

double f(double x)      /* Funktion f gibt double-Wert zurueck und hat ein Argument "x" mit dem Typ double */
{
    return sin(x);       /* es wird der sin des Argument zurueckgeben */ } /* Ende f */

int main()
{ int i,n=10;           /* i,n sind int Variablen, n wird mit 100 vorbelegt */
    double y[100];        /* Speicher fuer 100 Gleitkommazahlen */

/* "for" Schleife */
    for(i=0;i<n;i++)
        /* 1) Startwert fuer i=0  2) ausfuehren solange i<n ist  3) am Ende i um 1 erhoehen (i++) und wieder zu 2 */
    {
        /* Block mit Anweisungen */
        y[i]=f(1.5*i);
    }

/* while-Schleife */
    i=0;
    while(i!=n)        /* solange Ausfuehren wie i!=n */
    {
        y[i]=f(1.5*i);
        i=i+1;        /* entspricht der i++ Anweisung von oben */
    }

/* do-while-Schleife
   Bedingung wird am Ende geprueft <=> Schleife wird mindestens einmal durchlaufen. */
    i=0;
    do
    {
        y[i]=f(1.5*i);
        i=i+1;        /* entspricht der i++ Anweisung von oben */
    }
    while(!(i>=n));     /* solange Ausfuehren wie !(i>=n) */ } /* Ende main */
```

### do-while-Schleife:

**Parameter ist eine Bedingung, die Schleife beendet Test am Ende der Schleife, d.h. die Schleife wird mindestens einmal durchlaufen.**

Alle Schleifen benötigen **Bedingungen** also logische Operationen

Vergleiche:

`==, <=, >=, <, >, !=` → falsch/wahr, d.h 0 bzw. ≠0

logische Verkäpfungen:

`&&, ||, !` für “und”, “oder” und “nicht”

Beispiele: `(n<=1) || (n>10)`

`! ((a==1.5) && (b==2.0))`

**Achtung:** typischer Anfängerfehler ist “=” anstatt “==” zu benutzen.

Der C Compiler findet das in der Regel akzeptable, aber **das Ergebnis ist anders!**

Vergleiche von Gleitkommazahlen mit “==” sind ebenfalls problematisch

`1.5 != 1.499999999`

## Bedingte Ausführung

**if-Konstrukt ermöglicht die bedingte Ausführung von Programmteilen**

```
/* Datei: beispiel-2.3.c    Datum: 12.4.2016 */

...
double x,y;
...

/* if-Anweisung */
if(fabs(x)<0.3)          /* Bedingung */
{
    y = 1+x+0.5*pow(x,2); /* Block wird ausgefuehrt falls Bedingung "wahr" */
}
else
{
    y=exp(x);           /* Block wird ausgefuehrt falls Bedingung "falsch" */
}

