2 Prozedurale Programmierung

2.1 Der Weg zur Lösung

In der Prozeduralen Programmierung erfolgt die Erstellung von Computerprogrammen unter dem Ansatz, das Gesamtproblem als eine Summe von Teilproblemen und deren Lösung aufzubauen. Ein Programm schreitet bei dieser Sichtweise zur Gesamtlösung von Teillösung zu Teillösung (auch Prozedur genannt, von lat. procedere = voranschreiten). Eine Prozedur zielt dabei bereits auf die Wiederverwendbarkeit bei anderen Problemlösungen. Dieser Ansatz wird später in der modularen Programmierung umgesetzt.

Algorithmen und Lösungsstrukturen

Auf der Suche nach der Lösung von (Teil-)Problemen bedient man sich gewisser Lösungsstrukturen und Prozeduren, auch Algorithmen genannt. Sie sind eine genau definierte Vorschrift zur Lösung eines Problems oder bestimmter Arten von Problemen. Wichtig ist, dass diese Lösung in endlichen Schritten erfolgt.

Black Box/White Box

Am Anfang der Programmierung waren Probleme noch klein und überschaubar. In der Regel ließ sich das Ergebnis eines Programms aus den direkt Eingaben herleiten. Man begnügte sich mit der Black Box/White Box-Betrachtung.

Bild 2-1 Black Box

Bei der Black-Box-Sicht liegen Inputs vor, die nach Umwandlung durch Anweisungen Outputs erzeugen. Die White-Box-Sicht befasst sich ausschließlich mit dem Innenleben der Box.

Top-Down-Design

Mit der Zeit wurden die Problemlösungen immer komplexer und der Schwerpunkt lag auf der Suche nach effizienten Algorithmen. Methoden wie Top-Down-Design halfen bei der Findung von Lösungen komplexer Programmieraufgaben.

Bei dieser Methode wird grafisch ein komplexes Problem in Teilprobleme zerlegt. Sind diese Teilprobleme immer noch komplex, werden sie erneut in weitere Teilprobleme zerlegt. Dieser Vorgang kann so lange durchgeführt werden, bis es für jedes Teilproblem einen Lösungsalgorithmus gibt. Im einfachsten Fall ist dies eine Anweisung. Die Zusammensetzung aller Teillösungen liefert auch die Gesamtlösung. Wen wundert es da, dass diese Teillösungen sich oft in Funktionen wieder fanden. Häufig wiederkehrende Funktionen wurden dann in Standardbibliotheken zusammengefasst, wie zum Beispiel Sortierfunktionen.

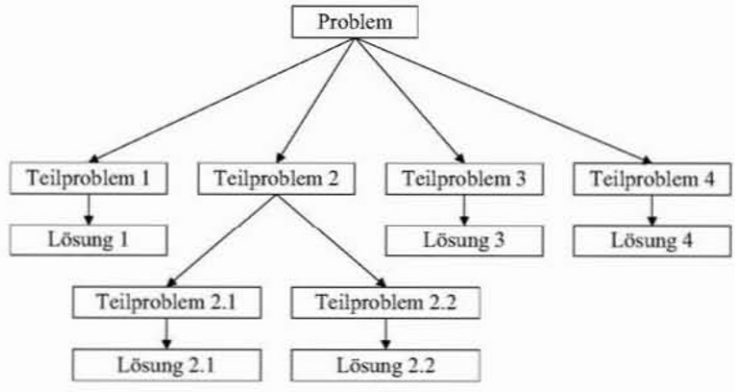


Bild 2-2 Top-Down-Design

Algorithmen

Unter einem Algorithmus versteht man eine genau definierte Handlungsvorschrift zur Lösung eines Problems. Er stellt eine der wichtigsten mathematischen Begriffe dar, vergleichbar etwa mit dem Begriff der Funktion. Algorithmen verfügen über folgende wichtigen Eigenschaften:

- Alle verwendeten Größen müssen bekannt sein
- Die Umarbeitung geschieht in endlichen Arbeitsschritten
- Die Beschreibung des Algorithmus ist vollständig
- Alle angegebenen Operationen sind zulässig

Ein Algorithmus wird in einer (informatischen) Sprache nach bestimmten Regeln definiert. Algorithmen verbinden in ihrer Anwendung oft sehr unterschiedliche Wissensgebiete miteinander. In diesem Buch finden sich Algorithmen aus dem ingenieurwissenschaftlichen Bereich. Sie bedürfen einer besonderen Sichtweise. Anders als der Naturwissenschaftler kann sich der Ingenieur nicht ausgiebig mit allen Randbedingungen eines Problems befassen. Die Zeit, die ihm für ein Projekt zur Verfügung steht, ist ebenso endlich wie der finanzielle Rahmen seiner Projekte. Er muss mit begrenzten Informationen Entscheidungen treffen und dabei Risiken abschätzen. Mit Hilfe von Computersystemen und fachrelevanter Algorithmen kann er sich diese Informationen durch Berechnungen und Simulationen in kürzester Zeit effizient besorgen.

Flussdiagramme und Struktogramme

Auf dem Weg zum eigentlichen Source-Code entwickelten sich noch einige mehr oder weniger grafische Verfahren zur Darstellung von Lösungsalgorithmen in sprachenunabhängiger Form.

Das Flussdiagramm in **Bild 2-3** zeigt die Aneinanderreihung von Anweisungen. Bei der Entwicklung gerät man leicht in Versuchung, nach Abfragen Abläufe zu splitten. Somit gelangen Sprünge in den Quellcode – eine schlechte Qualität der Programmierung.

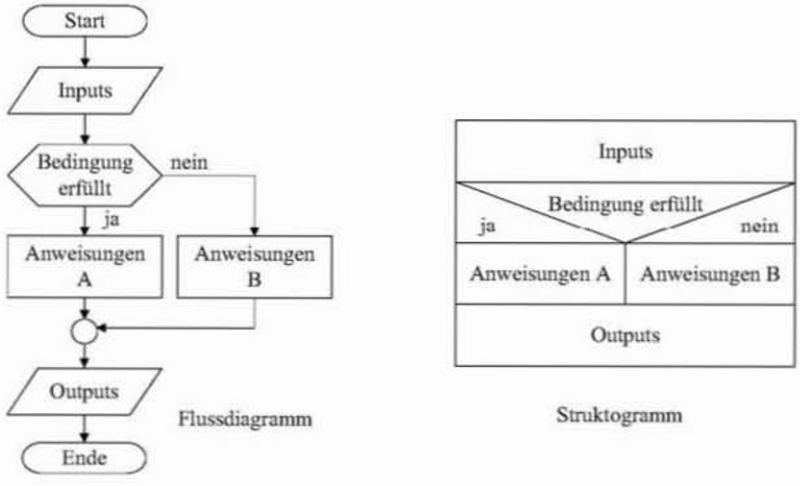


Bild 2-3 Grafische Darstellungen von Algorithmen

Das Nassi-Schneidermann-Diagramm, auch als Struktogramm bezeichnet, verhindert durch seine Grafikelemente solche schlechten Programmiermöglichkeiten. Ein weiterer Vorteil ist, dass sich in dieser grafischen Darstellung deutlich mehr Text unterbringen lässt. Struktogramme sind nach DIN 66261 genormt.

Softwarearchitektur

Mit zunehmender Komplexität der Software entstand der Begriff der Softwarearchitektur und die Aufgabe des Softwarearchitekten sowie damit verbundene neue Entwicklungswerkzeuge. Die Architektur in der Informatik beschreibt die grundlegenden Komponenten und deren Zusammenspiel innerhalb eines Softwaresystems. Die Softwarearchitektur ist Teil des Softwareentwurfs, innerhalb dessen sie entsteht. Während der Softwareentwurf sich detailliert mit der Umsetzung der Vorgaben befasst, zeigt die Softwarearchitektur die globalen Eigenschaften des Gesamtsystems.

Im Rahmen der Softwareentwicklung repräsentiert die Softwarearchitektur mit einem Architekturentwurf die früheste Softwaredesign-Entscheidung. Sie wird im Wesentlichen durch Softwarequalitätskriterien wie Modifizierbarkeit, Sicherheit oder Performance bestimmt. Eine einmal festgelegte Softwarearchitektur ist später nur mit hohem Aufwand abänderbar. Die Entscheidung über ihr Design ist somit eine der kritischsten und wichtigsten Punkte im Entwicklungsprozess einer Software.

Zur grafischen Visualisierung von Softwarearchitekturen werden unterschiedliche Methoden eingesetzt. Beispielsweise:

- Unified Modeling Language (UML)
- Fundamental Modeling Concepts (FMC)

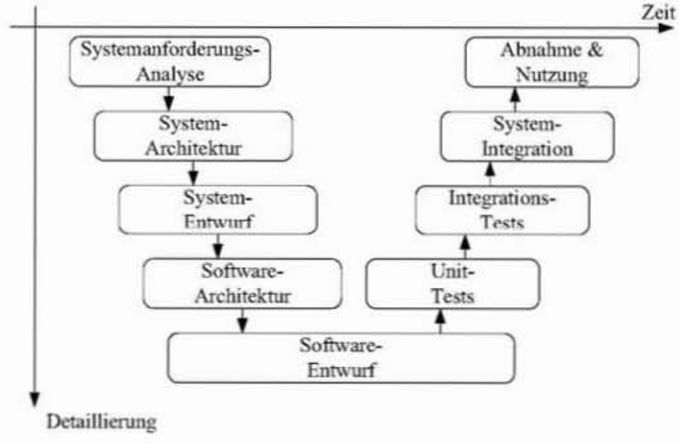


Bild 2-4 Das V-Modell zur Softwareentwicklung (Dröschel, Heuser, Midderhoff, 1998)

Die UML umfasst eine Menge von eng zusammenhängenden Modellierungselementen, mit denen ein Benutzer einen ausgewählten Aspekt eines Systems mit einem bestimmten Formalismus modellieren kann. Unterschieden werden verschiedene Spracheinheiten, die sich in Strukturdiagrammen wie

- Klassendiagramm
- Kompositionsstrukturdiagramm (auch: Montagediagramm)
- Komponentendiagramm
- Verteilungsdiagramm
- Objektdiagramm
- Paketdiagramm

und Verhaltensdiagrammen wie

- Aktivitätsdiagramm
- Anwendungsfalldiagramm (auch Use-Case- oder Nutzfalldiagramm)
- Interaktionsübersichtsdiagramm
- Kommunikationsdiagramm
- Sequenzdiagramm
- Zeitverlaufsdiagramm
- Zustandsdiagramm

darstellen. Später, bei der Objektorientierten Programmierung, wird das Klassendiagramm beschrieben und genutzt. Inzwischen existieren Entwicklungswerkzeuge, die ausgehend von Struktur- und Verhaltensdiagrammen über verschieden Entwicklungsstufen letztlich Quellcode in einer frei wählbaren Programmiersprache erzeugen.

Softwarequalität

Gleichzeitig mit der Prozeduralen Programmierung und deren Verwirklichung in immer größer werdenden Softwareprojekten kamen weitere Begrifflichkeiten (Kriterien) auf den Markt, die sich unter dem Begriff Softwarequalität zusammen fassen lassen. Nach ISO 9126 (DIN 66272) sind die nachfolgend dargestellten sechs Qualitätsmerkmale definiert.

Bild 2-5 Qualitätsmerkmale für Softwareprodukte nach ISO 1926 (DIN 66272)

Ein Kriterium ist die Robustheit einer Software. Darunter versteht man ihre Fähigkeit, auch unter nicht normalen Bedingungen zu funktionieren. So sollte ein Softwareabsturz keine „Katastrophe" auslösen. Der schlimmste Fall liegt vor, wenn eine Software falsche Ergebnisse liefert, denen man ihre „Falschheit" nicht sofort ansieht. Einen gewissen Charme hat die Software, die dem Nutzer eine aussagekräftige Fehlermeldung liefert und ihm die Entscheidung auf Abbruch oder Fortführung unter Bedingungen überlässt.

Ein weiteres Kriterium ist Korrektheit. Eine Software ist dann korrekt, wenn sie ihre Aufgaben (Requirements) gemäß einer Spezifikation exakt erfüllt. Man spricht hier von einer primären Qualität. Eine Software die sowohl robust als auch korrekt ist wird als zuverlässig bezeichnet.

Das Kriterium Änderbarkeit spielt bei der Kostenbetrachtung eine große Rolle, da ca. 65 % der Herstellkosten durch Änderungen entstehen, also durch Erweiterungen und Wartungen. Daraus resultiert wieder eine der häufigsten Fehlerursachen, die Entwicklung unter Zeitdruck. Erweiterungen ergeben sich durch Änderungen der Spezifikation, während Wartungen das Ergebnis gefundener Fehler sind.

Unterstützt wird die Änderbarkeit durch folgende Merkmale:

- Einfachheit des Softwareentwurfs
- Autonome Module mit einfachen Schnittstellen
- Kapselung von Datenstrukturen und damit nur Datenzugriff über Funktionen
- Sinnvolle Dokumentation, am besten im Code

Die Änderbarkeit von Software lässt sich mit herkömmlichen prozeduralen Programmiertechniken oft nur mit hohem Aufwand erreichen.

Ein sehr wichtiges Kriterium ist die Wiederverwendbarkeit. Sie stammt aus der Erfahrung, dass Prozeduren oft nach dem gleichen Muster aufgebaut sind. Eine wiederverwendete Prozedur muss nicht erneut auf eigene Fehler getestet werden und reduziert damit Kosten. Auch Wiederverwendbarkeit lässt sich kaum mit herkömmlichen prozeduralen Programmiertechniken erreichen. Mit diesem Kriterium befasst sich das Kapitel Modulare Programmierung.

Das Kriterium Kompatibilität ist ein Maß für die Verträglichkeit mit anderen Softwaremodulen. Dabei liegt der Fokus meist auf kompatible Datenformate, die sich durch einheitliche Standards erzielen lassen.

Das Kriterium Portabilität trifft eine Aussage über die Verwendung des Softwaremoduls auf verschiedenen Hardware-Umgebungen und Integrationsmöglichkeiten zu verschiedenen Softwareprodukten. Gerade das Portieren kann einen erheblichen Kostenfaktor darstellen.

Das Kriterium Verifizierbarkeit ist ein Maß für die Leichtigkeit, mit der sich Fehlerverfolgung und Fehlerbestimmung besonders in der Testphase umsetzen lassen.

Das Kriterium Integrität betrachtet die Software aus der Sicht der Kapselung von Daten und Prozeduren gegen unberechtigte Zugriffe und Veränderungen (siehe OOP).

Das Kriterium Benutzerfreundlichkeit unterteilt sich in viele Kapitel, so dass hier nur grundlegend die Nutzung von Funktionen, das Bereitstellen von Daten und der Zugriff auf Ergebnisse genannt werden sollen. Im Extremfall stellt das Modul ein grafisches Interface zur Verfügung.

Das Kriterium Performance betrachtet die ökonomische Nutzung von Ressourcen (wie Speicher). Die Entwicklung liefert immer schnellere Prozessoren, optimierende Compiler und immer höhere Bandbreiten.

Alle genannten Kriterien, und es ließen sich noch einige nennen, sind nicht immer direkt miteinander vereinbar. So steht die perfekte Anpassung an eine bestimmte Hardware im Widerspruch zur Portabilität. Doch es lassen sich immer Kompromisse finden, wenn man den Kriterien Prioritäten zuordnet. So entstehen Metriken, die den Entwicklern helfen. Grundsätzlich lässt sich Softwarequalität mit diszipliniertem und methodischem Vorgehen erreichen.

