

VORLESUNGSMITSCHRIEB
PROGRAMMIEREN FÜR PHYSIKER
PROF. STEINHAUSER

WINTERSEMESTER 2005 / 2006 (UNI KARLSRUHE)

Magnus Schlösser

23. Oktober 2006

Inhaltsverzeichnis

I Die Sprache C++	5
1 Einleitung	5
2 Grundlagen	6
2.1 Das erste Programm	6
2.2 Von der Hochsprache zum ausführbaren Programm	6
2.3 Standardein-/ausgabe mit <code>cin</code> , <code>cout</code>	7
2.4 Ein-/Ausgabe in Dateien	7
2.4.1 Ausgabe in Dateien	7
2.4.2 Eingabe in Dateien	7
3 Wichtige Datentypen	8
3.1 einfache Typen	8
3.2 Felder	8
4 Operatoren	9
4.1 arithmetische und logische Operatoren	9
4.2 Inkrement/Dekrementoperatoren (<code>++</code> , <code>--</code>)	9
4.3 Zusammengesetzte Zuweisungsoperatoren	9
4.4 Kommaoperator „ <code>,</code> “	10
4.5 C und C++	10
5 Kontrollstrukturen	11
5.1 Schleifenanweisungen	11
5.1.1 <code>for</code> -Schleife	11
5.1.2 <code>while</code> -Schleife	11
5.1.3 <code>do...while</code> -Schleife	12
5.2 Verzweigungen	12
5.2.1 Die <code>if</code> -Anweisung	12
5.2.2 <code>switch</code> -Anweisung	13
5.2.3 <code>break</code> , <code>continue</code>	13
6 Datentypen	14
6.1 Elementare Datentypen	14
6.2 Variablen und Konstanten	14
6.2.1 Konstanten	14
6.3 Datentypumwandlungen (Konversion)	14
6.3.1 Implizite Konversion (automatische Konversion)	15
6.3.2 Explizite Konversion (<code>cast</code>)	15
7 Felder und Strukturen	16
7.1 Felder	16
7.1.1 Eindimensionale Felder	16
7.1.2 Mehrdimensionale Felder	16
7.2 Strukturen	16
7.3 Unions	17
7.4 Aufzählungstyp	17
7.5 Namen für Typen: <code>typedef</code>	17

8 Funktionen	19
8.1 Funktionen	19
8.1.1 Deklaration einer Funktion (Funktionsprototyp)	19
8.1.2 Definition der Funktion	20
8.1.3 Aufruf der Funktion	20
8.1.4 Ergänzungen	20
8.2 Methoden	21
8.3 Rekursion und Iteration	21
8.3.1 Rekursion	21
8.3.2 Iteration	21
8.4 Wert- und Referenzparameter	22
8.4.1 bisher: Werteparameter	22
8.4.2 Referenzparameter	22
8.4.3 Zusammenfassung	22
8.4.4 Felder als Funktionsparameter	23
8.5 Überladung von Funktionen und Operatoren	23
8.5.1 Funktionen	23
8.5.2 Operatoren	23
8.6 Kommandozeilenparameter	23
9 Klassen und Objekte	25
9.1 Motivation	25
9.2 Klassen in C++	25
9.3 Freunde (friends)	26
9.3.1 Funktionen als Freunde	26
9.4 Konstruktoren und Destruktoren	27
9.4.1 Initialisierung mit Konstruktoren	27
9.4.2 Destruktoren	28
9.5 Vererbung	28
9.5.1 Prinzip	28
9.5.2 Sichtbarkeit bei Vererbung	29
9.5.3 Vererbungstypen	29
9.6 Schablonen	29
9.6.1 Funktionsschablonen	29
9.6.2 Klassenschablonen	30
9.7 STANDARD TEMPLATE LIBRARY (STL)	30
10 Pointer (Zeiger)	31
10.1 Motivation und Notation	31
10.2 Zeiger, Arrays, Strukturen und Funktionen	32
10.2.1 Zeiger und Arrays	32
10.2.2 Übergabe von Arrays an Funktionen	33
10.2.3 Zeiger auf Funktionen	33
10.2.4 Zeiger auf Strukturen	33
10.3 Dynamische Speicherverwaltung	33
II Mathematische Einschübe	35
1 Lineare Gleichungssysteme (LGS)	35
1.1 Motivation	35
1.2 Schritte	35
1.3 Verfahren von Gauss-Jordan	36