...
...
```

Das schließt die sehr kleine Einführung in C ab.

Weitere Konstrukte, arithmetische Operationen, mathematische Funktionen, ... werden wir in den Beispielen und Übungen kennenlernen.

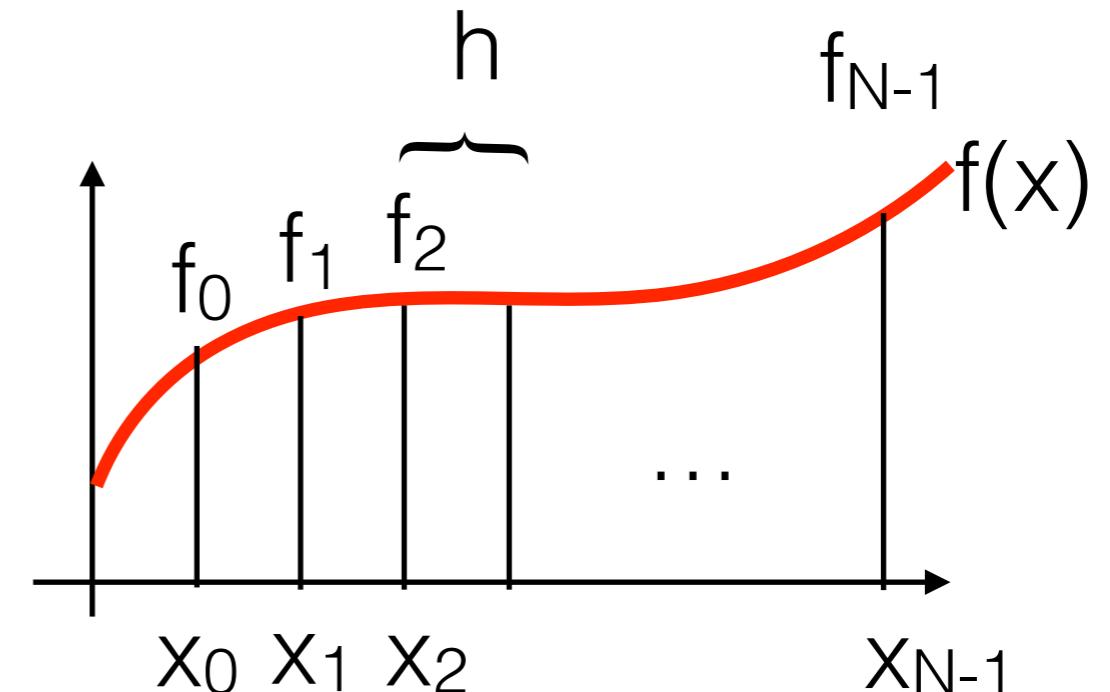
### 3. Einfache mathematische Operationen

#### Numerische Ableitung

Numerische Darstellung einer Funktion

glatte Funktionen

→ genügend gute Approximation  
auf einem Gitter  $\{x_i\}$



Wir nehmen hier (und fast im gesamten Verlauf der Vorlesung) an, dass die Gitterpunkte gleichmäßig verteilt sind. Bei genügend kleinem Abstand **h** kann man die Funktion mit wenigen Termen in einer Taylor-Reihe annähern:

$$f_{i+1} = f(x_i + h) = f_i + f'(x_i) h + \frac{1}{2} f''(x_i) h^2 + \mathcal{O}(h^3) \quad (1)$$

$$f_{i-1} = f(x_i - h) = f_i - f'(x_i) h + \frac{1}{2} f''(x_i) h^2 + \mathcal{O}(h^3) \quad (2)$$

Beide Gleichungen lassen sich nach der Ableitung umstellen


$$f'(x_i) = \frac{f_{i+1} - f_i}{h} - \frac{1}{2} f''(x_i) h + \mathcal{O}(h^2) \quad (3)$$

$$f'(x_i) = \frac{f_i - f_{i-1}}{h} + \frac{1}{2} f''(x_i) h + \mathcal{O}(h^2) \quad (4)$$

womit die Ableitung für kleine  $h$  durch (bis auf Terme  $\mathcal{O}(h)$ )

$$f'(x_i) \approx \frac{f_{i+1} - f_i}{h} \approx \frac{f_i - f_{i-1}}{h}$$

angenähert wird. Noch besser ist aber die beiden Gleichungen zu addieren:

$$f'(x_i) = \frac{f_{i+1} - f_{i-1}}{2h} + \mathcal{O}(h^2)$$

```
/* Datei: beispiel-3.1.c Datum: 12.4.2016 */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>      /* mathematische Funktionen */
```

```
int main()
{ double x=1.0,h=0.001,f0,f1,f2,dfa,dfb,dfc;
```

```
f0=sin(x-h);
f1=sin(x);
f2=sin(x+h);
```

```
dfa=(f2-f1)/h;
dfb=(f1-f0)/h;
dfc=(f2-f0)/(2*h);
```

```
/* gebe Differenzenquotienten (verschiedene Varianten aus), vergleiche mit exakter Ableitung
wähle Exponentialdarstellung für Ausgabe der Gleitkommazahlen */
```

```
printf("%15.6e %15.6e %15.6e %15.6e \n",dfa,dfb,dfc,cos(x));
```

```
return 0;
}
```

```
/* Ergebnis : numerische Ableitung von sin(x) mit Methode (a),(b) und (c)
```