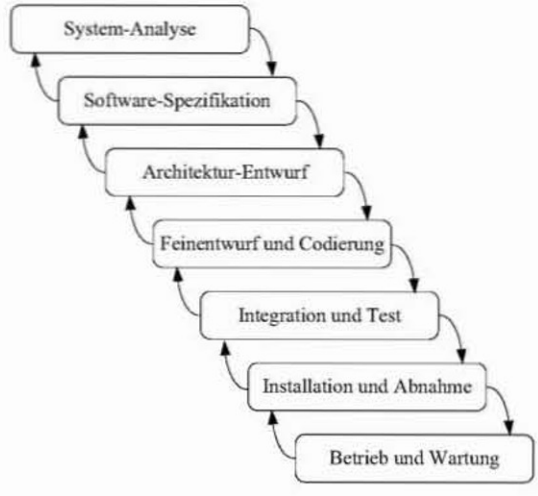


Bild 2-6 Das Wasserfall-Modell, der Ahnherr aller Prozessmodelle (nach Royce, 1970)

Übungen

Erstellen Sie den Algorithmus zur Berechnung der Hypotenuse eines rechtwinkligen Dreiecks nach Pythagoras in grafischer Form mit allen Fehlerabgrenzungen und suchen Sie weitere Algorithmen. Benutzen Sie dazu zuerst die Top-Down-Design-Methode, bevor Sie ein Flussdiagramm oder Struktogramm erstellen.

2.2 Funktionen

Der Begriff der Funktion wird in Informatik und Mathematik mit unterschiedlicher Bedeutung verwendet. In der Mathematik ist eine Funktion eine Abbildung der Funktionsargumente auf das jeweilige Funktionsergebnis. Dabei liefern dieselben Funktionsargumente immer dasselbe Ergebnis.

In der Informatik wird eine Funktion häufig auch als Prozedur bezeichnet. Anders als in der Mathematik brauchen in der Informatik Funktionen weder wertetreu noch frei von einer Wirkung zu sein. Das Ergebnis eines Funktionsaufrufs kann neben den beim Aufruf übergebenen Parametern auch vom augenblicklichen Zustand des verwendeten Speichers abhängen. Ebenso muss die Wechselwirkung zwischen unterschiedlichen Funktionen beachtet werden.

Deklaration und Definition von Funktionen

Jedes C++ Programm startet mit der Abarbeitung der Funktion mit dem Namen main. Alle anderen Funktionen werden dadurch definiert, dass sie nacheinander aufgeführt werden. Funktionen, die aufgerufen werden, müssen vorher definiert sein. Die Definition von Funktionen in Funktionen ist nicht erlaubt.

Code 2-1 Allgemeiner Programmaufbau

```
//Einbinden von Headerdateien
#include HeaderDateiName1
#include HeaderDateiName2
...

//Konstantenvereinbarungen
#define Name1 Value1
#define Name2 Value2
...

//Definition des Namensraums
using namespace std;

//Deklarationen von Funktionen als Prototypen
DatenTyp Funktionsname1 (Parameterliste1);
DatenTyp Funktionsname2 (Parameterliste2);
...

//Hauptfunktion
DatenTyp main (Parameterliste)
{
   ...
}

//Definitionen von Funktionen
//Da die Funktionen bereits oberhalb von main als Prototypen
//definiert sind, können sie auch nach main deklariert werden.

DatenTyp Funktionsname1 (Parameterliste1)
{
   ...
}
```

```
DatenTyp Funktionsname2 (Parameterliste2)
{
   ...
}
...
```

Prototypen

Bevor eine Funktion verwendet werden kann, muss dem Compiler zunächst mitgeteilt werden, dass es eine Funktion dieses Namens gibt, wie viele und welche Parameter sie hat und welchen Typ sie zurückliefert. Dies geschieht mit einem so genannten Prototyp der Funktion. Der Prototyp sieht genauso aus wie die Funktion selbst bis auf den Funktionskörper; dieser fehlt und wird durch ein einfaches Semikolon (;) ersetzt. Es ist sogar erlaubt, die Namen der Argumente wegzulassen und nur ihre Typen anzugeben. Analog zu den Variablen stellt der Prototyp die Deklaration der Funktion und die Funktion selbst mit ihrem Anweisungsblock die Definition dar.

Syntax: Prototyp einer Funktion

```
DatenTyp Funktionsname (Parameterliste);
```

Funktionsaufbau

Eine Funktion hat den nachfolgend dargestellten allgemeinen Aufbau.

Syntax: Allgemeiner Aufbau einer Funktion

```
DatenTyp Funktionsname (Parameterliste)
{
   Anweisungsblock
   return [Datentyp [Rückgabewert]];
}
```

Sie besitzt einen Datentyp, einen Namen, eine Parameterliste, einen Anweisungsblock und einen Rückgabewert.

Funktionen ohne Übergabe-Parameter

Funktionen können ohne Parameterangaben aufgerufen werden. Dann muss in der Parameterliste der fundamentale Datentyp void stehen.

Syntax: Prozedur - Funktion ohne Übergabeparameter

```
DatenTyp Funktionsname (void)
{
   Anweisungsblock
   return [Rückgabewert];
}
```

Funktionen ohne Rückgabewert

Ist der Datentyp der Funktion void, dann hat die Funktion keinen Rückgabeparameter und die return-Anweisung entfällt. Diese Form der Funktion wird auch als Prozedur bezeichnet.

Syntax: Prozedur - Funktion ohne Rückgabewert

```
void Funktionsname (Parameterliste)
{
   Anweisungsblock
}
```

Die return-Anweisung in Funktionen

Eine Funktion endet spätestens mit der Ausführung der letzten Anweisung und der Programmablauf liegt wieder bei der rufenden Funktion. Ohne Angabe eines Rückgabewertes ist dieser je nach Datentyp der Funktion automatisch auf das Nullelement des Datentyps gesetzt.

Mit der return-Anweisung wird ein vorzeitiges Ende der Funktion erreicht und der bei dieser Anweisung angegebene Wert wird als Rückgabewert der Funktion der rufenden Funktion übergeben.

Überladen von Funktionen

Es gibt den Fall, dass eine Funktion konzeptionell die gleichen Aufgaben mit unterschiedlichen Datentypen ausführen muss. Die Nutzung desselben Namens für gleiche Operationen auf unterschiedlichen Datentypen wird als Überladen bezeichnet. Bekannt ist diese Technik bereits aus den Grundoperationen. Zum Beispiel wird eine Addition (+) für unterschiedliche Datentypen ausgeführt.

Beispiel: Überladen von Funktionen

```
#include <iostream.h>

int Max (int a, int b)
   {return (a>=b)? a : b; }
double Max (double a, double b)
   {return (a>=b)? a : b; }

int main()
{
   cout << "Der groessere Wert von 12 und 13 ist: ";
   cout << Max (12, 13) << endl;
   cout << "Der groessere Wert von 12,2 und 12,1 ist: ";
   cout << Max (12.2, 12.1) << endl;

   system("Pause");
}
```

Inline-Funktionen

Eine einfache Funktion, wie oben definiert, kann auch *inline* definiert werden. Dies ist eine Aufforderung an den Compiler, den Code der Funktion an der Stelle im Code *online* zu generieren, an der ihr Aufruf steht, statt sie durch übliche Aufrufmechanismen nutzbar zu machen.

Beispiel: Inline-Funktion

```
Inline int Max (int a, int b)
   {return (a>=b)? a : b; }
```

Der Vorteil dieser effizienteren Codeerzeugung wird hinfällig, wenn die Funktion an vielen Stellen aufgerufen wird.

Rekursionen

Rekursive Algorithmen lassen sich oft durch rekursive Prozeduren elegant lösen. Eine rekursive Prozedur ist in der Lage, sich selbst aufzurufen. Dabei wird oft die Lösung eines (n)-Problems auf ein (n-1)-Problem zurückgeführt.

Beispiel: Rekursive Bestimmung von n-Fakultät

```
#include <iostream.h>

long int Fakul (long int n)
{
   long int fak;
   if (n == 1)
      fak = 1;
   else
      fak = n * Fakul(n-1);
   return fak;
}

int main()
{
   cout << "Der Wert von 7! ist: ";
   cout << Fakul(7) << endl;
   system("Pause");
}
```

Vorgabe-Argumente

Beim Entwurf von Funktionen steht man oft vor der Wahl, ob die Funktion unkompliziert in der Anwendung oder universell verwendbar sein soll.

Beispiel: Funktion zum Zeichnen von Ellipsen

```
void ellipse (int x, int y,
              int stangle, int endangle,
              int xradius, int yradius);
```

Die ersten beiden Parameter geben den Mittelpunkt der Ellipse an, die letzen beiden Parameter die beiden Halbachsen. Die Parameter stangle und endangle erlauben, einen Teil der Ellipse zu zeichnen: Sie geben den Anfangs- und Endwinkel (in Altgrad) an. Für den Normalfall einer vollständigen Ellipse müssen hier stets die Werte 0 und 360 eingesetzt werden. Die Funktion wäre leichter zu merken, wenn man diese beiden Werte weglassen könnte. Deshalb ist es in C++ erlaubt, bei der Deklaration einer Funktion Argumenten Vorgabewerte zu geben, die automatisch verwendet werden, wenn die Argumente weggelassen werden. Damit der Compiler eindeutig erkennen kann, welche Parameter weggelassen wurden, gilt die Einschränkung, dass solche Vorgabe-Argumente am Ende der Parameterliste stehen müssen. Die Vorgabeargumente dürfen nur bei der ersten Deklaration angegeben werden!

Als Beispiel folgt eine Funktion (mit dem doppeldeutigen Namen Ellipse), bei der die Winkelargumente weggelassen werden können.

Beispiel: Funktion zum Zeichnen von Ellipsen mit Vorgabewerten für Start- und Endwinkel

```
void Ellipse (int x,int y,int a,int b,int w0=0,int w1=360)
{
   ellipse (x, y, w0, w1, a, b);
}
```

Beispiel: Aufrufe der Funktion zum Zeichnen von Ellipsen

```
Ellipse (40, 40, 30, 20);
Ellipse (80, 40, 30, 20, 0, 90);
```

Durch die beiden Aufrufe werden eine volle und eine viertel Ellipse gezeichnet.

Anwendungsbeispiel: Partielle Differentialgleichungen – Membranfläche

In gewöhnlichen Differentialgleichungen treten nur Funktionen mit einer unabhängig Veränderlichen auf. Dagegen spricht man von einer partiellen Differentialgleichung, wenn die gesuchte Funktion

$$y = y(x_1, x_2, ..., x_n) \tag{2.1}$$

von mehreren Veränderlichen x_1, x_2, ... , x_n abhängt und in der Gleichung partielle Ableitungen der Form

$$\frac{\partial y}{\partial x_i}, \frac{\partial^2 y}{\partial x_i \partial x_j}, \text{ usw.} \tag{2.2}$$

auftreten. Um deren Lösung numerisch zu bestimmen, überzieht man die x, y-Ebene mit einem zweidimensionalen Gitter der Maschenweite h.

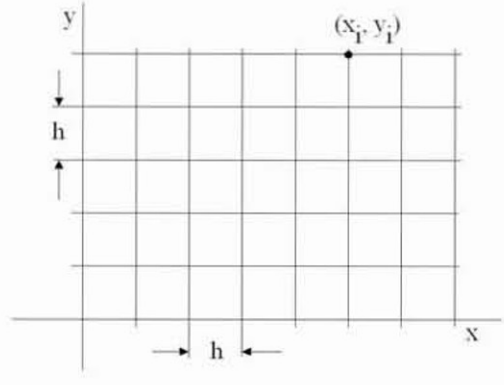


Bild 2-7 Gitterpunkte in der Ebene

Die Gitterpunkte bestimmen sich durch

$$x_i = x_0 + i \cdot h \quad \text{und} \quad y_j = y_0 + j \cdot h \,. \tag{2.3}$$

Außerdem wollen wir folgende Abkürzung

$$u_{i,j} = u(x_i, y_j) \tag{2.4}$$

verwenden

Ähnlich wie zuvor werden auch hier partielle Ableitungen erster und höherer Ordnung durch Differenzenquotienten approximiert (diskretisiert). So ergibt sich

$$\frac{\partial u}{\partial x}(x_i, y_j) = \frac{u_{i+1,j} - u_{i-1,j}}{2h} + O(h^2) \tag{2.5}$$

und

$$\frac{\partial u}{\partial y}(x_i, y_j) = \frac{u_{i,j+1} - u_{i,j-1}}{2h} + O(h^2)\,. \tag{2.6}$$

Ein häufig auftretender Differentialoperator ist der Laplace-Operator Δ

$$\Delta u := \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \tag{2.7}$$

mit der Differenzenapproximation

$$\Delta u(x_i, y_j) \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}\,. \tag{2.8}$$

Dies lässt sich symbolisch und anschaulich durch den in **Bild 2-8** dargestellten Berechnungsoperator umsetzen.

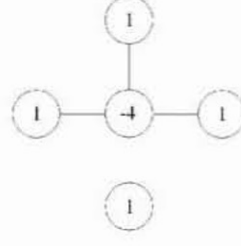


Bild 2-8 Berechnungsoperator

Als konkretes Beispiel wird die nachfolgend dargestellte elastische Membran verwendet.

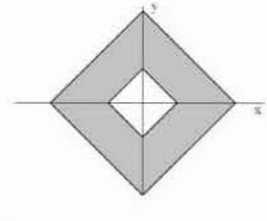


Bild 2-9 Form der technischen Membrane

Sie ist an den Rändern fest eingespannt und erfüllt die Laplacesche Differentialgleichung

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=0\,. \tag{2.9}$$

Dabei ist u die Höhe der Membrane über der (x,y)-Ebene. Die Randwerte seien u = 0 für den äußeren und u = 1 für den inneren Rand. Aus der Symmetrieeigenschaft des Laplace-Operators genügt die Betrachtung eines Viertelstücks.

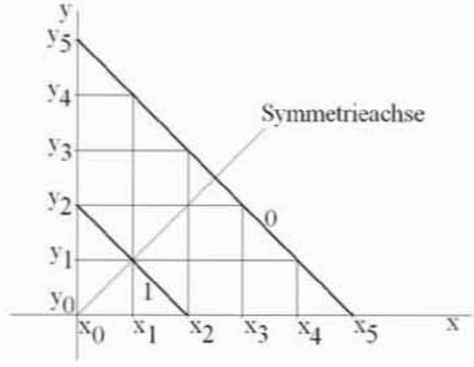


Bild 2-10 Membranausschnitt

Durch Anwendung des Operators ergeben sich nachfolgende Differenzen.

Verbesserung der Zeilenwerte (von Schritt n nach n+1):

j=0, i=3:

$$-u_{2,0}^{(n+1)}+4u_{3,0}^{(n+1)}-u_{4,0}^{(n+1)}=u_{3,-1}^{(n)}+u_{3,1}^{(n)}$$

daraus folgt:

$$4u_{3,0}^{(n+1)}-u_{4,0}^{(n+1)}=2u_{3,1}^{(n)}+1 \text{ weil } u_{2,0}^{(n+1)}=1.$$

j=0, i=4:

$$-u_{3,0}^{(n+1)}+4u_{4,0}^{(n+1)}-u_{5,0}^{(n+1)}=u_{4,-1}^{(n)}+u_{4,1}^{(n)}$$

daraus folgt:

$$-u_{3,0}^{(n+1)}+4u_{4,0}^{(n+1)}=0 \text{ weil } u_{5,0}^{(n+1)}=0.$$

j=1, i=2:

$$-u_{1,1}^{(n+1)}+4u_{2,1}^{(n+1)}-u_{3,1}^{(n+1)}=u_{2,0}^{(n)}+u_{2,2}^{(n)}$$

daraus folgt:

$$4u_{2,1}^{(n+1)}-u_{3,1}^{(n+1)}=u_{2,2}^{(n)}+2 \text{ weil } u_{1,1}^{(n+1)}=u_{2,0}^{(n)}=1.$$

j=1, i=3:

$$-u_{2,1}^{(n+1)}+4u_{3,1}^{(n+1)}-u_{4,1}^{(n+1)}=u_{3,0}^{(n)}+u_{3,2}^{(n)}$$

daraus folgt:

$$-u_{2,1}^{(n+1)}+4u_{3,1}^{(n+1)}=u_{3,0}^{(n)} \text{ weil } u_{4,1}^{(n+1)}=u_{3,2}^{(n)}=0.$$

j=2, i=2:

$$-u_{1,2}^{(n+1)}+4u_{2,2}^{(n+1)}-u_{3,2}^{(n+1)}=u_{2,1}^{(n)}+u_{2,3}^{(n)}$$

daraus folgt:

$$-u_{2,1}^{(n+1)}+4u_{2,2}^{(n+1)}=u_{2,1}^{(n)}$$

weil $u_{3,2}^{(n+1)}=u_{2,3}^{(n)}=0$ und $u_{1,2}^{(n+1)}=u_{2,1}^{(n+1)}$ (sym.).