2	Interpolation	37
2.1	Motivation	37
2.2	Polynom-Interpolation	37
3	Numerische Integration	39
3.1	NEWTON-COTES-Formeln	39
3.1.1	Methode zur Gewinnung der Integrationsformeln	39
3.1.2	Beispiele	40
3.1.3	Fehlerabschätzung bei NC-Formeln	41
3.1.4	Allgemeine Trapez- und Simpsonregel	41
4	Gewöhnliche Differentialgleichungen	44
4.1	Problemstellung	44
4.2	Polynomzug-Verfahren (EULER)	44
4.3	RUNGE-KUTTA-VERFAHREN	45

Teil I

Die Sprache C++

1 Einleitung

Rechner

- **Hardware**
- **Software**
 - Betriebssystem
 - Anwenderprogramme
 - Programmiersprache (FORTRAN, PASCAL, C, C++)

Programmiersprachen

- **Maschinensprache** („0“, „1“)
- **Assembler** (add i #12, d0)
- **Hochsprache** (Befehle, ähnlich der Umgangssprache und mit Compiler in Maschinensprache übersetzt)

Leistungsmerkmale von C/C++

- schnelle Laufzeit
- gute Portabilität
- geringer Sprachumfang
- ermöglicht objektorientiertes Programmieren
- Überladen von Funktionen, Klassen und Operatoren möglich
- Templates (Schablonen)

2 Grundlagen

2.1 Das erste Programm

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

int main()
{
    cout << "Hallo Karlsruhe!" << endl;
    return( 0 );
}
```

1. `hallo.cc` abspeichern
2. compilieren mit `g++ -o hallo hallo.cc`
3. ausführen mit `hallo` oder `./hallo`

Bemerkungen

1. `#include ...`
kein C++ Befehl, sondern Anweisung für Präprozessor
Einlesen von Dateien
`iostream`: Eingabe und Ausgabe
`cmath`: Mathematische Funktionen
2. `using namespace std;`
in C++ gehören alle Variablen und Funktionen zu einem Namesraum. Standardobjekte sind im Namesraum „`std`“ definiert.
3. `int main(){ ... }`
definiert die Funktion `main()` muß in jedem Programm genau einmal vorkommen und wird beim Programmstart ausgeführt.
4. Unterscheidung zwischen Groß- und Kleinschreibung (**case-sensitive**)
5. Kommentare:
// : 1 Zeile
/* ... */ : mehrere Zeilen
6. „;“ am Ende von Befehlen
7. { ... } definiert Block \equiv Zusammenfassung von Anweisungen (kein „;“ am Ende)
8. Namen (Bezeichner) werden aus Buchstaben, Ziffern und „_“ aufgebaut.

2.2 Von der Hochsprache zum ausführbaren Programm

1. **Idee, Analyse** (Modell, Algorithmus)
2. **Programmwurf** (Flußdiagramm, welche Variable, Klassen ...)
3. **Quelltext erstellen**
4. **Lauffähiges Programm** (Maschinencode)

Beispiel Umrechnung von FAHRENHEIT in CELSIUS

1. Lese Zahl ein. Interpretiere sie als Temperatur in Grad-FAHRENHEIT. Berechne den entsprechenden Wert in CELSIUS. Gebe Ergebnis aus:

$$T_C = \frac{5}{9}(T_F - 32)$$

2. (a) lies Wert von T_F in Variable `tf`
 (b) berechne T_C : `tc = 5 * (tf - 32) / 9`
 (c) gib `tf` und `tc` aus

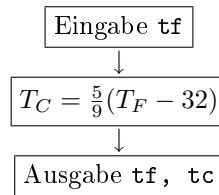


Abbildung 1: Flußdiagramm

3. `g++ -o fah2cel fah2cel.cc`

2.3 Standardein -/ausgabe mit cin, cout

- Ein -/Ausgabe mittels Datenströmen (`streams`)
- „<<“ ; „>>“ Schiebeoperatoren; geben Flußrichtung an
- `endl`; Zeilenumbruch. Auch: `"\n"` möglich
- Formatierung der Ausgabe

2.4 Ein -/Ausgabe in DateienPrinzip

1. öffne Datenstrom
2. lese/schreibe
3. schließe Datei

2.4.1 Ausgabe in Dateien