```
5.398815e-01
```

```
5.407230e-01
```

```
5.403022e-01
```

```
5.403023e-01
```

```
*/
```

## Deklaration und Belegung der Variablen

## Berechne Ableitung von sin(x) mit drei Methoden

## Ausgabe in Exponentialdarstellung

**Dritte Variante mit Fehler in  $h^2$  ist wesentlich genauer!**

Bleibt noch festzulegen, was eine **gute Wahl für h** ist!

Theoretisch sollte  $h \rightarrow 0$

Wegen der endlichen Genauigkeit unserer Gleitkommadarstellung ist das aber nicht ratsam.

Beispiel: Computer mit 5 signifikanten Stellen.

$$h = 10^{-4} \quad \rightarrow \quad f_0 = \sin(0.9999) = 0.84141 \quad f_2 = \sin(1.0001) = 0.84152$$

$$\frac{f_2 - f_0}{2h} = \frac{0.00011}{2 \cdot 10^{-4}} = 0.55000 \neq \cos(1.0) = 0.54030$$

$$h = 10^{-3} \quad \rightarrow \quad f_0 = \sin(0.999) = 0.84093 \quad f_2 = \sin(1.001) = 0.84201$$

$$\frac{f_2 - f_0}{2h} = 0.54000$$

Subtraktion fast gleicher Zahlen, wie in Differenzenquotienten ist oft ungenau.  
Durch die Division, wird der Fehler dann relevant!

Für besseres Ergebnis sollte h deutlich größer als die Genauigkeit der  
Gleitkommadarstellung des Computers sein!

Berücksichtigung der höheren Terme der Taylor-Entwicklung für  $f(x \pm 2h), \dots$  möglich. Meist jedoch nicht nötig.

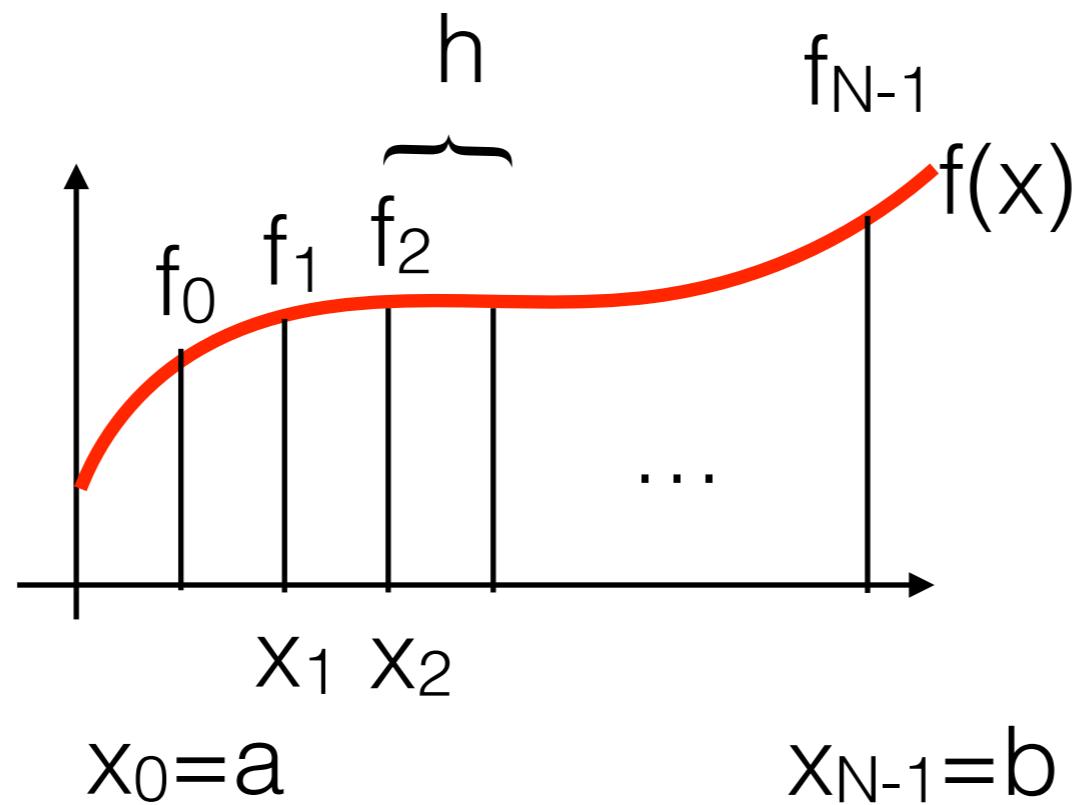
Geschickte Kombination ermöglicht auch eine Berechnung höherer Ableitungen.

Beispielsweise erhält man leicht

$$f''(x_i) = \frac{f_{i+1} + f_{i-1} - 2f_i}{h^2} + \mathcal{O}(h^2)$$

## Numerische Integration

Nutzen wieder, dass unsere Funktion auf einem äquidistanten Gitter gegeben ist