Verbesserung der Spaltenwerte (von Schritt n+1 nach n+2):

i=2, j=1:

$$-u_{2,0}^{(n+2)}+4u_{2,1}^{(n+2)}-u_{2,2}^{(n+2)}=u_{1,1}^{(n+1)}+u_{3,1}^{(n+1)}$$

daraus folgt:

$$4u_{2,1}^{(n+2)}-u_{2,2}^{(n+2)}=u_{3,1}^{(n+1)}+2 \text{ weil } u_{2,0}=u_{1,1}=1.$$

i=2, j=2:

$$-u_{2,1}^{(n+2)}+4u_{2,2}^{(n+2)}-u_{2,3}^{(n+2)}=u_{1,2}^{(n+1)}+u_{3,2}^{(n+1)}$$

daraus folgt:

$$-u_{2,1}^{(n+2)}+4u_{2,2}^{(n+2)}=u_{2,1}^{(n+1)}$$

weil $u_{2,3}=u_{3,2}=0$ und $u_{1,2}=u_{2,1}$ (sym.).

i=3, j=0:

$$-u_{3,-1}^{(n+2)}+4u_{3,0}^{(n+2)}-u_{3,1}^{(n+2)}=u_{2,0}^{(n+1)}+u_{4,0}^{(n+1)}$$

daraus folgt:

$$4u_{3,0}^{(n+2)}-2u_{3,1}^{(n+2)}=u_{4,0}^{(n+1)}+1$$

weil $u_{2,0}=1$und $u_{3,-1}=u_{3,1}$ (sym.).

i=3, j=1:

$$-u_{3,0}^{(n+2)}+4u_{3,1}^{(n+2)}-u_{3,2}^{(n+2)}=u_{2,1}^{(n+1)}+u_{4,1}^{(n+1)}$$

daraus folgt:

$$-u_{3,0}^{(n+2)}+4u_{3,1}^{(n+2)}=u_{2,1}^{(n+1)} \text{ weil } u_{3,2}=u_{4,1}=0.$$

i=4, j=0:

$$-u_{4,-1}^{(n+2)}+4u_{4,0}^{(n+2)}-u_{4,1}^{(n+2)}=u_{3,0}^{(n+1)}+u_{5,0}^{(n+1)}$$

daraus folgt:

$$4u_{4,0}^{(n+2)}=u_{3,0}^{(n+1)}$$

weil $u_{4,1}=u_{5,0}=0$ und $u_{4,-1}=u_{4,1}$ (sym.).

Durch Umstellung ergeben sich die 10 Iterationsgleichungen:

$$u_{4,0}^{(n+1)}=\frac{1}{15}(2u_{3,1}^{(n)}+1)$$

$$u_{3,0}^{(n+1)} = 4u_{4,0}^{(n)}$$

$$u_{3,1}^{(n+1)} = \frac{1}{15}(u_{2,2}^{(n)} + 4u_{2,2}^{(n)} + 1)$$

$$u_{2,1}^{(n+1)} = 4u_{3,1}^{(n+1)} - u_{3,0}^{(n)}$$

$$u_{2,2}^{(n+1)} = \frac{1}{4}(u_{2,1}^{(n+1)} + u_{2,1}^{(n)})$$

$$u_{2,2}^{(n+2)} = \frac{1}{15}(4u_{2,1}^{(n+1)} + u_{3,1}^{(n+1)} + 2)$$

$$u_{2,1}^{(n+2)} = 4u_{2,2}^{(n+1)} - u_{2,1}^{(n+1)}$$

$$u_{3,1}^{(n+2)} = \frac{1}{14}(4u_{2,1}^{(n+1)} + u_{4,0}^{(n+1)} + 1)$$

$$u_{3,0}^{(n+2)} = 4u_{3,1}^{(n+2)} - u_{2,1}^{(n+1)}$$

$$u_{4,0}^{(n+2)} = \frac{1}{4}u_{3,0}^{(n+1)}$$

Tabelle 2-1 Struktogramm – Differenzenapproximation

Eingabe Startwerte $u_{4,0}^{(0)} = 0; u_{3,0}^{(0)} = 0; u_{3,1}^{(0)} = 0; u_{2,1}^{(0)} = 0; u_{2,2}^{(0)} = 0$
n Iterationsschritte
$u_{4,0}^{(n+1)} = \frac{1}{15}(2u_{3,1}^{(n)} + 1)$
$u_{3,0}^{(n+1)} = 4u_{4,0}^{(n+1)}$
$u_{3,1}^{(n+1)} = \frac{1}{15}(u_{2,2}^{(n)} + 4u_{3,0}^{(n)} + 2)$
$u_{2,1}^{(n+1)} = 4u_{3,1}^{(n+1)} - u_{3,0}^{(n)}$
$u_{2,2}^{(n+1)} = \frac{1}{4}(u_{2,1}^{(n+1)} + u_{2,1}^{(n)})$
$u_{2,2}^{(n+2)} = \frac{1}{15}(4u_{2,1}^{(n+1)} + u_{3,1}^{(n+1)} + 2)$
$u_{2,1}^{(n+2)} = 4u_{2,2}^{(n+2)} - u_{2,1}^{(n+1)}$

	$u_{3,1}^{(n+2)} = \frac{1}{14}(4u_{2,1}^{(n+1)} + u_{4,0}^{(n+1)} + 1)$
	$u_{3,0}^{(n+2)} = 4u_{3,1}^{(n+2)} - u_{2,1}^{(n+1)}$
	$u_{4,0}^{(n+2)} = \frac{1}{4}u_{3,0}^{(n+1)}$

Das Programm enthält in der Reihenfolge die Iterationsgleichungen. Es ist jedoch darauf zu achten, welcher Iterationswert in die jeweilige Gleichung eingeht. Ich habe dies durch die Variablenfolge u, v, w gekennzeichnet. Erst nach Beendigung der beiden Iterationsschritte erfolgt eine Verschiebung und ein neuer Durchlauf kann beginnen.

Code 2-2 Anwendungsbeispiel Partielle Differentialgleichung – Membrane

```
/* Membrane.cpp
*/

#include <iostream.h>

// Iterationen
float f40 (float u31) {return 1/15*(2*u31+1);}
float f30 (float v40) {return 4*v40;}
float f31 (float u22, float u30) {return 1/15*(u22+4*u30+2);}
float f21 (float v31, float u30) {return 4*v31-u30;}
float f22 (float v21, float u21) {return 1/4*(v21+u21);}

float g22 (float v21, float v31) {return 1/15*(4*v21+v31+2);}
float g21 (float w22, float v21) {return 4*w22-v21;}
float g31 (float v21, float v40) {return 1/14*(4*v21+v40+1);}
float g40 (float v30) {return 1/4*v30;}

int main ()
{
   int i,k;
   float data[11][11];
   float u[5];
   float v[5];
   float w40,w30,w31,w21,w22;
   FILE *puffer;
   puffer = fopen("C:\\Temp\\Membran.dat","w");

// Startwerte
   for (i=0; i<11; i++)
      for (k=0; k<11; k++)
         data [i][k] = 0;
   data [3][5] = 1;
   data [4][4] = 1;
   data [4][6] = 1;
   data [5][3] = 1;
   data [5][7] = 1;
   data [6][4] = 1;
```

```
   data [6][6] = 1;
   data [7][5] = 1;

// Eingabe und Anzeige
   cout << endl << "Eingabe:"  << endl;
   for (i=0; i<11; i++)
   {
      for (k=0; k<11; k++)
         cout << data[i][k] << " ";
      cout << endl;
   }
   cout << endl;

// Auswertung
   u[0] = data[5][9];
   u[1] = data[5][8];
   u[2] = data[4][8];
   u[3] = data[4][7];
   u[4] = data[3][7];

   for (i=0; i<100; i++)
   {
      v[0] = f40(u[2]);
      v[1] = f30(v[0]);
      v[2] = f31(u[4], u[1]);
      v[3] = f21(v[2], u[1]);
      v[4] = f22(v[3], u[3]);

      w22 = g22(v[3], v[2]);
      w21 = g21(w22, v[3]);
      w31 = g31(v[3], v[0]);
      w40 = g40(v[1]);

      u[0] = w40;
      u[1] = w30;
      u[2] = w31;
      u[3] = w21;
      u[4] = w22;

      data [5][9] = u[0];
      data [5][8] = u[1];
      data [4][8] = u[2];
      data [4][7] = u[3];
      data [3][7] = u[4];
   }

// Übertragung (Symmetrie)
   data [6][8] = data [4][8];
   data [6][7] = data [4][7];
   data [7][7] = data [3][7];
   data [7][6] = data [3][6];
   data [8][6] = data [2][6];
   data [8][5] = data [2][5];
   data [9][5] = data [1][5];
   data [1][5] = data [5][9];
```

```
    data [2][5] = data [5][8];
    data [2][6] = data [4][8];
    data [3][6] = data [4][7];
    data [2][4] = data [2][6];
    data [3][4] = data [3][6];
    data [7][4] = data [7][6];
    data [8][4] = data [8][6];
    data [3][3] = data [3][7];
    data [4][3] = data [4][7];
    data [6][3] = data [6][7];
    data [7][3] = data [7][7];
    data [4][2] = data [4][8];
    data [5][2] = data [5][8];
    data [6][2] = data [6][8];
    data [5][1] = data [5][9];

// Ausgabe
    for (i=0; i<11; i++)
    {
       for (k=0; k<11; k++)
       {
          printf("\t%2.2e", data [i][k]);
          fprintf(puffer, "\t%2.2e", data [i][k]);
       }
       printf ("\n ");
       fprintf (puffer, "\n ");
    }
    cout << endl;
    fclose(puffer);
    system("Pause");
}
```

Zunächst werden die Positionspunkte der Viertel-Membran berechnet und danach entsprechend der Symmetrie übertragen. Die Daten werden in eine Datei geschrieben und können so nachfolgend ausgewertet werden. Ich benutze dazu gerne das Excel Programm von Microsoft.

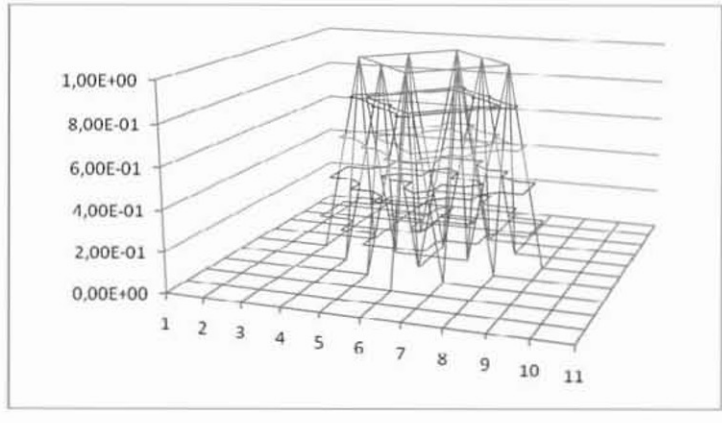


Bild 2-11 Membranverformung

Übungen

Eine Elektronenröhre hat den nachfolgend dargestellten Aufbau. Wir gehen von der vereinfachten Annahme aus, dass die Elektroden nach links und rechts unendlich fortgeführt sind. Dann genügt die Potentialverteilung im Raum zwischen den Elektroden der Laplaceschen Differentialgleichung.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \,. \tag{2.9}$$

Die Kathode hat das Potential 0 Volt und das Gitter -1 Volt. Wie groß muss die Anodenspannung gewählt werden, damit das Potential zwischen den Gitterfäden nicht negativ wird? Benutzen Sie die dargestellte Aufteilung unter Berücksichtigung symmetrischer Verhältnisse.

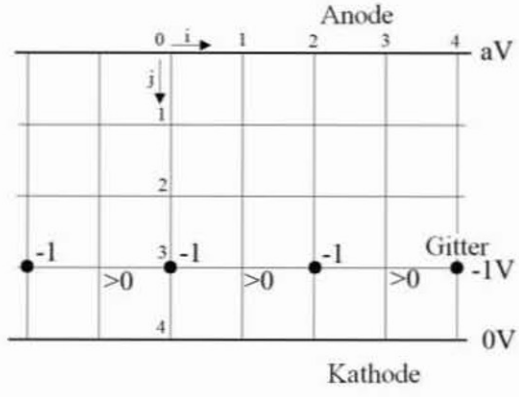


Bild 2-12 Schematischer Aufbau einer Elektronenröhre

Ermitteln Sie durch die Anwendung des Operators die sich daraus ergebenden Differenzen. Durch Umstellen und Einsetzen ergeben sich daraus Iterationsgleichungen. In diesen Gleichungen wird die Anodenpannung vorgegeben und nachgeprüft, ob die Gitterspannung größer null ist.

2.3 Parameter

Parameterübergabe
Namen von Funktionsargumenten werden als im äußeren Block einer Funktion deklariert betrachtet.

Beispiel: Parameterübergabe

```
void f(int x)
{
   int x; //FEHLER!
}
```

Mit dem Aufruf einer Funktion werden in der Regel auch Parameter übergeben. Dabei wird ein zusätzlicher Speicher (Funktionsstack) erzeugt, in den Kopien der aktuellen Parameter angelegt werden. Nach Verlassen der Funktion wird der Stack wieder gelöscht (Ausnahme statische Variable).

Das Ändern einer globalen Variablen aus einer Funktion heraus wird als Seiteneffekt bezeichnet und kann unter Umständen zu verwirrenden Ergebnissen führen.

Code 2-3 Seiteneffekt

```
/* Seiteneffekt.cpp
*/

#include <iostream.h>

int x; //global

int main ()
{
    extern int f(int y);

    x=9;
    cout << (f(9)*f(x)) << endl;
    x=9;
    cout << (f(x)*f(9)) << endl;

    system("Pause");
}

int f(int y)
{
   x-=9;
   return y*y+1;
}
```

Dieses Beispiel liefert die Ausgabe:

```
82
6724
```

Parameter lassen sich in C++ per value, per pointer oder per reference an Funktionen übergeben. Die Entscheidung, wie ein Parameter übergeben wird, wird für jeden einzelnen in der Parameterliste getroffen: Die Übergabe als Wert durch die Nennung der Variablen, die Übergabe als Zeiger durch den Adressoperator zur Variablen oder die Übergabe als Referenz durch den Referenzoperator zur Variablen.

Call by value

Wird ein Funktionsparameter als Wert (per value) an eine Funktion übergeben, dann wird eine Kopie der Variablen im Speicher (Stack) angelegt. Innerhalb der Funktion kann der Inhalt der Kopie beliebig verändert werden. Der Programmierer kann sicher sein, dass mit Beendigung der Funktion der Inhalt des Originals nicht verändert wurde.

Code 2-4 Vertauschung der Inhalte zweier int-Variablen mittels *call by value*

```
/* CallByValue.cpp
*/

#include <iostream.h>

void swap(int a,int b)
{
   int tmp=a;
   a=b;
   b=tmp;
};

int main()
{
   int a=1;
   int b=9;

   swap(a,b);
   cout << "a: " << a << endl;
   cout << "b: " << b << endl;

   system("Pause");
};
```

Das Programm liefert:

```
a: 1
b: 9
```

Die Werte werden also nicht vertauscht. Das liegt daran, dass mit dem Aufruf der Funktion *swap* die Werte der Variablen in entsprechend lokale Variable der Funktion kopiert werden. Die Vertauschung findet nur unter den lokalen Variablen der Funktion statt. Beim Verlassen der Funktion werden die lokalen Variablen a, b und tmp verworfen.