```

ofstream ausgabe;
ausgabe.open("daten.txt");
ausgabe << daten << endl;
ausgabe.close();
  
```

2.4.2 Eingabe in Dateien

```

ifstream einlesen;
einlesen.open("daten.txt");
einlesen >> x2 >> y2;
einlesen.close();
  
```

3 Wichtige Datentypen

3.1 einfache Typen

- `int` Integer (ganze Zahlen) 32bit $-2^{31}, \dots, 2^{31} - 1$
- `float` Gleitkommatypen 32bit $\pm 1, 17 \cdot 10^{-38} \dots, \pm 3, 4 \cdot 10^{-38}$
- `char` Zeichen 8bit

Syntax: `typ Variablenamen;`

Bsp: `char ch`

3.2 Felder

Syntax: `Typ Arrayname[Größe1] ... [Größe2];`

Bsp:

- `int vek[3];` \iff 3-dim Vektor
- `int mat[4][3];` \iff Matrix mit 4 Zeilen und 3 Spalten

`array1.cc`

Achtung: Indexbereich von 0 bis „Dimension-1“!!!

4 Operatoren

4.1 arithmetische und logische Operatoren

Ein Operator verknüpft Teilausdrücke zu einem neuen Gesamtausdruck.
Operatoren können logische oder arithmetisch (Eingabe-) Argumente haben.
Operatoren können logische oder arithmetisch (Ausgabe-) Werte haben.

Bsp:

- arithmetische Argumente und arithmetische Werte:
+ , - , * , / , %
% ist der Modulooperator (Rest der ganzzahligen Division)
- arithmetische oder logische Argumente und logische Werte:
== , < , > , <= , >= , !=
- logische Argumente und logische Werte
|| (oder) , && (und)

Priorität:

Jedem Operator ist eine Priorität zugeordnet, die festlegt wie ein (nicht vollständig geklammerter) Ausdruck ausgewertet wird.

Beispiel:

`a * b + c * d = (a * b) + (c * d)`

4.2 Inkrement/Dekrementoperatoren (++ , --)

```
int i;
++i; //i=i+1
--i; //i=i-1
i++;
i--;
```

Pre-In(De)krementoperator

..1.. erhöhen
..2.. auswerten

Post-In(De)krementoperator

..1.. auswerten
..2.. erhöhen

Beispiel:

```
int i,j;
i=7;
j=++i; //→j=8, i=8
i=7;
j=i++; //→j=7, i=8
```

4.3 Zusammengesetzte Zuweisungsoperatoren

+ = , - = , * = (und)
int i,j;
i += 5;
j *= i + 3;

4.4 Kommaoperator “,”

dient zur Trennung von mehreren Anweisungen, die an der selben Stelle im Programm ausgeführt werden sollen.

Bsp:

```
int a, b, c;  
a = ( b = 8 , c = 3 , b - c ) - 1;  
Bitte nicht so programmieren!
```

4.5 C und C++

In der Regel: „C“ Teilmenge von „C++“

Neu in C++

- //-Kommentare
- Deklaration von Variablen an beliebiger Stelle
- Ein/Ausgabe
C: `printf`, `scanf`
C++: `cout`, `cin`
- Variablentyp `bool`
- Klassen
- Überladen von Operatoren

5 Kontrollstrukturen

5.1 Schleifenanweisungen

5.1.1 for-Schleife

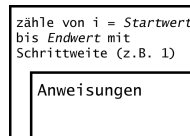


Abbildung 2: Struktogramm der for-Schleife

1. Syntax

```
for(ausdruck1; ausdruck2; ausdruck3)
{
    Anweisungen
}
```

- (a) Initialisierung
- (b) Bedingung
- (c) wird im Anschluß an "Anweisungen" ausgeführt

2. Bsp:

for_2.cc

```
int i;
for( i = 1; i <= 100 ; i++ )
{
    cout << i << " ";
}
```

3. Bemerkung: for(;;) Endlosschleife

5.1.2 while-Schleife

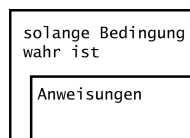


Abbildung 3: Struktogramm der while-Schleife

1. Syntax