Ziel ist die numerische Bestimmung des Integrals  $\int_a^b dx f(x)$

1) **Versuch:** finde eine interpolierende Polynomfunktion für die  $f(x)$  auf dem Intervall  $[a,b]$ .

→ Polynom  $N-1$  - Grades (Lagrange Formel)

Erfahrung zeigt, dass das Polynom oszilliert und die Funktion schlecht beschreibt.

2) Versuch: approximiere die Funktion lokal durch stückweise Interpolationspolynome niedriger Ordnung.

Typisches Beispiel: **Trapezregel**

$$\int_a^b dx f(x) = \int_{x_0}^{x_1} dx f(x) + \int_{x_1}^{x_2} dx f(x) + \cdots + \int_{x_{N-2}}^{x_{N-1}} dx f(x)$$

Das Problem reduziert sich auf die Teilintegrale  $\int_{x_i}^{x_{i+1}} dx f(x)$

Taylor-Entwicklung für das Intervall ergibt mit numerischer Ableitung

$$f(x) = f_i + (x - x_i) \frac{f_{i+1} - f_i}{h} + \mathcal{O}(h^2)$$

und damit kann man das Integral berechnen

$$\int_{x_i}^{x_{i+1}} dx f(x) = h f_i + \frac{h^2}{2} \frac{f_{i+1} - f_i}{h} + \mathcal{O}(h^3) = \frac{h}{2} (f_{i+1} + f_i) + \mathcal{O}(h^3)$$

Schließlich summiert man über alle Teilintervalle

$$\int_a^b dx f(x) = \frac{h}{2} f_0 + h (f_1 + \dots + f_{N-2}) + \frac{h}{2} f_{N-1} + \underbrace{(N-1) \cdot \mathcal{O}(h^3)}_{\mathcal{O}(h^2)}$$

Generell haben numerische Näherungen von Integralen die Form

$$\int_a^b dx f(x) \approx \sum_{i=0}^{N-1} \omega_i f_i$$

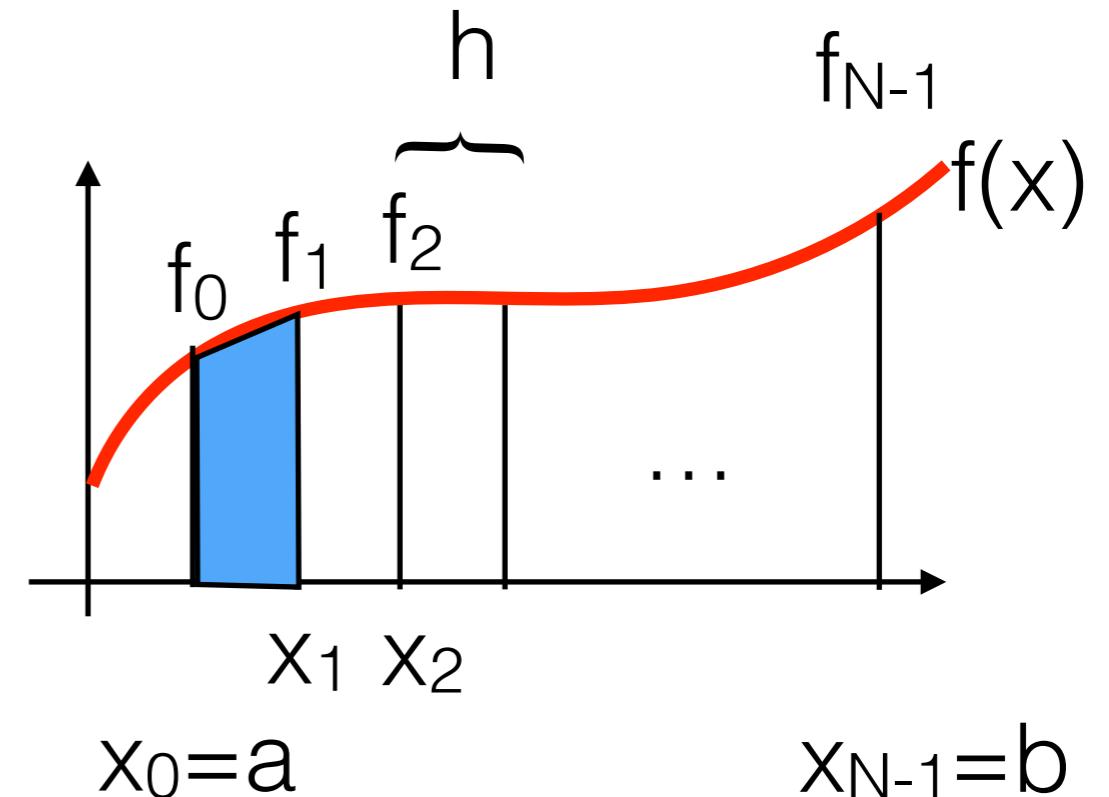
Für die Trapezregel lesen wir ab:

$$h = \frac{b-a}{N-1}$$

$$x_i = a + i \cdot h$$

$$\omega_0 = \omega_{N-1} = \frac{h}{2}$$

$$\omega_i = h \text{ für } i = 1, \dots, N-2$$