Call by pointer

Das im vorherigen Fall durchaus gewünschte Verhalten führt dann zu Problemen, wenn die Vertauschung auch für die rufende Funktion gelten soll, denn die gerufene Funktion kann

höchstens nur einen Rückgabewert (mittels return) liefern. Die typische Lösung ist, dass nicht die Werte an sich, sondern nur deren Adressen an die Funktion übergeben werden.

Code 2-5 Vertauschung der Inhalte zweier int-Variablen mittels *call by pointer*

```
/* CallByPointer.cpp
*/

#include <iostream.h>

void swap(int *a,int *b)
{
   int tmp=*a;
   *a=*b;
   *b=tmp;
};

int main()
{
   int a=1;
   int b=9;

   swap(&a,&b);
   cout << "a: " << a << endl;
   cout << "b: " << b << endl;

   system("Pause");
};
```

Das Programm liefert nun auch das gewünschte Ergebnis:

```
a: 9
b: 1
```

Call by reference

C++ kennt noch eine weitere Form der Parameterübergabe, die Übergabe per *reference*. Dies wird dadurch gekennzeichnet, dass dem Funktionsparameter ein & vorangestellt wird. In diesem Zusammenhang ist der Operator nicht mit dem Adressoperator & zu verwechseln. Das Testbeispiel bekommt dadurch die nachstehende Form.

Code 2-6 Vertauschung der Inhalte zweier int-Variablen mittels *call by reference*

```
/* CallByReference.cpp
*/

#include <iostream.h>

void swap(int &a,int &b)
{
   int tmp=a;
   a=b;
   b=tmp;
};
```

```
int main()
{
   int a=1;
   int b=9;

   swap(a,b);
   cout << "a: " << a << endl;
   cout << "b: " << b << endl;

   system("Pause");
};
```

Und auch dieses Programm liefert das gewünschte Ergebnis:

```
a: 9
b: 1
```

Die Parameterübergabe *call by reference* ist gegenüber der Parameterübergabe *call by pointer* besser lesbar und daher vorzuziehen. Das trifft auch dann zu, wenn es sich bei den Übergabeparametern um Strukturen und Objekte handelt. Dazu später mehr.

Nach diesen Betrachtungen wenden sich oft Lernwillige C++-Anfänger mit Grausen ab. Für die Unerschütterlichen folgt ein Anwendungsbeispiel.

Anwendungsbeispiel: call by pointer – Sortieralgorithmus Bubblesort

Dieses Beispiel zeigt mit dem einfachen Sortieralgorithmus (Bubblesort) eine sinnvolle Anwendung von Funktionszeigern. Die Funktion kann beliebig große eindimensionale Felder des Datentyps int sortieren.

Code 2-7 Anwendungsbeispiel Funktionszeiger – Sortieralgorithmus Bubblesort

```
/* BubbleSort.cpp
*/

#include <iostream.h>

static void Tausche (int *p1, int *p2)
{
   int t;
   t = *p1, *p1 = *p2, *p2=t;
}

static void BubbleSort (
   int *Vektor,     // Adresse des Sortierbereichs
   int Anzahl)      // Anzahl der Elemente
{
   if (Anzahl) {
      for (int i=0;i<Anzahl-1;i++) {
         for (int j=i+1;j<Anzahl;j++) {
            if (Vektor[i]>Vektor[j]) {
               Tausche (Vektor+i,Vektor+j);
```

```
            }
        }
    }
}

int main ()
{
   int i;
   int data[] = {3, 1, 9, 2, 5, 8, 7, 6, 4};

   cout << endl << "unsortiert:"  << endl;
   for (i=0; i<sizeof(data)/sizeof(int); i++) cout << data[i] << "
";
   cout << endl;

   BubbleSort (data, sizeof (data)/sizeof(int));

   cout << "sortiert:" << endl;
   for (i=0; i<sizeof(data)/sizeof(int); i++) cout << data[i] << "
";
   cout << endl;

   system("Pause");
}
```

Übungen

Der Bubblesort-Algorithmus arbeitet bei größeren Datenmengen ziemlich uneffektiv. In der Standard-Bibliothek *StdLib.h* gibt es auch eine Quicksort-Implementierung *qsort*. Wie müsste das Programm für das Sortieren von Werten eines beliebigen Datentyps aussehen? Wie sieht das Programm mit *call by reference* aus?

Felder als Parameter

Die Übergabe von Feldern als Parameter an Funktionen kann auf verschiedene Art und Weise erfolgen. Dazu ein Beispiel, bei dem eine feste Anzahl Felder übergeben wird.

Beispiel: Übergabemöglichkeiten eines Feldes als Parameter

```
void fa(int a[4]);
void fb(int a[]);
void fc(int *a);
```

Die Funktion fa im Beispiel erwartet 4 Integer-Werte. Dabei wird nicht das Feld kopiert, sondern es wird lediglich die Adresse des Feldes übergeben. Folglich sind Änderungen direkt auch in der aufrufenden Funktion wirksam. Die Funktion fb prüft nicht einmal, ob die 4 erwarteten Werte auch vorhanden sind. Auch hier ist jede Änderung direkt in der aufrufenden Funktion wirksam. Die Funktion fc erwartet den Zeiger auf eine Integer-Variable und stellt so auch eine Möglichkeit der Übergabe dar. Man sollte jedoch die Angabe der Dimensionen nutzen, damit der Betrachter direkt eine Feldübergabe erkennt.

Soll eine Veränderung des Feldes innerhalb der gerufenen Funktion verhindert werden, muss dem Parameter das Schlüsselwort const vorangestellt werden. Dann wird der Compiler jede Änderung des Feldes innerhalb der Funktion mit einer Fehlermeldung ahnden.

Beispiel: Funktion mit konstantem Feld

```
void fa(const int a[4])
{
    a[2] = 3; // Fehlermeldung des Compilers!
}
```

Mehrdimensionale Felder als Parameter

Bei mehrdimensionalen Feldern kann nur die erste Dimension offen bleiben, wie das bei eindimensionalen Feldern der Fall ist. Jede weitere Dimension muss festgelegt werden.

Beispiele: Zulässig sind folgende Prototypen:

```
void fa(int a[3][5]);
void fb(int a[][5]);
void fc(int (*a)[5]);
```

Die zweite Dimension muss angegeben werden, da es ansonsten innerhalb der Funktion nicht möglich ist, die korrekte Speicherstelle zu ermitteln. So hat a[1][1] in unserem Beispiel die sechste Position im Speicher. Wäre die zweite Dimension sieben, dann hätte a[1][1] die achte Stelle. Im Fall eines zweidimensionalen Feldes wird die zweite Dimension auch vom Compiler geprüft. Der Aufruf einer dieser Funktionen mit einem Feld, das als b[5][3] definiert wurde, wird vom Compiler entdeckt und verhindert. Die Klammern um *a sind bei fc erforderlich, da der Parameter ansonsten nicht als zweidimensionales Integer-Feld interpretiert wird, sondern als ein eindimensionales Feld von Zeigern auf Integer.

Anwendungsbeispiel: Lösung linearer Gleichungssysteme – Temperaturverteilung

Der allgemeine Fall eines linearen Gleichungssystems liegt vor, wenn m Gleichungen mit n Unbekannten gegeben sind.

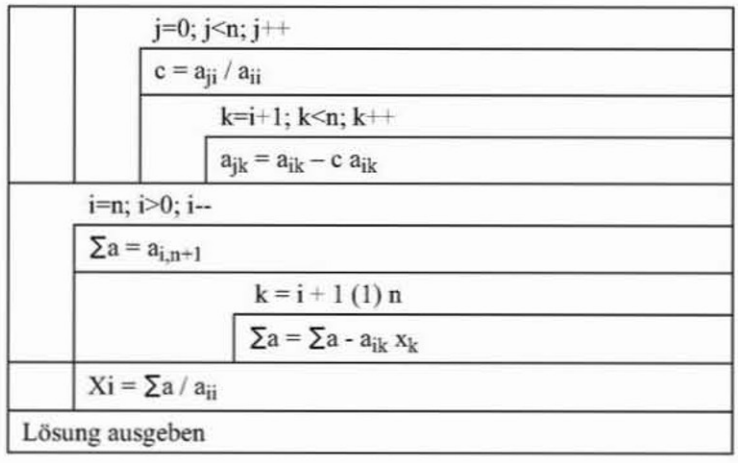
$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = c_1 \\ &a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = c_2 \\ &\ldots \\ &a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = c_m \end{aligned} \tag{2.10}$$

Die reellen Zahlen a_{ik} (i=1,…,m; k=1,…,n) sind die Koeffizienten des Systems. Die reellen Zahlen c_i werden als Absolutglieder bezeichnet. Das Gleichungssystem wird als homogen bezeichnet, wenn die Absolutglieder verschwinden.

Eine Methode zur Bestimmung der Lösung ist das Gaußsche Verfahren, auch als Gaußsches Eliminationsverfahren bezeichnet. Man entfernt durch Multiplikation von Gleichungen mit einer Zahl und Addition zu einer anderen aus (n-1) von n Gleichungen eine Unbekannte. Entfernt aus (n-2) der neuen (n-1) Gleichungen eine zweite Unbekannte. Das Verfahren wird solange wiederholt, bis nur eine Gleichung mit einer Unbekannten vorliegt. Aus ihr wird die Unbekannte bestimmt und durch rückwirkendes Einsetzen alle anderen.

Tabelle 2-2 Struktogramm – Gauß-Elimination

Eingabe der Koeffizienten des Gleichungssystems
i=0; i<n; i++



Code 2-8 Anwendungsbeispiel – Gauß-Elimination

```
/* GaussElimination.cpp
   Loesung eines linearen Gleichungssystems
   nach der Methode der Gauss-Elimination
*/

#include <iostream>
#include <iomanip>
using namespace std;

// Matrizengröße
const int n=4;

void Ausgabe(double a[][n])
{
   for (int k=0;k<n;k++)
   {
      for (int j=0;j<n;j++) cout  <<  setw(7)
         << setiosflags(ios::fixed|ios::right) << a[k][j] << "  ";
      cout << endl;
   }
   cout << endl;
}

int main ()
{
   double a[n][n];      //n*n Matrix
   double b[n],x[n];    //Loesungsvektor
   int i,j,k;           //Counter
   double f;            //Faktor zur Zeilenaddition

// Eingabe der Matrix
   cout << "Bitte Matrix zeilenweise eingeben" << endl;
   for (i=0; i<n; i++)
      for (j=0; j<n; j++)
         cin >> a[i][j];
```

```
// Eingabe des Vektors der rechten Seite
   cout << "Bitte Loesungsvektor eingeben" << endl;
   for (i=0;i<n;i++)
      cin >> b[i];

// Gauss Elimination
// Transformation auf Dreiecksmatrix
   for (i=0;i<n-1;i++)           //n-1 Eliminationsschritte
   {
      Ausgabe(a);
      for (j=i+1; j<n; j++)      //alle folgenden Zeilen bearbeiten
      {
         f = - a[j][i] / a[i][i];//Faktor bestimmen
         a[j][i] = 0.0;          //dieses Element wird zu 0 gemacht
         for (k=i+1;k<n;k++)     //zum Rest der j-ten Zeile das
                                 //f-fache der i-ten Zeile addieren
            a[j][k] = a[j][k]+f*a[i][k];
         b[j] = b[j]+f*b[i];     //ebenso auf der rechten Seite
      }
   }
   Ausgabe(a);

// x[i] aus Dreiecksmatrix bestimmen
   for (i=n-1;i>=0;i--)
   {
      x[i] = b[i];                  //rechte Seite
      for (j=n-1;j>i;j--)
         x[i] = x[i]-a[i][j]*x[j]; //alle bekannten x[j] einsetzen
      x[i] = x[i]/a[i][i];         //durch Diagonalelement dividieren
   }

// Ergebnis ausgeben
   cout << "Ergebnis: " << endl;
   for (i=0;i<n;i++)
      cout << x[i] << endl;

   system("Pause");
}
```

Ein einfaches Beispiel soll die Anwendung demonstrieren.

$$\begin{aligned} 2x_1 + 6x_2 - 3x_3 + 12x_4 &= -6 \\ 4x_1 + 3x_2 + 3x_3 + 15x_4 &= 6 \\ 4x_1 - 3x_2 + 6x_3 + 6x_4 &= 6 \\ -3x_2 + 5x_3 - 2x_4 &= 14 \end{aligned} \tag{2.11}$$

Bild 2-13 zeigt das Ergebnis.

Temperaturverteilung

Nun folgt als Praxisbeispiel (**Bild 2-14**) die Temperaturverteilung innerhalb eines Kanals mit rechteckigem Querschnitt. An der Rohrwand werden unterschiedliche Temperaturen gemessen.

Es sollen alle Temperaturen an den Gitter-Punkten bestimmt werden unter der Annahme, dass ein innen liegender Punkt den Mittelwert aller benachbarten Punkte hat.

```
Bitte Matrix zeilenweise eingeben
2
6
-3
12
4
3
3
15
4
-3
6
6
0
-3
5
-2
Bitte Loesungsvektor eingeben
-6
6
6
14
2.000000  6.000000  -3.000000  12.000000
4.000000  3.000000  3.000000  15.000000
4.000000  -3.000000  6.000000  6.000000
0.000000  -3.000000  5.000000  -2.000000

2.000000  6.000000  -3.000000  12.000000
0.000000  -9.000000  9.000000  -9.000000
0.000000  -15.000000  12.000000  -18.000000
0.000000  -3.000000  5.000000  -2.000000

2.000000  6.000000  -3.000000  12.000000
0.000000  -9.000000  9.000000  -9.000000
0.000000  0.000000  -3.000000  -3.000000
0.000000  0.000000  2.000000  1.000000

2.000000  6.000000  -3.000000  12.000000
0.000000  -9.000000  9.000000  -9.000000
0.000000  0.000000  -3.000000  -3.000000
0.000000  0.000000  0.000000  -1.000000

Ergebnis:
-3.000000
2.000000
4.000000
-0.000000
Drücken Sie eine beliebige Taste . . .
```

Bild 2-13 Ergebnis der Beispielauswertung

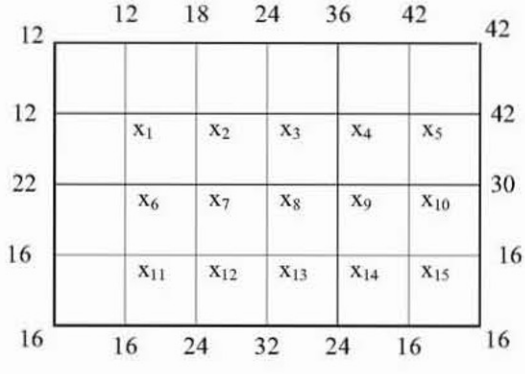


Bild 2-14 Gitter-Modell mit Randtemperaturen in Grad Celsius

Jeder Punkt geht zu einem Viertel in die Gleichungen ein und da wir nicht umständlich mit Brüchen arbeiten wollen, multiplizieren wir die Gleichungen mit 4 und so ergeben sich die nachfolgenden Gleichungen.

$$
\begin{aligned}
x_1&: 4x_1 - x_2 - x_6 = 24\\
x_2&: -x_1 + 4x_2 - x_3 - x_7 = 18\\
x_3&: -x_2 + 4x_3 - x_4 - x_8 = 24\\
x_4&: -x_3 + 4x_4 - x_5 - x_9 = 36\\
x_5&: -x_4 + 4x_5 - x_{10} = 84\\
x_6&: -x_1 + 4x_6 - x_7 - x_{11} = 22\\
x_7&: -x_2 - x_6 + 4x_7 - x_8 - x_{12} = 0\\
x_8&: -x_3 - x_7 + 4x_8 - x_9 - x_{13} = 0\\
x_9&: -x_4 - x_8 + 4x_9 - x_{10} - x_{14} = 0\\
x_{10}&: -x_5 - x_9 + 4x_{10} - x_{15} = 30\\
x_{11}&: -x_6 + 4x_{11} - x_{12} = 34\\
x_{12}&: -x_7 - x_{11} + 4x_{12} - x_{13} = 24\\
x_{13}&: -x_8 - x_{12} + 4x_{13} - x_{14} = 32\\
x_{14}&: -x_9 - x_{13} + 4x_{14} - x_{15} = 24\\
x_{15}&: -x_{10} - x_{14} + 4x_{15} = 32
\end{aligned}
\tag{2.12}
$$

Die entsprechenden Daten liest die Hauptfunktion aus einer zusätzlichen Datenfunktion. Das Hauptprogramm ist entsprechend anzupassen (Leser).

Code 2-9 Ergänzende Dateneingabe-Funktion

```
void Daten (double a[][n],double b[])
{
   int i, j;

   cout << "-------- Daten ----------";
   cout << "n = " << n << endl;
   for (i=0; i<n; i++)
      for (j=0; j<n; j++)
      {
         a[i][j] = 0;
         if (i==j) a[i][j] = 4;
         }
   a[0][1] = -1, a[0][5] = -1;
   a[1][0] = -1, a[1][2] = -1;
   a[1][6] = -1;
   a[2][1] = -1, a[2][3] = -1;
   a[2][7] = -1;
   a[3][2] = -1, a[3][4] = -1;
   a[3][8] = -1;
   a[4][3] = -1, a[4][9] = -1;
   a[5][0] = -1, a[5][8] = -1;
   a[5][10] = -1;
   a[6][1] = -1, a[6][5] = -1;
   a[6][7] = -1, a[6][11] = -1;
   a[7][2] = -1, a[7][6] = -1;
   a[7][8] = -1, a[7][12] = -1;
```

```
    a[8][3] = -1, a[8][7] = -1;
    a[8][9] = -1, a[8][13] = -1;
    a[9][4] = -1, a[9][8] = -1;
    a[9][14] = -1;
    a[10][5] = -1, a[10][11] = -1;
    a[11][6] = -1, a[11][10] = -1;
    a[11][12] = -1;
    a[12][7] = -1, a[12][11] = -1;
    a[12][13] = -1;
    a[13][8] = -1, a[13][12] = -1;
    a[13][14] = -1;
    a[14][9] = -1, a[14][13] = -1;
    b[0] = 24;
    b[1] = 18;
    b[2] = 24;
    b[3] = 36;
    b[4] = 84;
    b[5] = 22;
    b[6] = 0;
    b[7] = 0;
    b[8] = 0;
    b[9] = 30;
    b[10] = 34;
    b[11] = 24;
    b[12] = 32;
    b[13] = 24;
    b[14] = 32;
}
```

Die ermittelten Temperaturdaten lassen sich in einem Diagramm zur Verdeutlichung darstellen.

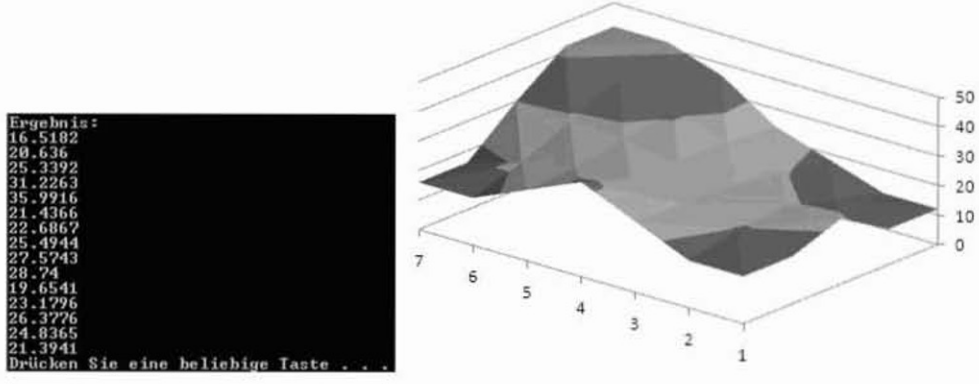


Bild 2-15 Ergebnis der Temperaturverteilung

Übungen

Schreiben Sie ein ergänzendes Programm, mit dessen Hilfe Sie die Randtemperaturen verändern können. Wie sieht die Temperaturverteilung für ein rundes Rohr aus?

2.4 Deklarationen und Gültigkeitsbereiche

Deklarationen

Bevor ein Bezeichner in einem C++ Programm benutzt werden kann, muss er deklariert werden. Für einfache Datentypen wurde dies bereits beschrieben. Eine Deklaration teilt dem Compiler mit, auf welche Art von Entität sich der Name bezieht.

Beispiele: Deklarationen und Initialisierungen

```
char z;
int count = 5;
const double pi = 3.14159265;
extern int fehler;
const char* jahreszeit[] = {"Frühling","Sommer","Herbst","Winter"};
```

Deklarationen assoziieren nicht nur einen Typ mit einem Namen, sie definieren auch eine Entität für den Namen, auf den sie sich beziehen.

Eine Deklaration besteht aus vier Teilen: Einem optionalen Spezifizierer, einem Basistyp, einem Deklarator und einem optionalen Initialisierer. Mit Ausnahme von Funktionen und Namensbereichsdefinitionen wird ein Deklarator mit einem Semikolon beendet.

Beispiel: Deklaration

```
char* Formen[] = {"Kreis","Rechteck","Dreieck"};
```

Hier ist der Basistyp char, der Deklarator *Formen[] und der Initialisator ={...}.

Ein Spezifizierer ist ein Schlüsselwort wie etwa virtual oder extern, das ein Nicht-Typ-Attribut für das Deklarierte spezifiziert. Ein Deklarator ist aus einem Namen und optionalen Deklaratoroperatoren zusammengesetzt. Die häufigsten sind in der nachfolgenden Tabelle aufgeführt.

Tabelle 2-3 Deklaratoren

Deklarator-operator	Bezeichnung	Typ
*	Zeiger	Präfix
*const	konstanter Zeiger	Präfix
&	Referenz	Präfix
[]	Feld	Postfix
()	Funktion	Postfix

Die Postfixoperatoren binden stärker als die Präfixoperatoren.

Deklarieren mehrerer Namen

Es ist möglich, mehrere Namen in einer einzigen Deklaration zu deklarieren.

Beispiele: Deklarationen von Variablen

```
int x,y;
char a,b,c;
```

Gültigkeitsbereiche

Eine Deklaration führt einen Namen in einen Gültigkeitsbereich ein. Der Name kann somit in einem bestimmten Teil des Programms benutzt werden.

Für einen in einer Funktion deklarierten Namen (oft als lokaler Name bezeichnet) erstreckt sich der Gültigkeitsbereich vom Punkt der Deklaration bis zum Ende des Blocks, in dem die Deklaration auftrat. Ein Block ist ein von einem Paar geschweifter Klammern {} begrenzter Codebereich.

Ein Name wird als global bezeichnet, wenn er außerhalb einer Funktion, einer Klasse oder eines Namensbereichs definiert wird. Der Gültigkeitsbereich eines globalen Namens erstreckt sich vom Punkt seiner Deklaration bis zum Ende der Datei, in der er deklariert wurde. Die Deklaration eines Namens in einem Block kann eine Deklaration in einem umschließenden Block oder einen globalen Namen verdecken. Ein Name kann in einem Block so definiert werden, dass er sich auf eine andere Entität bezieht. Nach Verlassen des Blocks erhält der Name seine vorherige Bedeutung zurück.

Code 2-10 Demoprogramm – globale und lokale Variable

```
/* GlobalLokal.cpp
   Gültigkeitsbereiche von Variablen
*/
#include <iostream.h>

int x = 9;    //  globales x

void f()
{
   int x;     //lokales x verdeckt globales x
   x=1;       //Zuweisung an lokales x
   cout << "x.L = " << x << endl;
   {
      int x; //verdeckt erstes lokales x
      x=2;   //Zuweisung an zweites lokales x
      cout << "x.V = " << x << endl;
   }
   cout << "x.L = " << x << endl;
   x=3;       //Zuweisung an erstes lokales x
   cout << "x.L = " << x << endl;
}

int main()
{
   f();
   int* p = &x; //nimmt die Adresse des globalen x
   cout << "p = " << *p << endl;

   system("Pause");
}
```

```
x.L = 1
x.U = 2
x.L = 1
x.L = 3
p = 9
Drücken Sie eine beliebige Taste . . .
```

Bild 2-16 Programmausgabe - globale und lokale Variable

Ein verdeckt globaler Name kann durch den Bereichsoperator (::) sichtbar gemacht werden.

Code 2-11 Demoprogramm – verdeckt globale Variable

```
/* VerdecktGlobal.cpp
   Verdeckt globale Variable
*/

#include <iostream.h>

int x = 9;    //  globales x

void f()
{
   int x=1;  //lokales x verdeckt globales x
   cout << "x.L = " << x << endl;
   ::x=2;     //Zuweisung an globales x
   x=3;       //Zuweisung an lokales x
   cout << "x.L = " << x << endl;
}

int main()
{
   f();
   int* p = &x; //nimmt die Adresse des globalen x
   cout << "p = " << *p << endl;

   system("Pause");
}
```

```
x.L = 1
x.U = 2
x.L = 1
x.L = 3
p = 9
Drücken Sie eine beliebige Taste . . .
```

Bild 2-17 Programmausgabe – verdeckt globale Variable

Es ist auch möglich, mit einem einzigen Namen ohne Benutzung des Bereichs-Operators (::) zwei verschiedene Objekte in einem Block anzusprechen.

Code 2-12 Verschiedene Objekte

```
/* Verschiedene.cpp
   Verschiedene Objekte
*/

#include <iostream.h>

int x = 9;          //  globales x

void f()
{
   int y=x;         //benutzt globales x: y=9
   cout << "y.L = " << x << endl;
   x=22;            //Zuweisung an lokales x
   y=x;             //benutzt lokales x:y=22
   cout << "x.L = " << x << endl;
   cout << "y.L = " << x << endl;
}

int main()
{
   f();
   int* p = &x; //nimmt die Adresse des globalen x
   cout << "p = " << *p << endl;

   system("Pause");
}
```

```
y.L = 9
x.L = 22
y.L = 22
p = 22
Drücken Sie eine beliebige Taste . . .
```

Bild 2-18 Programmausgabe – verschiedene Objekte

Namensräume

Ein Namensraum (engl. namespace) ist ein Gültigkeitsbereich, in dem beliebige Objekte wie Variablen, Funktionen und andere Strukturen deklariert werden können. Alle Bibliotheken der Standard Template Library (STL) sind zum Namensraum *std* zusammengefasst und werden durch die using-Anweisung

Syntax: Definition des Standard-Namensraums

```
using namespace std;
```

deklariert, wie bereits in einigen Beispielen gezeigt. Ein Namensraum kann deklariert werden und Objekte in ihm werden durch den Bereichs-Operator (::) angesprochen. Der Bereichs-Operator wird auch als Scope-Operator bezeichnet.

Code 2-13 Demoprogramm – Namensräume

```
/* Namensraeume.cpp
   Zeigt die Wirkung von Namensräumen
*/

#include <iostream.h>

// Deklaration von Namensraum A
namespace A
{
   int i = 5;
   int f(int x) {return 2*x+1;}
}

// Deklaration von Namensraum B
namespace B
{
   int i = 6;
   int f(int x) {return 2*x;}
}

int main ()
{
   cout << "Der Wert von i in A: " << A::i << endl;
   cout << "Der Wert von i in B: " << B::i << endl;
   cout << "Der Wert von f(3) in A: " << A::f(3) << endl;
   cout << "Der Wert von f(4) in B: " << B::f(4) << endl;

   system("Pause");
}
```

```
Der Wert von i in A: 5
Der Wert von i in B: 6
Der Wert von f(3) in A: 7
Der Wert von f(4) in B: 8
Drücken Sie eine beliebige Taste . . .
```

Bild 2-19 Programmausgabe – Namensräume

Namensräume können auch geschachtelt deklariert werden.

Code 2-14 Demoprogramm – geschachtelte Namensräume

```
/* Geschachtelt.cpp
   Geschachtelte Namensräume
*/

#include <iostream.h>

//Deklaration der Namensraeume
namespace A
{
```

```
    int i = 5;
    namespace B {int i = 6;}
}

int main ()
{
    int i = 4;
    cout << "Der Wert von i in main: " << i << endl;
    cout << "Der Wert von i in A: " << A::i << endl;
    cout << "Der Wert von i in B: " << A::B::i << endl;

    system("Pause");
}
```

```
Der Wert von i in main: 4
Der Wert von i in A: 5
Der Wert von i in B: 6
Drücken Sie eine beliebige Taste . . .
```

Bild 2-20 Programmausgabe – geschachtelte Namensräume

Die Speicherklasse automatic

Für lokale Variable und Funktionsargumente wird automatisch Speicherplatz reserviert. Mit jedem Einstieg in eine Funktion oder einen Block { } bekommen diese eigene Kopien. Will man diesen Sachverhalt explizit angeben, so gibt es dazu das (überflüssige) Schlüsselwort *auto*.

Beispiel:

```
//Folgende Deklarationen haben die gleiche Bedeutung
int i = 5;
auto int i = 5;
```

Außerhalb ihres Blocks sind *automatic* vereinbarte Variable unsichtbar.

Code 2-15 Demoprogramm – *automatic* vereinbarte Variable

```
/* Automatic.cpp
   Automatic vereinbarte Variable
*/

#include <iostream.h>

int main ()
{
    auto int i = 4;
    {
        int i = 5;
        cout << "Der Wert von i: " << i << endl;
    }
    cout << "Der Wert von i: " << i << endl;

    system("Pause");
}
```

```
Der Wert von i: 5
Der Wert von i: 4
Drücken Sie eine beliebige Taste . . .
```

Bild 2-21 Programmausgabe – *automatic* vereinbarte Variable

Die Speicherklasse static

Im Gegensatz zur Speicherklasse *automatic* behält eine statische Variable ihren Wert so lange, bis sie eine neue Wertzuweisung erhält. Wird eine statische Variable am Anfang nur definiert und nicht deklariert, so erhält sie vom Compiler automatisch das Nullelement des Datentyps.

Code 2-16 Demoprogramm – *static* vereinbarte Variable

```
/* Static.cpp
   Static vereinbarte Variable
*/

#include <iostream.h>

int main ()
{
   int Summe(int x);
   for (int i=1; i<5; i++)
      cout << i << "\t" << Summe(i) << endl;

   system("Pause");
}

int Summe(int x)
{
   static int s;

   int i = 5;
   return (s+=x);
}
```

```
1       1
2       3
3       6
4       10
Drücken Sie eine beliebige Taste . . .
```

Bild 2-22 Programmausgabe--*static* deklarierte Variable

Die Speicherklasse extern

Ist eine Variable nicht in einem Block deklariert, muss sie dort als *extern* deklariert werden.

Code 2-17 Demoprogramm – *extern* vereinbarte Variable

```
/* Extern.cpp
   Extern vereinbarte Variable
*/

#include <iostream.h>

int main ()
{
   int i = 1;
   extern int k;
   cout << i << "\t" << k << endl;

   system("Pause");
}

int k = 3;    //extern
```

```
1       3
Drücken Sie eine beliebige Taste . . .
```

Bild 2-23 Programmausgabe – *extern* deklarierte Variable

Funktionsnamen sind ebenso im ganzen Programm deklariert und damit gehören sie auch zur Speicherklasse *extern*.

Die Speicherklasse volatile

Mit *volatile* deklarierte Variable sind von der Compiler-Optimierung ausgeschlossen. Dies ist besonders für Systemvariable wichtig.