```
/* Datei: beispiel-3.2.c      Datum: 12.4.2016 */
```

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

## Definition der zu integrierenden Funktion

```
/* Definition der zu integrierenden Funktion */
double f(double x)
{
    return exp(x);
}
```

## Funktion die Stützstellen und Gewichte festlegt.

```
/* Routine, die Gitterpunkte und Gewichte fuer ein Intervall [a,b] festlegt */
```

```
void trapez(int n, double a, double b, double *xp, double *wp)
```

```
/* n legt die Anzahl der Stuetzstellen fest.
```

```
a,b sind "normale" double Parameter
```

```
xp und wp sind Zeiger(Pointer) auf ein double Feld,
es wird die Adresse des Feldes gespeichert !!! */
```

```
{  
int i;  
double h;
```

## Adresse des Feldes xp und wp ist Parameter!

## n-1 Umwandlung in “double” (type casting)

```
h=(b-a)/(double)(n-1); /* Berechne Schrittweite */
```

## for-Schleife legt “normale” x und w fest

```
for(i=1;i<n-1;i++)
{
    xp[i]=a+i*h; /* xp = Anfangsadresse des Feldes */
    wp[i]=h; /* xp[i] = Nehme die Speicherstelle, */
              /* die i Speicherstellen weiter liegt */
}
```

```
xp[0]=a; /* Lege Punkte und Gewichte am Rand fest */
wp[0]=h/2.0;
xp[n-1]=b;
wp[n-1]=h/2.0;
}
```

## Spezialfall der Randpunkte

## keine Rückgabe, die Felder xp und wp wurden verändert

```

int main()
{
    double a,b;          /* Intervallgrenzen */
    int i,n;             /* Schleifenvariable, Anzahl der Stuetzstellen */
    double exact,diff,sum; /* Variablen, um Ergebnis zu speichern */
    double *x,*w;         /* Zeiger auf Speicherplaetze, d.h. Adressen */

    printf("Bitte geben Sie a,b und n ein: "); /* Eingabe */
    scanf("%lf %lf %d",&a,&b,&n);
}

```

**x und w sind Adresse (=Zeiger) auf Felder**

**Eingabe: a,b und n sollen verändert werden:  
Speicherort=Adresse ist nötig und  
wird mit “&” Operator bestimmt**

```

x=(double *)malloc(n*sizeof(double));
w=(double *)malloc(n*sizeof(double));

```

**malloc reserviert Speicherplatz und gibt Adresse  
dieses Speicherplatzes zurück  
Umwandlung in Zeiger auf double erforderlich**

```

trapez(n,a,b,x,w); /* uebergibt
                      /* Adressen */

```

**Aufruf von Trapez definiert Stützstellen und Gewichte**

```

/* Gitterpunkte und Gewichte sind nun bei x und w gespeichert */
/* Diese kann man fuer beliebige Funktionen benutzen */

```

```

sum=0.0;           /* Bestimmung eines Integrals mit den Gitter und Gewichten */

```

```

for(i=0;i<n;i++)
{
    sum+=f(x[i])*w[i]; /* "+=" Operator summiert f(xi)*w(xi) auf Summe auf */
}

```

```

exact=exp(b)-exp(a);
diff=fabs(sum-exact);

```

```

printf("N      trapez      exact      diff \n\n");
printf("%d      %15.6e      %15.6e      %15.6e \n",n,sum,exact,diff);

```

```

free(x); /* Speicherbereich wieder freigeben */
free(w); /* ("deallozieren") */
}

```

**Bestimmung des Integrals**

**Ausgabe des Ergebnisses, des exakten Ergebnisse  
und des Fehlers**