Anwendungsbeispiel: Lineare Optimierung – Produktionsoptimierung

Bleiben wir weiter bei den Gleichungssystemen. Bei vielen wirtschaftlichen Problemen gilt es für eine Ziel- oder Kostenfunktion mit n Variablen

$$f = c_0 + c_1x_1 + c_2x_2 + ... + c_nx_n \tag{2.13}$$

einen Extremwert (minimale Kosten, maximale Ausnutzung, etc.) zu finden, für die zusätzlich gegebene Nebenbedingungen (Gleichungen oder Ungleichungen) erfüllt sind.

$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = a_1 \\ &a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n \leq a_2 \\ &\text{usw.} \end{aligned} \tag{2.14}$$

mit $x_1>=0,\ldots, x_n>=0$.

Zur Lösung gibt es einige Verfahren. Wir befassen uns an dieser Stelle mit der Simplex-Methode von Dantzig. Eine Produktion erstellt mit zwei Maschinengruppen M_1, M_2 zwei Produkte P_1, P_2 mit folgenden Zeiten und Gewinnen:

Tabelle 2-4 Produktionsübersicht

	M_1	M_2	Gewinn
P_1 (A)	6	2	3 Euro
P_2 (B)	4	4	5 Euro
	160 Std./Woche	120 Std./Woche	

Die Firma arbeitet mit 40 Stunden/Woche und verfügt über 4 Maschinen der Gruppe M1 und 3 Maschinen der Gruppe M2. Unter der Annahme, dass immer die ganze Produktion verkauft werden kann, ist der maximale Gewinn gesucht.

Kommen wir zur Aufstellung des Gleichungssystems. A sei die wöchentliche Produktion des Produkts P1 und B die wöchentliche Produktion des Produktes P2. Mit der Gruppe M1 stehen 160 Stunden und mit der Gruppe M2 120 Stunden pro Woche zur Verfügung. Damit gilt:

Gesucht: Maximum von

$$3A+5B \Rightarrow \text{Maximum} \tag{2.15}$$

Restriktionen:

$$\begin{aligned} &6A+4B \leq 160 \\ &2A+4B \leq 120 \\ &A,B \geq 0 \end{aligned} \tag{2.16}$$

Grafisch ist dieses Problem direkt lösbar, wenn man die Restriktionen als Funktionen im gültigen Bereich zeichnet.

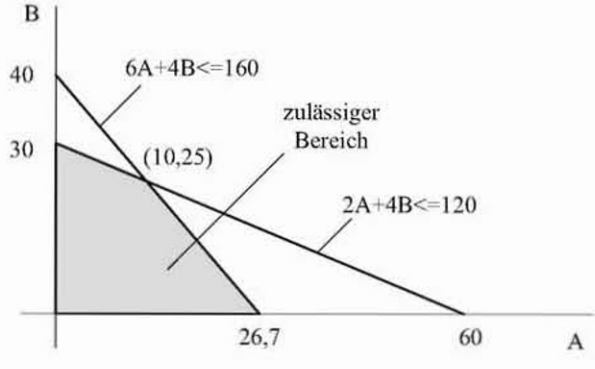


Bild 2-24 Grafische Lösung

Doch wir interessieren uns für die Simplex-Methode. Dazu erstellt man für das Problem eine Tabelle der nachfolgenden Form.

Als Pivotspalte wird diejenige gewählt, in der in der untersten Zeile der größte negative Wert steht. In diesem Fall -5. Als Pivotzeile wird diejenige gewählt, bei der sich der Quotient aus Wert der letzten Spalte geteilt durch Wert der Pivotspalte den kleinsten Wert ergibt. In diesem Fall ist 160/4 = 40 > 120/4 = 30. Somit ist das Pivotelement die 4 in der zweiten Zeile.

Tabelle 2-5 Auswertungstabelle

6	4	160
2	4	120
-3	-5	0

Die Tabelle wird nun folgendermaßen umgeformt. Die Elemente in der Pivotzeile werden durch das Pivotelement dividiert. Die Elemente der Pivotspalte werden durch das Pivotelement dividiert und bekommen das umgekehrte Vorzeichen. Das Pivotelement selbst wird durch seinen Reziprokwert ersetzt. Alle übrigen Elemente werden nach der Rechteckregel umgeformt.

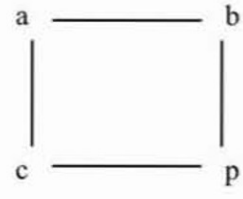


Bild 2-25 Rechteckregel

Sei a ein beliebiges Element aus der alten Tabelle, b und c die im Rechteck zum Pivotelement p angeordneten Elemente der alten Tabelle, dann ergibt sich für den neuen Wert

$$a = a - b \cdot \frac{c}{p}. \tag{2.17}$$

Die neue Tabelle sieht also bereits nach der ersten Umformung wie in **Tabelle 2-6** dargestellt aus.

Tabelle 2-6 Auswertungstabelle nach der ersten Umformung

4,00	-1,00	40
0,50	0,25	30
-0,50	1,25	150

Die Tabelle wird solange umgeformt, bis in der unteren Zeile nur noch positive Elemente stehen.

Tabelle 2-7 Auswertungstabelle nach der letzten Umformung

0,25	-0,25	10
-0,125	0,375	25
0,125	1,125	155

Zusätzlich werden die Elemente der untersten Zeile und die Elemente der letzten Spalte mit fortlaufenden Zahlen markiert. Bei jeder Umwandlung werden dann die Marken von Pivotzeile und Pivotspalte vertauscht. Am Ende stehen in den Zeilen 1, 2,..., n in der hinteren Spalte die Werte für $x_1, x_2, \ldots, x_n$.

Das Verfahren liefert das gleiche Ergebnis, wie wir es von der grafischen Lösung kennen, nämlich dass das Maximum bei $3 \cdot 10 + 5 \cdot 25 = 155$ Euro liegt.

Tabelle 2-8 Struktogramm – Simplex-Methode

Grunddaten n=4				
Feld-daten	i=0; i<n-1; i++			
		j=0; i<n-1; i++		
		a(i,j) = ...		
do				
	Bestimme Pivotspalte	i=0; i<n-3; i++		
		x=min(a(n-2,i),a(n-2,i+1))		
		x= a(n-2,i)		
		Ja	Nein	
		ps=i	./.	
		x= a(n-2,i+1)		
		Ja	Nein	
		ps=i+1	./.	
	Bestimme Pivotzeile	i=0; i<n-3; i++		
		x=a(i,n-2)/a(i,ps)		
		y=a(i+1,n-2)/a(i+1,ps)		
		z=min(x,y)		
		z=x		
		Ja	Nein	
		pz=i	./.	
		z=y		
		Ja	Nein	
		pz=i+1	./.	

	Umformung Pivotzeile	i=0; i<n-1; i++	
		i!=ps	
		Ja	Nein
		$b(pz,i) = \frac{a(pz,i)}{a(pz,ps)}$	./.
	Umformung Pivotspalte	i=0; i<n-1; i++	
		i!=pz	
		Ja	Nein
		$b(i,ps) = -\frac{a(i,ps)}{a(pz,ps)}$	./.
	Umformung Pivotelement	$b(pz,ps) = \frac{1}{a(pz,ps)}$	
	Umformung Feld	i=0; i<n-1; i++	
		j=0; i<n-1; j++	
		i!=pz && j!=ps	
		Ja	Nein
		$b(i,j) = a(i,j) - a(i,ps)\frac{a(pz,j)}{a(pz,ps)}$	./.
	Übergabe a(i,j)=b(i,j), i=0…n-1, j=0…n-1		
	Marken vertauschen x=a(pz,n-1) a(pz,n-1)=a(n-1,ps) a(n-1,ps)=x		
	Schleifenbedingung	i=0; i<n-2; i++	
	k=0	a(n-2,i)<0	
		Ja	Nein
		k=1	./.
	solange k=1		

Code 2-18 Anwendungsbeispiel Lineare Optimierung – Produktionsoptimierung

```
/* ProduktionsOptimierung.cpp
   Das Programm bestimmt die
   Lösung eines Testbeispiels
*/

#include <iostream>
#include <iomanip>
```

```
#include <math.h>
#include <stdlib.h>
using namespace std;
const int n=4;

void Ausgabe (double a[][n])
{
   int i, j;

   for (int i=0;i<n;i++)
   {
      for (int j=0;j<n;j++) cout  <<  setw(7)
         << setiosflags(ios::fixed|ios::right) << a[i][j] << "  ";
      cout << endl;
   }
   cout << endl;
}

int main ()
{
   int i, j;
   int ps, pz;
   double a[4][4], b[4][4];
   double x, y, z;

// Beispieldaten
   a[0][0] = 6;
   a[0][1] = 4;
   a[0][2] = 160;
   a[1][0] = 2;
   a[1][1] = 4;
   a[1][2] = 120;
   a[2][0] = -3;
   a[2][1] = -5;
   a[2][2] = 0;
// Marken
   a[3][0] = 1;
   a[3][1] = 2;
   a[3][2] = 3;
   a[3][3] = 4;
   a[0][3] = 1;
   a[1][3] = 2;
   a[2][3] = 3;

   Ausgabe(a);

// Auswertungsschleife
   do {
   // Pivotspalte
      for (i=0; i<n-3; i++)
      {
         x = min(a[n-2][i],a[n-2][i+1]);
         if (x==a[n-2][i]) ps=i;
         if (x==a[n-2][i+1]) ps=i+1;
      }
```

```
      cout << "PivotSpalte = " << ps << endl;

   // Pivotzeile
      for (i=0; i<n-3; i++)
      {
         x = a[i][n-2]/a[i][ps];
         y = a[i+1][n-2]/a[i+1][ps];
         z = min(x, y);
         if (z==x) pz=i;
         if (z==y) pz=i+1;
      }
      cout << "PivotZeile = " << pz << endl;
      cout << "Pivotelement = " << a[pz][ps] << endl << endl;

   // Umformung der Pivotzeile
      for (i=0; i<n-1; i++)
         if (i!=ps)
            b[pz][i] = a[pz][i] / a[pz][ps];

   // Umformung der Pivotspalte
      for (i=0; i<n-1; i++)
         if (i!=pz)
            b[i][ps] = - a[i][ps]/a[pz][ps];

   // Pivotelement
      b[pz][ps] = 1/a[pz][ps];

   // Tabellenrest
      for (i=0; i<n-1; i++)
         for (j=0; j<n-1; j++)
            if (i!=pz && j!=ps)
               b[i][j] = a[i][j]-a[i][ps]*a[pz][j]/a[pz][ps];

   // Übergabe
      for (i=0;i<n-1;i++)
         for (j=0;j<n-1;j++)
             a[i][j]=b[i][j];

   // Marken vertauschen
      x = a[pz][n-1];
      a[pz][n-1] = a[n-1][ps];
      a[n-1][ps] = x;

   // Ausgabe
      Ausgabe(a);

   // Abbruch
      j=0;
      for (int i=0;i<n-2;i++)
      {
         if (a[n-2][i]<0.0) j=1;
      }
   } while (j==1);
   system("Pause");
}
```

```
6.000000  4.000000  160.000000  1.000000
2.000000  4.000000  120.000000  2.000000
-3.000000  -5.000000  0.000000  3.000000
1.000000  2.000000  3.000000  4.000000

PivotSpalte = 1
PivotZeile = 1
Pivotelement = 4.000000

4.000000  -1.000000  40.000000  1.000000
0.500000  0.250000  30.000000  2.000000
-0.500000  1.250000  150.000000  3.000000
1.000000  2.000000  3.000000  4.000000

PivotSpalte = 0
PivotZeile = 0
Pivotelement = 4.000000

0.250000  -0.250000  10.000000  1.000000
-0.125000  0.375000  25.000000  2.000000
0.125000  1.125000  155.000000  3.000000
1.000000  2.000000  3.000000  4.000000

Drücken Sie eine beliebige Taste . . .
```

Bild 2-26 Ergebnis des Testbeispiels

Übungen

Schreiben Sie das Programm so um, dass beliebige Ungleichungen erfasst und ausgewertet werden können.

2.5 Zeiger auf Funktionen

Datenzeiger und Funktionszeiger

In C++ lassen sich keine Variablen erklären, die Funktionen als Wert haben. Dies scheitert technisch daran, dass je zwei Funktionen im Allgemeinen verschiedenen Speicherbedarf haben. Erlaubt sind dagegen Zeiger auf Funktionen. Ebenso wie Datenzeiger sind Funktionszeiger typisiert. Zwei Funktionszeiger sind nur dann kompatibel, wenn sie auf Funktionen mit dem gleichen Parameterschema (gleiche Anzahl, gleiche Typen) und typgleichem Funktionswert zeigen.

Wie Zeiger auf Funktionen definiert werden, zeigt das nachfolgende Beispiel.

Beispiel: Deklaration einer Funktion mit zwei int-Datentypen und dem Rückgabetyp char

```
char Summe (int, int);
```

Beispiel: Deklaration eines Zeigers auf eine Funktion vom vorhergehenden Beispieltyp

```
char (*FktZeiger) (int, int);
```

Die Klammern um den Namen mit dem Stern dürfen nicht vergessen werden, denn sonst wird kein Zeiger auf eine Funktion deklariert, sondern eine Funktion, die einen Zeiger auf (im Beispiel) *char* zurückgibt.

Beispiel: Keine Deklaration eines Funktionszeigers

```
char *Zeiger (int, int);
```

Um einem Zeiger auf eine Funktion etwas zuzuweisen, müsste eigentlich die Adresse einer Funktion bestimmt werden. Für Zeiger auf Funktionen gilt eine ähnliche Konvention wie bei den Feldtypen. Funktionsnamen stehen prinzipiell für einen Zeiger auf die Funktion. Erst durch die runden Klammern der Parameter dahinter wird der Zeiger dereferenziert und die Funktion aufgerufen. Deshalb müssen in C++ auch beim Aufruf einer Funktion ohne Parameter immer die runden Klammern stehen!

Das folgende Beispiel zeigt, wie Zeiger auf Funktionen definiert werden, und wie man ihnen Funktionen als Wert zuweist:

Code 2-19 Demoprogramm für die Anwendung von Funktionszeigern

```
/* Funktionszeiger.cpp
   f ist ein Zeiger auf eine Funktion
   mit einem Argument vom Typ int
   und dem Ergebnistyp int
*/

#include <iostream.h>

int (*f) (int);
int f1 (int i)   {return i;}
int f2 (int n)   {return 2*n;}
int f3 (float x) {return 3*x;}
```

```
int main ()
{
    int n;

    f = f1;          // steht für f = &f1;
    n = f(1);        // steht für n = (*f)(1); jetzt ist n = 1
    cout << n << endl;
    f = f2;
    n = f(1);        // jetzt ist n = 2
    cout << n << endl;
//  f = f3;          // Fehler: unverträgliche Zeigertypen!

    system("Pause");
}
```

```
1
2
Drücken Sie eine beliebige Taste . . .
```

Bild 2-27 Programmausgabe – Funktionszeiger

Anwendungsbeispiel: Differentialgleichungen – Mechanische Schwingungen

Bei Differentialgleichungen handelt es sich um Gleichungen, die zur Berechnung einer bestimmten Funktion dienen. Das Wesentliche einer Differentialgleichung ist, dass neben der Funktion oder der unabhängig Veränderlichen auch mindestens eine Ableitung der gesuchten Funktion auftritt.

Die numerische Behandlung einer Differentialgleichung lässt sich nach vielen Methoden durchführen. Das Euler-Cauchy-Verfahren ist eine einfache Methode und hat damit eine größere Fehlerrate gegenüber anderen Methoden. Da sie aber einfach zu handhaben ist, findet sie oft Anwendung.

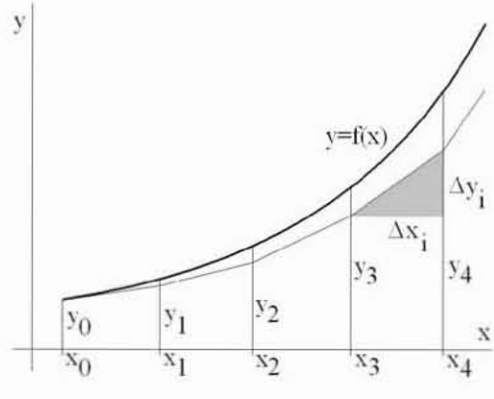


Bild 2-28 Näherung nach Euler-Cauchy

Es sei

$$y = f(x) \tag{2.18}$$

die analytische Lösung der Differentialgleichung

$$y' = f(x, y). \tag{2.19}$$

Aus der Differentialgleichung folgt die Anfangsbedingung

$$y_0' = f(x_0, y_0) \tag{2.20}$$

als bekannter Wert. Eine Veränderung des Abszissenwertes

$$x_1 = x_0 + \Delta x \tag{2.21}$$

ergibt den neuen Ordinatenwert

$$y_1 = y_0 + y_0' \cdot \Delta x. \tag{2.22}$$

Auf diese Weise erhält man einen Polygonzug, der der gesuchten Lösungsfunktion angenähert ist. Bei diesem Verfahren, wird also der Differentialquotient

$$\frac{dy}{dx} \tag{2.23}$$

durch den Differenzenquotienten

$$\frac{\Delta y}{\Delta x} \tag{2.24}$$

ersetzt.

Wählt man Δx genügend klein, kommt man der analytischen Lösung beliebig nahe. Der Nachteil wird aus der Darstellung ebenfalls recht deutlich. Mit zunehmenden Schritten entfernt sich die numerische Lösung immer mehr von der analytischen. Ein genaueres Verfahren ist z. B. das Runge-Kutta-Verfahren.

Die Betrachtung des Kräftegleichgewichts an beweglichen Massenpunkten führt unmittelbar zu einer Differentialgleichung. Ob dies nun einfache Bewegungsmodelle wie z. B. Rotationen sind oder komplexere Modelle wie z. B. Schwingungssysteme. Der Änderung des Bewegungszustandes setzt der Massenpunkt seine träge Masse entgegen

$$F = -m \cdot a. \tag{2.25}$$

Die momentane Beschleunigung a ergibt sich als Differentialquotient

$$a = \frac{dv}{dt} = v', \tag{2.26}$$

so dass die erste Anwendung des Euler-Cauchy-Verfahrens die während der Zeiteinheit dt auftretende Geschwindigkeitsänderung dv liefert

$$v = -\frac{F}{m} t. \tag{2.27}$$

Für die momentane Geschwindigkeit gilt weiterhin der Differentialquotient

$$v = \frac{ds}{dt} = s', \tag{2.28}$$

so dass die zweite Anwendung des Euler-Cauchy-Verfahrens den während der Zeiteinheit dt zurückgelegten Weg ds annähernd beschreibt

$$s = v \cdot t . \tag{2.29}$$

Die momentane Geschwindigkeit wird dabei aus Einfachheitsgründen für die Berechnung durch die Anfangsgeschwindigkeit des betrachteten Zeitintervalls ersetzt.

Als konkretes Beispiel soll nachfolgend eine mechanische Schwingung dienen. Unter einer mechanischen Schwingung versteht man die periodische Bewegung einer Masse um eine Mittellage. Den einfachsten Fall bildet ein Feder-Masse-System. Bei der Bewegung findet ein ständiger Energieaustausch zwischen potentieller und kinetischer Energie statt. Die potentielle Energiedifferenz wird auch als Federenergie bezeichnet. Die bei der Bewegung umgesetzte Wärmeenergie durch innere Reibung in der Feder soll unberücksichtigt bleiben. Wirken auf ein schwingendes System keine äußeren Kräfte, bezeichnet man den Bewegungsvorgang als freie Schwingung, andernfalls als erzwungene Schwingung.

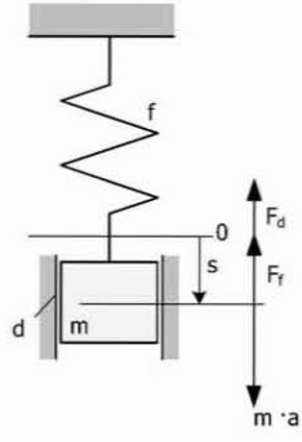


Bild 2-29 Freie gedämpfte Schwingung

Die bei der realen Schwingung stets auftretende Widerstandskraft, Bewegung im Medium und Reibungskraft (Stokes'sche Reibung, im Gegensatz zur Coulomb'schen oder Newton'schen Reibung), etc., soll in erster Näherung als geschwindigkeitsproportional angesehen werden. Bei der Betrachtung einer freien gedämpften Schwingung wirkt zum Zeitpunkt t an der Masse die Federkraft

$$F_f = f \cdot s \tag{2.30}$$

mit der Federkonstante f. Für die Dämpfungskraft folgt unter Einführung der Dämpfungskonstanten d als Maß für die Dämpfungsintensität

$$F_d = 2 \cdot m \cdot d \cdot \dot{s} \tag{2.31}$$

Nach dem d'Alembertschen Prinzip folgt

$$m \cdot \ddot{s} = -f \cdot s - 2 \cdot m \cdot d \cdot \dot{s} . \tag{2.32}$$

Umgestellt

$$\frac{dv}{dt} = -\frac{f}{m}s - 2 \cdot d \cdot v \tag{2.33}$$

Nach dem Euler-Cauchy-Verfahren ersetzt man den in der Gleichung (2.33) enthaltenen Differentialquotienten durch einen Differenzenquotienten

$$\Delta v = -\left(\frac{f}{m}s + 2dv\right)\Delta t \cdot \tag{2.34}$$

Für hinreichend kleine Differenzen Δt bekommt man so eine angenäherte Lösung Δv.

Tabelle 2-9 Struktogramm – Simulation einer freien gedämpften Schwingung

Eingabe m, f, d, s_0, v_0, t_0, d_t, t_e	
	Solange $t < t_e$
	$\Delta v_i = -\left(\frac{f}{m}s_i + 2dv_{i-1}\right)\Delta t \cdot$
	$v_i = v_{i-1} + \Delta v_i$
	$\Delta s_i = v_i \cdot \Delta t$
	$s_i = s_{i-1} + \Delta s_i$
	$t_i = t_{i-1} + \Delta t$
	Ausgabe s_i, v_i, t_i

Code 2-20 Anwendungsbeispiel Deterministische Simulation – Freie gedämpfte Schwingung

```
/* FreieSchwingung.cpp
   Das Programm simuliert den
   Bewegungsverlauf einer
   freien gedämpften Schwingung
*/

#include <iostream.h>
using namespace std;

int main ()
{
   double m, f, d;
   double s, v, t, dt, ds, dv, te;
   FILE *puffer;
   puffer = fopen("C:\\Temp\\Schwingung.dat","w");

// Eingaben
   cout << "Masse m [kg] = ";
   cin >> m;
   cout << "Federkonstante f [kg/s^2] = ";
   cin >> f;
```

```
    cout << "Dämpfungskonstante d [1/s] = ";
    cin >> d;
    cout << "Ausgangsposition s0 [m] = ";
    cin >> s;
    cout << "Ausgangsgeschwindigkeit v0 [m/s] = ";
    cin >> v;
    cout << "Ausgangszeit t [s] = ";
    cin >> t;
    cout << "Schrittweite dt [s] = ";
    cin >> dt;
    cout << "Endzeit te [s] = ";
    cin >> te;

// Analyse
    do {
       dv = -(f/m*s+2*d*v)*dt;
       v = v+dv;
       ds = v*dt;
       s = s+ds;
       t = t+dt;
       cout << t << v << s << endl;
    // Ausgabe in eine Datei
       fprintf(puffer,"%4.1e\t%4.1e\t%4.1e\n", t, v, s);
    } while (t<te);

    fclose(puffer);
    system("Pause");
}
```

Die nachfolgenden Beispielsdaten dienen zur Überprüfung des korrekten Programmablaufs.

```
Masse m [kg] = 50
Federkonstante f [kg/s^2] = 80
Dämpfungskonstante d [1/s] = 0.4
Ausgangsposition s0 [m] = -5
Ausgangsgeschwindigkeit v0 [m/s] = 0
Ausgangszeit t [s] = 0
Schrittweite dt [s] = 0.1
Endzeit te [s] = 10
```

Bild 2-30 Beispieldaten einer freien gedämpften Schwingung

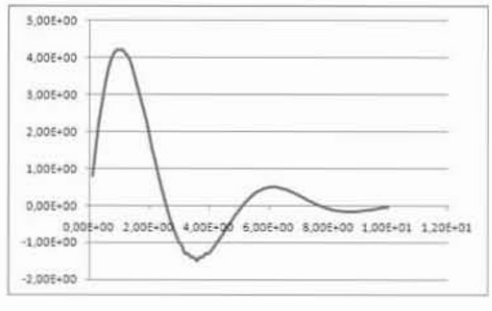


Bild 2-31 v-t Diagramm der Beispieldaten

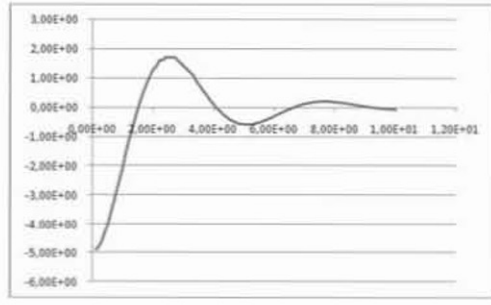


Bild 2-32 s-t Diagramm der Beispieldaten

Übungen

Ersetzen Sie die Bestimmung der inkrementellen Wegänderung durch einen Funktionsaufruf etwa in der Form:

```
   double (*fS) (double, double, double, double, double);

// freie gedämpfte Schwingung
   double fFGS (double f, double m, double s, double d, double v)
      return (f/m*s+2*d*v);

// Analyse

   fS = fFGS;
   do {
      dv = - fS(f,m,s,d,v)*dt;
      v = v+dv;
      ds = v*dt;
      s = s+ds;
      t = t+dt;
      cout << t << v << s << endl;
   // Ausgabe in eine Datei
      fprintf(puffer,"%4.1e\t%4.1e\t%4.1e\n", t, v, s);
   } while (t<te);
```

Dadurch sind Sie in der Lage, unterschiedliche Schwingungsformen für einen Berechnungsablauf zu wählen. So ist die Annahme einer linearen Federkennlinie nicht immer ausreichend genau. In

$$F_f = f \cdot s\left(1 + c \cdot s^2\right) \tag{2.35}$$

überlagert eine kubische Parabel die lineare Federkennlinie. Die Dämpfungskraft lässt sich auch als Newton'sche Reibung, also dem Quadrat der Geschwindigkeit proportional ansetzen:

$$F_d = c \cdot \mathrm{sgn}(v) \cdot v^2 \tag{2.36}$$

Schwingungen können auch durch rotierende Massen erzeugt werden.

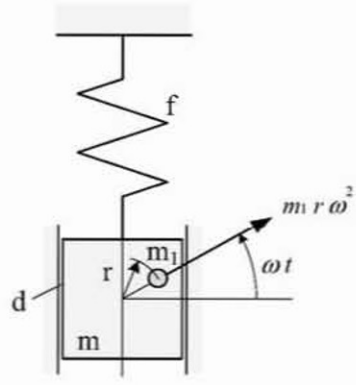


Bild 2-33 Erzwungene Schwingung durch eine rotierende Masse

Die im Abstand r außerhalb des Drehpunktes mit der Winkelgeschwindigkeit ω rotierende Masse m_1 hat in Schwingungsrichtung den Fliehkraftanteil

$$F_e = m_1 \cdot r \cdot \omega^2 \cdot \sin(\omega \cdot t) \tag{2.37}$$

Der Ansatz gestaltet sich wie zuvor mit

$$m\ddot{s} = -f \cdot s - 2 \cdot (m + m_1) \cdot d \cdot v - m_1 \cdot r \cdot \omega^2 \cdot \sin(\omega \cdot t). \tag{2.38}$$

wobei m die Masse m_1 beinhaltet. Es folgt weiter

$$\Delta v = -\left(\frac{f}{m} s + 2dv - \frac{m_1}{m} r\omega^2 \sin(\omega t)\right) \Delta t \cdot \tag{2.39}$$

Diese Gleichung ist für eine konstante Winkelgeschwindigkeit ausgelegt. Ansonsten muss der Algorithmus eine zusätzliche Gleichung ω = f(t) bestimmen.

Ein weiterer Anwendungsfall ist die erzwungene Schwingung mit rotierender und oszillierender Masse, wie sie in **Bild 2-34** dargestellt ist. Als Ansatz betrachten wir darin einen idealisierten Schubkurbeltrieb. Die vorhandenen Massenkräfte für rotierende und oszillierende Massen werden auch als Massenkräfte erster und zweiter Ordnung bezeichnet.

Ihr Anteil in Schwingungsebene beträgt für die Massenkraft erster Ordnung angenähert

$$F_I = m_1 \cdot r \cdot \omega^2 \cdot \cos(\omega \cdot t) \tag{2.40}$$

und für die Massenkraft zweiter Ordnung

$$F_{II} = m_1 \cdot r \cdot \omega^2 \cdot \frac{r}{l} \cos^2(\omega \cdot t) \tag{2.41}$$

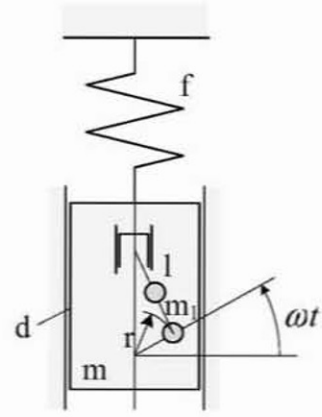


Bild 2-34 Erzwungene Schwingung durch rotierende und oszillierende Massen

Die Differentialgleichung der Bewegung lautet

$$m\ddot{s} = -\left(f \cdot s + 2 \cdot m \cdot d \cdot v - m_1 \cdot r \cdot \omega^2 \left(\cos(\omega \cdot t) - \frac{r}{l}\cos^2(\omega \cdot t)\right)\right) \tag{2.42}$$

Aus dieser leitet sich wieder die nachfolgende Geschwindigkeitsänderung ab

$$\Delta v = -\left(\frac{f}{m}s + 2dv - \frac{m_1}{m}r\omega^2\left(\cos(\omega t) - \frac{r}{l}\cos^2(\omega t)\right)\right)\Delta t \tag{2.43}$$

Untersuchen Sie zum Schluss noch den Fall, dass die Federmasse mit berücksichtigt wird.

Setzen Sie alle diese Schwingungsformen als Funktionen ein und schaffen Sie eine Auswahlmöglichkeit.

2.6 Callback-Funktionen

Eine Callback-Funktion nennt man eine Funktion, die einer anderen Funktion als Parameter übergeben wird, und von dieser dann aufgerufen wird. Das erlaubt es, Funktionen allgemein zu definieren und erst beim Aufrufen der Funktion durch Angabe der Callback-Funktion die Handlungsweise genau zu bestimmen. Häufig wird auch die als Callback benutzte Funktion nicht fest definiert, sondern wird als so bezeichnete anonyme Funktion zur Laufzeit definiert.

Hier ein kurzes Beispiel für die Verwendung von Callback-Funktionen. Ein Integer-Feld soll auf unterschiedliche Weise sortiert werden. Dazu müssen zwei Werte in einer Funktion irgendwie miteinander verglichen werden.

Beispiel: Ein Feld mit n Elementen soll unterschiedlich sortiert werden

```
int a[] = {0,11,23,9,34,5,7,28,12,9,13,15};
```

Für die Sortierung muss ein Vergleich zwischen zwei Feldelementen durchgeführt werden.

Beispiel: Allgemeine Form einer Vergleichsfunktion

```
Vergl (int a[i], int a[j])
```

Diese wird in einer Sortierfunktion, quasi als Platzhalter, eingesetzt.

Beispiel: Sortierfunktion

```
void Sort_a (CompFunc Vergl) {
   int m, t;
   int Anz;
   Anz = sizeof(a)/sizeof(int);
   for (int i=0; i<Anz; ++i) {
      m = i;
      for (int j=i+1; j<Anz; ++j)
         if (Vergl (a[j], a[m])) m =j;
      t = a[i];
      a[i] = a[m];
      a[m] = t;
   }
}
```

Damit der Compiler die Vergleichsfunktion akzeptiert, muss sie als Parameter an die Sortierfunktion übergeben werden, außerdem muss der Typ der Vergleichsfunktion deklariert werden.

Beispiel: Typ der Vergleichsfunktion

```
typedef bool CompFunc (int, int);
```

Es fehlen noch die eigentlichen Sortierfunktionen, die natürlich vom gleichen Typ sein müssen.

Beispiel: Zwei Vergleichsfunktionen vom Typ CompFunc

```
bool Kleiner  (int a, int b) {return a<b; }
bool Groesser (int a, int b) {return a>b; }
```

Jetzt kann der Aufruf der Sortierfunktion nach unterschiedlichen Methoden erfolgen. Nachfolgend das gesamte Programm.

Code 2-21 Anwendungsbeispiel Callback-Funktionen – Sortieralgorithmus

```
/* Callback.cpp
   Sortierprogramm mit
   callback-Funktionen
*/
#include <iostream.h>

// Typ der Vergleichsfunktionen
typedef bool CompFunc (int, int);

// Zwei Vergleichsfunktionen (vom Typ CompFunc):
bool Kleiner  (int a, int b) {return a<b; }
bool Groesser (int a, int b) {return a>b; }

void Sort_a (CompFunc vergl); // Deklaration der Sortierfunktion

int a[] = {0,11,23,9,34,5,7,28,12,9,13,15};

int main ()
{
   int i;
   int Anz;
   Anz = sizeof(a)/sizeof(int);
// a aufsteigend sortieren
   Sort_a (Kleiner);
// Ausgabe
   for (i=0;i<Anz;i++) cout << a[i] << " ";
   cout << endl;

// a absteigend sortieren
   Sort_a (Groesser);
// Ausgabe
   for (i=0;i<Anz;i++) cout << a[i] << " ";
   cout << endl;

   system("pause");
}

// Definition der Sortierfunktion
void Sort_a (CompFunc Vergl) {
   int m, t;
   int Anz;
   Anz = sizeof(a)/sizeof(int);
   for (int i=0; i<Anz; ++i) {
      m = i;
      for (int j=i+1; j<Anz; ++j)
         if (Vergl (a[j], a[m])) m =j;
      t = a[i];
      a[i] = a[m];
      a[m] = t;
   }
}
```

```
0 5 7 9 9 11 12 13 15 23 28 34
34 28 23 15 13 12 11 9 9 7 5 0
Drücken Sie eine beliebige Taste . . .
```

Bild 2-35 Programmausgabe – Sortierung mit Callback-Funktionen

Anwendungsbeispiel: Massenträgheitsmomente – Verschiebesatz von Steiner

Die beschleunigte Drehung eines starren Körpers in der Ebene um eine feste Achse (Rotation) wird durch die Einwirkung eines Drehmoments M hervorgerufen. Dabei vollführt jedes Massenteilchen dm eine beschleunigte Bewegung. Aus der Kinematik ist bekannt, dass ein Massenteilchen auf gekrümmter Bahn einer Normal- und einer Tangentialbeschleunigung unterliegt.

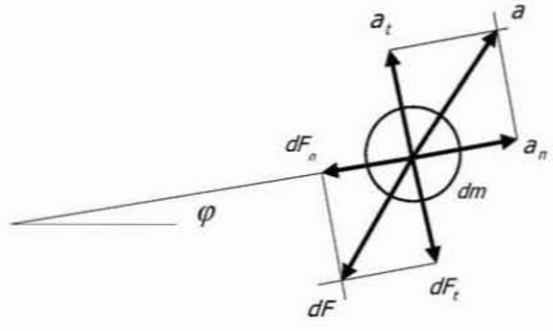


Bild 2-36 Beschleunigte Drehung

Dies führt nach dem d'Alembertschen Prinzip zu dem Ansatz

$$dF_t = dm \cdot a_t \tag{2.44}$$

und

$$dF_n = dm \cdot a_n . \tag{2.45}$$

Während das Normalkraftdifferential dFn kein Drehmoment hervorruft, seine Wirkung geht durch den Drehpunkt, ruft das Tangentialkraftdifferential dFt einen Drehmomentanteil von

$$dM = r \cdot dF_t \tag{2.46}$$

hervor. Für die Gesamtheit aller Anteile gilt damit

$$M = \int r \cdot dF_t = \int r \cdot dm \cdot a_t \tag{2.47}$$

mit

$$a_t = r \cdot \varepsilon . \tag{2.48}$$

Darin ist ε die Winkelbeschleunigung, die für alle Masseteile gleich ist, also

$$M = \varepsilon \int r^2 \cdot dm . \tag{2.49}$$

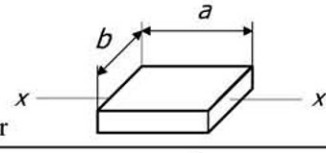
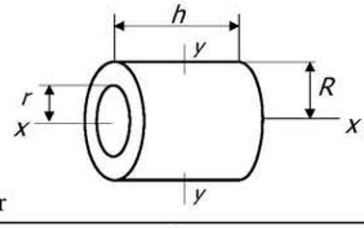
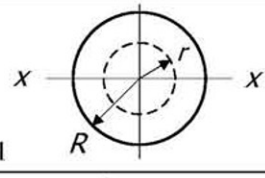
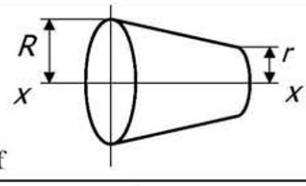
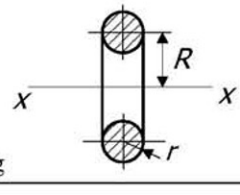
Analog zur Massenträgheit bei der Translation ($F = m \cdot a$), bezeichnet man

$$I_d = \int r^2 \cdot dm \tag{2.50}$$

als Massenträgheitsmoment eines starren Körpers. Genauer, da es sich auf eine Achse bezieht, als axiales Massenträgheitsmoment. Zu beachten ist, dass das Quadrat des Abstandes in die Gleichung eingeht.

Die nachfolgende Tabelle zeigt eine Zusammenstellung der Massenträgheitsmomente einfacher Grundkörper. Auch komplizierter gestaltete Körper lassen sich mit diesen Gleichungen berechnen, da die Summe der Massenträgheitsmomente einzelner Grundkörper gleich dem Massenträgheitsmoment des aus diesen bestehenden starren Körpers ist. Dies liegt in der Eigenschaft des Integrals in Gleichung 2.50.

Tabelle 2-10 Axiale Massenträgheitsmomente

Körper	Massenträgheitsmoment
 Quader	$I_{dx} = \frac{m}{12}\left(a^2 + b^2\right)$
 Hohlzylinder	$I_{dx} = \frac{m}{2}\left(R^2 + r^2\right)$ $I_{dy} = \frac{m}{4}\left(R^2 + r^2 - \frac{h^2}{3}\right)$
 Hohlkugel	$I_{dx} = 0{,}4m\frac{R^5 - r^5}{R^3 - r^3}$
 Kegelstumpf	$I_{dx} = 0{,}3m\frac{R^5 - r^5}{R^3 - r^3}$
 Kreisring	$I_{dx} = m\left(R^2 + \frac{3}{4}r^2\right)$

Nicht immer fällt die Drehachse des Grundkörpers mit der des starren Körpers zusammen. Dazu betrachten wir nach **Bild 2-37** das Massenträgheitsmoment bezüglich einer zweiten Achse gegenüber der Schwerpunktachse.

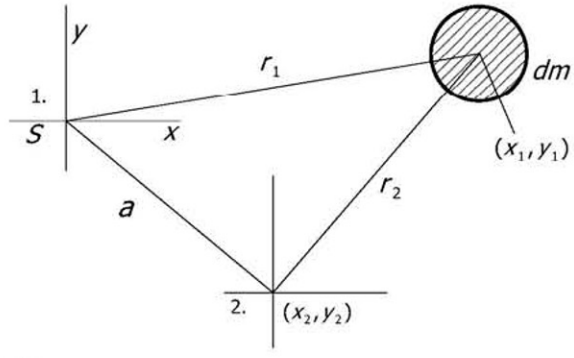


Bild 2-37 Achsenverschiebung

Es gilt für den Radius r_2 die geometrische Beziehung

$$\begin{aligned} r_2^2 &= (x_1 - x_2)^2 + (y_1 - y_2)^2 \\ &= x_1^2 - 2x_1x_2 + x_2^2 + y_1^2 - 2y_1y_2 + y_2^2 \\ &= r_1^2 + a^2 - 2(x_1x_2 + y_1y_2) \end{aligned} \tag{2.51}$$

Die eingesetzt in 2.51

$$I_d = \int r_1^2 \cdot dm + a^2 \int dm - 2\int (x_1x_2 + y_1y_2)\,dm \tag{2.52}$$

ergibt. Darin ist

$$\int dm = m$$

und

$$-2\int (x_1x_2 + y_1y_2)\,dm$$

das statische Moment des starren Körpers bezüglich des Schwerpunkts, also Null. Damit folgt die, als Satz von Steiner (Verschiebungssatz) bekannte Gesetzmäßigkeit

$$Id_2 = Id_1 + m \cdot a^2 . \tag{2.53}$$

Nun können wir die in der Tabelle angegebenen Formeln programmieren und mit Hilfe des Steiner'schen Satzes auf jede beliebige Achse umrechnen.

Code 2-22 Anwendungsbeispiel – Massenträgheitsmomente finiter Körper

```
/* Massen.cpp
   Bestimmung von axialen
   Massenträgheitsmomenten
*/
```

```
#include <iostream.h>

// Typ der Vergleichsfunktionen
typedef double Koerper (double, double, double);

// Massenträgheitsmomente (vom Typ Koerper):
double Quader  (double m,double a, double b)
{ return m/12*(a*a+b*b);}
double Zylinder (double m,double R, double r)
{ return m/2*(R*R+r*r);}
double Kugel (double m,double R, double r)
{ return 0.4*m*(R*R*R*R*R+r*r*r*r*r)/(R*R*R-r*r*r);}
double Kegel (double m,double R, double r)
{ return 0.3*m*(R*R*R*R*R-r*r*r*r*r)/(R*R*R-r*r*r);}
double Ring (double m,double R, double r)
{ return m*(R*R+3/4*r*r);}

// Deklaration der callback Funktion
double Idx (Koerper Id, double m, double x, double y);


int main ()
{
   char weiter;
   double a, m, x, y;
   double Id1, Id2;

   do
   {
      cout << "weiter (E=Ende/Q/Z/H/K/R=Berechnung) ";
      cin >> weiter;
      switch (weiter)
      {
      case 'E': /* Ende */
         break;
      case 'Q': /* Quader */
      case 'Z': /* Zylinder */
      case 'H': /* Hohlkugel */
      case 'K': /* Kegel */
      case 'R': /* Ring */
      // Eingaben
         cout << "Masse [kg] = ";
         cin >> m;
         cout << "Abstand [mm] = ";
         cin >> a;
         cout << "Koerpermass x [mm] = ";
         cin >> x;
         cout << "Koerpermass y [mm] = ";
         cin >> y;
         switch (weiter)
         {
         case 'Q':
            Id1 = Idx (Quader, m, x, y);
            break;
```

```
            case 'Z':
               Id1 = Idx (Zylinder, m, x, y);
               break;
            case 'H':
               Id1 = Idx (Kugel, m, x, y);
               break;
            case 'K':
               Id1 = Idx (Kegel, m, x, y);
               break;
            case 'R':
               Id1 = Idx (Ring, m, x, y);
               break;
            }
            Id2 = Id1+m*a*a;
            printf ("\nId1 = %8.2f   Id2 = %8.2f\n\n",Id1,Id2);
            break;
         default:
            cout << "Falsche Eingabe!" << endl;
         }
      }
      while (weiter!='E');
}

double Idx (Koerper Id, double m, double x, double y)
{
   return Id(m, x, y);
}
```

Das nachfolgende **Bild 2-38** zeigt die Ergebnisse von Testberechnungen.

Übungen

Erstellen Sie nach dem gleichen Schema ein Programm zur Volumenberechnung nach finiten Elementen. Die Grundlage ist die Überlegung, dass sich komplexe Maschinenteile in einfache Grundkörper zerlegen lassen. Wichtig für die sichere Handhabung ist, dass einfache und wenige Grundkörper benutzt werden.

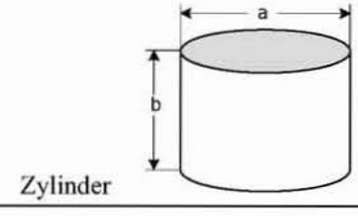
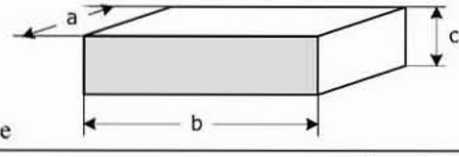
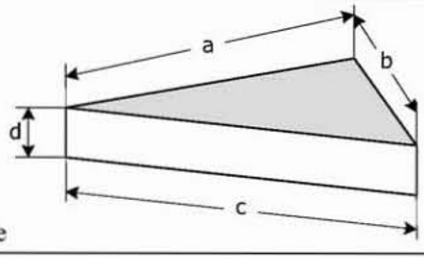
Die Zerlegung eines Maschinenelements in diese Grundkörper kann im positiven wie im negativen Sinne gesehen werden. Ein positives Element ist z. B. eine Welle, ein negatives Element eine Bohrung. Es können auch mehrere gleiche Grundelemente auftreten.

Bauen Sie dieses Programm nach seiner Fertigstellung weiter aus. Mit der zusätzlichen Angabe der Materialdichte lässt sich das Programm zur Gewichtsberechnung nutzen. Mit einer weiteren Kennung, der Baugruppennummer, kann die Gewichtsberechnung auf Gruppen- und Gesamtgewichte ausgedehnt werden.

```
weiter (E=Ende/Q/Z/H/K/R=Berechnung) Q
Masse m [kg] = 100
Abstand a [mm] = 50
Koerpermass x [mm] = 200
Koerpermass y [mm] = 120

Id1 = 453333.33   Id2 = 703333.33

weiter (E=Ende/Q/Z/H/K/R=Berechnung) Z
Masse m [kg] = 100
Abstand a [mm] = 50
Koerpermass x [mm] = 200
Koerpermass y [mm] = 120

Id1 = 2720000.00   Id2 = 2970000.00

weiter (E=Ende/Q/Z/H/K/R=Berechnung) H
Masse m [kg] = 100
Abstand a [mm] = 50
Koerpermass x [mm] = 200
Koerpermass y [mm] = 120

Id1 = 2199510.20   Id2 = 2449510.20

weiter (E=Ende/Q/Z/H/K/R=Berechnung) K
Masse m [kg] = 100
Abstand a [mm] = 50
Koerpermass x [mm] = 200
Koerpermass y [mm] = 120

Id1 = 1411591.84   Id2 = 1661591.84

weiter (E=Ende/Q/Z/H/K/R=Berechnung) R
Masse m [kg] = 100
Abstand a [mm] = 50
Koerpermass x [mm] = 200
Koerpermass y [mm] = 120

Id1 = 4000000.00   Id2 = 4250000.00
```

Bild 2-38 Testdaten und deren Ergebnisse

Tabelle 2-11 Finite Elemente zur Volumenberechnung

 Zylinder	$V = \frac{\pi}{4} a^2 b$
 Rechteckplatte	$V = a \cdot b \cdot c$
 Dreieckplatte	$V = \sqrt{s(s-a)(s-b)(s-c)}$ $s = \frac{a+b+c}{2}$