```
while( ausdruck )
{
    Anweisungen
}
```

- 2. (a) Jede for-Schleife kann als while-Schleife formuliert werden.
- (b) Im Allgemeinen:
 - for-Schleife falls Anzahl der Schleifendurchläufe bekannt
 - while-Schleife falls Anzahl der Schleifendurchläufe NICHT bekannt.
- (c) Wird "ausdruck" in Schleife nicht manipuliert, so ergibt sich eine Endlosschleife.
- (d) Abweïschleife

while_for.cc

5.1.3 do...while-Schleife

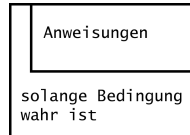


Abbildung 4: Struktogramm der do-while-Schleife

1. Syntax

```
do
{
    Anweisungen;
}while( ausdruck );
```

do_while.cc

2. Bemerkungen:

Anweisungen werden mind. 1x ausgeführt.

5.2 Verzweigungen

5.2.1 Die if-Anweisung

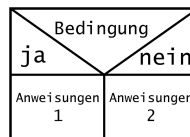


Abbildung 5: Struktogramm der if-Anweisung

1. Struktogramm

- (a) Berechne „Ausdruck“
- (b) true → Anweisung 1
false → Anweisung 2

2. Syntax

```
if( Ausdruck )
{
    Anweisungen1;
}
else
{
    Anweisungen2;
}
```

if_1.cc

3. Bemerkungen

- (a) else ist nicht notwendig
- (b) if-else-Zuordnung: aufpassen bei geschachtelten if's

- (c) Ausdruck:
 - a=2 Zuweisung
 - a==2 logische Bedingung
- (d) Fragezeichenoperator
 - i. Syntax: Bedingung ? Erg1 : Erg2;
Falls „Bedingung“ wahr, wird Erg1 zurück gegeben, sonst Erg2
 - ii. Bsp:


```
float a,b,erg;
...
erg = ( a > b ? a : b); // gibt den größeren Wert zurück
```

5.2.2 switch-Anweisung

1. Struktogramm
Ausdruck entspricht Schalter. Ergebnis von Ausdruck muss vom Typ Integer sein.

2. Syntax

```
switch( Ausdruck )
{
    Fall1 :
        Anweis1;
        break;
    Fall2 :
        Anweis2;
        break;
    default :
        Anweis;
}
```

switch_1.cc

3. Bemerkungen

- (a) „default“ ist optional
- (b) „break“ nicht notwendig

5.2.3 break, continue

1. break : bricht Schleife ab
2. continue : unterbricht Abarbeitung der Anweisungen
→Berechnung der Ausdrucksbedingung
→neuer Durchlauf
Kann bei switch, for, while, do ... while verwendet werden.

3. Prinzip

```
while( ausdruck )
{
    ...
    continue; //geht sofort zurück zu while
    ...
    break; //bricht die while-Schleife sofort ab.
}
```

6 Datentypen

6.1 Elementare Datentypen

(vordefiniert, keine Struktur)

bool true, false	char wchar_t Zeichen	short int long ganze Zahlen	float double long double gebrochenen Zahlen
---------------------	-------------------------	--------------------------------	--

- genaue Art der internen Zahldarstellung hängt von Hardware und Compiler ab.
- Gültigkeitsbereich reicht bis zum Ende des Blocks (Scope)

scope.cc

Bsp:

```
int g1 = 100;
int main()
{
    int l1=50;
    {
        int l2=10;
    }
}
```

- Deklaration u. Definition können gleichzeitig erfolgen.
- Integertypen können als `unsigned` definiert werden.

Bsp:

```
short: -32768...+32767 = 215 - 1
unsigned short: 0...+65535 = 216 - 1 (=2 Byte)
```

unsign_short.cc

- `sizeof()`

Liefert Speicherbedarf eines Datentyps in Bytes

Bsp:

```
sizeof(char)=1
sizeof(float)=4
sizeof(double)=8
sizeof(long double)=12
```

sizeof.cc

- Wertebereich

limits.cc

6.2 Variablen und Konstanten

6.2.1 Konstanten

Feststehende Werte können am Anfang als Konstante definiert werden. Im Programm kann dieser Wert nicht mehr geändert werden.

Bsp:

```
const float pi = 3.14159;
:
pi=5.4; // <- Fehlermeldung
```

6.3 Datentypumwandlungen (Konversion)

Treffen in einem Ausdruck Variablen unterschiedlichen Typs aufeinander. „Mix von Typen“

Bsp

```
int i='0'; // → i=48
```

Frage: *Was passiert?*

Was ist erlaubt?

Was ist gefährlich?

6.3.1 Implizite Konversion (automatische Konversion)

1. Zwischen integralen Typen
einfach für Compiler; bei unterschiedlicher Größe
 - fülle mit Nullen auf
 - streiche führende Stellen
2. Zwischen Integer- und Gleitkommatypen
von `int` nach `float` immer erlaubt.
Bsp: `float f=5;`
Von `float` nach `int`: Warnung vom Compiler
Bsp: `int i=5.5;`

konversion_impl.cc

generell: Konversion nach „oben“ immer erlaubt!`char → short → int → long int → float → double → long double`**Achtung** `float f= 3/4; //→ f = 0.0`

1. Int. Division : „3/4=0“
2. Konversion : → f=0.0

Lösung `float f=3/4.;`**6.3.2 Explizite Konversion (cast)****Ziel** Erzwingen der Konversion**Syntax** `<Typ>(<Ausdruck>)`**Bsp:**

```
int i;
float a;
i = int( ( i + 2 ) * a );//→ keine Warnung
```

7 Felder und Strukturen

Felder und Strukturen sind abgeleitete Datentypen. Im Allgemeinen: abgeleitete Datentypen setzen sich aus elementaren zusammen.

7.1 Felder

Idee: Verarbeitung von Daten mit gemeinsamen Eigenschaften: Fasse in einem Feld(Array) zusammen.

7.1.1 Eindimensionale Felder

1. **Syntax:** Typ array_name[Größe]

2. **Initialisierung:**

array_ini.cc

- (a) in Kombination mit Deklaration
- (b) an beliebiger Stelle im Programm

3. **Bemerkung:**

- (a) Das erste Element hat Index 0; das letzte „Größe-1“
- (b) Achtung: keine Fehlermeldung falls Index zu groß.
- (c) Zuweisungsoperator „=“ nicht definiert für Felder

Bsp:

```
int a[10];
int b[10];
:
a=b; //→ nicht erlaubt.
a[5] = b[7]; //→ erlaubt.
```

- (d) Vergleichsoperator „==“ erlaubt; vergleicht nicht den Inhalt der Felder.

7.1.2 Mehrdimensionale Felder

siehe oben (?)

7.2 Strukturen

Idee zusammengehörende Information unterschiedlichen Datentyp → Strukturen (Verbund)

1. **Syntax:**

struct_geburt.cc

```
struct Name
{
    Komponentenliste
} [Strukturvariable];
```

Zugriff auf einzelne Elemente mit Punktoperator „.“

Syntax:

```
<Name der Struktur>.<Komponente>;
```

2. **Bemerkungen:**

- (a) In C: `struct geburt test;`
In C++: `(struct) geburt test;` Das heißt, dass `struct` auch wegfallen kann.
- (b) Strukturvariablen können auch Felder sein
- (c) Verschachtelung von Strukturen möglich

struct_vektor.cc

7.3 Unions

1. Strukturen

Datentyp mit mehreren Komponenten, je hat dabei separaten Speicherplatz.

2. Union

Datentyp mit mehreren Komponenten; definiert einen Speicherplatz, den man mehrere Namen geben kann. `union_wert.cc`

3. Syntax

```
union Name
{
    Komponentenliste
} [Unionvariable];
```

7.4 Aufzählungstyp

- Datentyp mit limitierter Anzahl von Werten
- Zugriff mit Namen

`enum_1.cc`

1. Syntax

```
enum Name
{
    Komponentenliste
} [enum-Variable];
```

2. Bsp:

```
enum Wochentag {Sonntag, Montag, ... , Samstag};
...
Wochentag heute = Dienstag;
```

3. Bemerkung:

- Compiler ordnet jeder Komponenten eine Integerzahl zu
Sonntag \rightarrow 0, ..., Samstag \rightarrow 6
Kann bei Definition selbst angegeben werden: Bsp: `enum Wochentag {Sonntag = 7, Montag = 6 }`
- Compiler unterscheidet streng zwischen verschiedenen Aufzähltypen
- Konversion: `enum` \rightarrow `int` ok!
`int` \rightarrow `enum` Fehlermeldung!

7.5 Namen für Typen: typedef

- Definition von eigenen Typen
- Erweiterung der Grundlagen von C++

1. Syntax: `typedef <Typ> <Typ-Name>;`

2. Bsp.:

```
(a) typedef string Name;
     $\rightarrow$  „Name“ Synonym für „string“
    Name author;
    string author; äquivalent!
```

`typedef_1.cc`

- (b) `typedef int gruppe[10];` `typedef_2.cc`
→ „gruppe“ ist neuer Typ, gruppe ist ein Array der Dimension 10; jeder Eintrag ist ein `int`.

8 Funktionen

Motivation ;

- Funktionen werden dann eingesetzt, wenn bestimmte Teilaufgaben mehrfach erledigt werden müssen.
- effiziente Programmerstellung / übersichtlichere Programme
- modulare Programmerstellung möglich
- C++ ist objektorientiert
 - frei Funktionen (=Funktionen)
 - Methoden (Mitgliederfunktion)

Bsp:

1. Funktion:

```
#include <cmath>
:
float x,y;
:
x = pow( y , 2 );// x = y2
```

2. Methode:

```
#include <string>
:
int i; string s; char c;
:
c = s.at(i); //c ist i.Zeichen des Strings
```

8.1 Funktionen

8.1.1 Deklaration einer Funktion (Funktionsprototyp)

- Spezifiziere Datentyp des Funktionswertes
- Spezifiziere Datentyp der Parameter, die übergeben werden
- Info an den Compiler über wesentliche Eigenschaften der Funktion; keine vollständige Beschreibung

Syntax <Rückgabety> <Fktname> (<Parameterliste mit Typen>);

Bsp:

```
float f(int x, float y);
<Rückgabety>: int, float, ... ,
void: Funktion liefert kein Ergebnis zurück → Prozedur.
<Parameterlist>:
```

- () keine Parameter werden übergeben.
- (void) -“-
- (int, float) äquivalent
- (int x, float y) -“-

8.1.2 Definition der Funktion

Syntax

```
<Rückgabety> <Fktname> (<Parameterliste mit Typen>)
{
    Anweisungen
}
```

Bsp:

```
float f(int x, float y)
{
    return(2*x+y);
}
```

entspricht $f(x, y) = 2x + y$

Bemerkung:

1. In der Parameterliste müssen Typ und Name vorhanden sein
2. übergebene Parameter = lokale Variablen in Funktion
3. Im Anweisungsblock weitere Variablen definieren, Blöcke, ...
4. Deklaration und Definition können in einem Schritt erfolgen.
5. Falls separate Deklaration → Definition kann an beliebiger Stelle im Programm stehen
6. „return(...)“ - Anweisung führt zum Verlassen der Funktion.
Rückgabety „void“ → kein „return(...)“.

8.1.3 Aufruf der Funktion

Syntax <Fktname> (<Aktualparameterliste>);

Bsp.:

```
zahl3 = f( zahl1, zahl2 );
```

funktion_1.cc

(<Aktualparameterliste>) : Anzahl, Typ und Reihenfolge müssen mit Parameterliste bei der Definition übereinstimmen.

Bem.:

1. Die „fertigen“ Funktionen sind in den Header-Dateien deklariert und definiert.
z.B.: `cmath` → `pow`
2. Beim Programmstart wird Funktion `main()` aufgerufen.

8.1.4 Ergänzungen

1. Inline-Funktionen

„normale Funktionen“ :

- Code getrennt vom Hauptprogramm
- Aufruf → Sprung in die Funktion
- Parameterübergabe
- Rücksprung
- Ergebnisrückgabe

⇒ kostet Zeit „overhead“

„inline“ : Code wird direkt ins Hauptprogramm kopiert
sinnvoll für kurze Funktionen

funktion_1_inl.cc

Syntax inline <Rückgabety> ...

2. Funktionen mit default-Werten

funktion_def.cc

Default-Werte für Parameter können bei der Funktionsdefinition angegeben werden. Diese werden beim Aufruf eingesetzt, wenn kein aktuelles Argument angegeben ist.

3. const-Parameter

funktion_1_const.cc

Schutz des Arguments vor Veränderung innerhalb der Funktion

Syntax <Rückgabety> <Fkt-Name>(const <Var-Name> ...)

8.2 Methoden

1. (*freie*) Funktionen operieren global auf der Ebene des Programms. „frei“: nicht an bestimmtes Objekt gebunden
2. Methoden operieren im Bereich eines bestimmten Objekts „objektorientiert“. *Siehe dazu „Kapitel 9 Klassen“*

8.3 Rekursion und Iteration

8.3.1 Rekursion

Eine Funktion ruft sich selbst direkt oder indirekt aus.

2 Bedingungen für Rekursion:

1. Es muss einen Endpunkt geben
2. Die zuleistende Arbeit muss bei jedem Aufruf reduziert werden

8.3.2 Iteration

Wiederholte Ausführung einer bestimmten Sequenz von Befehlen; Abbruchbedingung notwendig.

Bsp: Fakultät

fakultaet.cc

$$f(0) = 1; \quad f(n) = f(n-1) \cdot n; \quad n \in \mathbb{N}$$

$$f(n) = n!$$

Bemerkungen

1. Iteration: Feld (fester Länge) wird benötigt
2. „Rekursion = Schleife + unbeschränkt großes Feld“
3. Auch Rekursion benötigt Platz: es werden viele Funktionsaufrufe erzeugt, allerdings nach Bedarf

Weitere Beispiele

- **Fibonacci**

fibonacci.cc

- **Größter Gemeinsamer Teiler**

ggt_rek.cc

$$ggT(a, b) = \begin{cases} a & a = b \\ ggT(b - a, a) & b > a \\ ggT(a - b, b) & a > b \end{cases}$$

8.4 Wert- und Referenzparameter

8.4.1 bisher: Werteparameter

- Werte werden in Funktionen / Methoden transformiert
aktuelle Parameterwerte werden berechnet = Initialwert für formale Parameter
- kein Rücktransport des Werts

Bsp:

```
void swap( int x, int y )
{
    int t = x;
    x = y;
    y = t;
    return;
}
int main()
{
    int a = 3;
    int b = 5;
    swap( a , b );// Aufruf ohne Wirkung
}
```

8.4.2 Referenzparameter

im obigem Beispiel:

```
void swap( int x, int y ) → void swap( int &x, int &y )
x, y sind nun Referenzen; & der Referenz- oder Adressoperator; Bsp:
int zahl;
int &ref = zahl;
zahl = 5; ref = 5; // identisch
zahl = 7; //ändert auch ref
ref = 8; //änder auch zahl
```

fkt_tausche_1.cc

Bemerkungen:

1. aktuellen Parameter selbst werden in die Funktion transformiert
2. Alle Aktionen auf formalen Parameter = Aktionen auf aktuellen Parameter
im Beispiel: alle Rechenoperationen mit x und y finden mit a,b statt
3. Referenzen müssen bei der Deklaration initialisiert werden. Funktionsaufrufe erzeugt, allerdings nach Bedarf

Weitere Beispiele

- **Größter Gemeinsamer Teiler**
Achtung: konstanter Wert bei Referenz nicht erlaubt `ggT(a, b, 6)`
Abhilfe: `void g(const int &x)` in diesem Beispiel sinnlos.

ggT_ref.cc

8.4.3 Zusammenfassung

- Transport von Werte in eine Funktion:
 - als Wertparameter

- als Referenzparameter
- als Wert einer globalen Variablen
- als Wert einer Komponente von zugehörigem Verbundes (nur Methoden)
- Transport von Werten aus einer Funktion:
 - Rückgabewert
 - als Wert eines Referenzparameters
 - als Wert einer globalen Variablen
 - als Wert einer Verbundkomponente

Bemerkung

- Zeiger (Pointer) können auch zum Transport verwendet werden
- Rückgabetyper einer Funktion kann auch Referenz sein

8.4.4 Felder als Funktionsparameter

- Felder können nicht per Wertübergabe in eine Funktion transportiert werden feld_ref_1.cc
- Felder werden als Referenz übergeben → sogar ohne explizites „&“ feld_ref_3.cc

Achtung:

- `int (&x)[10]` // Klammern nötig, weil `[]` stärker binden als `&`.
(`int &x[10] ≐ int &(x[10])`) // Felder von Referenzen
Überlicherweise wird ein Feld ohne `&` übergeben.
- C++ transformiert Array-Parameter automatisch in „Referenzparameter“ (genauer Zeiger auf Feldanfang)
Änderung von Feldelementen daher stets auch in aufrufendem Programmteil.
- Länge von Felder muss separat übergeben werden

8.5 Überladung von Funktionen und Operatoren**8.5.1 Funktionen**

C++: für einen Funktionsnamen kann es mehrere Funktion-Definitionen geben fkt_ueber_1.cc
 $\hat{=}$ Funktionsname überladen (`overloaded`)

Vorteil: Name bei Funktionsaufruf gleich, aber Anzahl, Typen der Argumente verschieden fkt_ueber_2.cc

8.5.2 Operatoren

Idee Benutze „üblichen“ Operatoren (`+`, `-`, ...) für benutzerdefinierte Datentypen

Syntax `<Returndatentyp> operator ⊗ (Argumentenliste)`
`{ Funktionscode }` ← wie Funktion mit Namen `operator ⊗` op_ueber_1.cc

8.6 Kommandozeilenparameter

- Unter C/C++ ist jedes Programm eine Funktion \longrightarrow `int main()`
- `main()` können beim Aufruf des Programms Parameter übergeben werden \rightarrow Kommandozeilenoperator

Syntax `int main(int argc, char *argv[])`

argc: Anzahl der Parameter (inkl. Programmnamen)

***argv[]:** Array von Zeigern auf ASCII-Strings

`argv[i]` enthält in Stringform den i-ten Kommandozeilenparameter
(`argv[0]` Programmname)

`komm_zeil_par.cc`

9 Klassen und Objekte

9.1 Motivation

1. Ziel: mehr Sicherheit und Flexibilität durch Zugriffskontrolle auf Elemente.
Wichtig bei großen Programmen; modulare Programmieren
2. Idee: Abkapselung von Daten/Funktionen/Anweisungen die zusammengehören → Zusammenfassung in Klasse
Manipulation nur durch spezielle Anweisungen
Zugriff nur über definierte Schnittstellen.
3. Bsp.: Fenstersystem („KDE“, „Windows“)
Bediener verschiebt Fenster → Inhalt muss neu gezeichnet werden.
Fenster $\hat{=}$ Variable der „Fensterklasse“. Es kennt selbst die Funktion, die aufgerufen werden muss.
4. Header-Dateien:
üblicherweise wird jede Klasse in separater Datei abgelegt
klasse.h: Klassendeklaration
klasse.cc: Definitionen
Einbinden in Hauptprogramm mit
`#include "klasse.h"`

9.2 Klassen in C++

1. Programmtechnisch ist eine Klasse eine Struktur mit
 - (a) privaten Daten: nur innerhalb der Klasse verfügbar
 - (b) private Methoden (Implementierungsfunktionen) nur innerhalb der Klasse verfügbar
 - (c) öffentliche Daten: für die Kommunikation mit anderen Objekten
 - (d) öffentliche Methoden (Interface-Funktionen), über die von außen auf das Objekt zugegriffen werden kann.

2. Definition einer Klasse

class_vek.cc

```
class <Name>
{
    private:
        <private Daten und Methoden>
    public:
        <öffentlich Daten und Methoden>
};
```

- (a) Erzeugung einer Klassenvariablen analog zu Strukturen
- (b) `struct` $\hat{=}$ `class`, wobei die Komponenten öffentlich sind.
- (c) `class`: „`private`“ ist default
- (d) `private` und `public` können beliebig oft und in beliebiger Reihenfolge verwendet werden.

3. Methoden

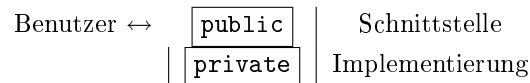
- (a) Deklaration, Definition und Aufruf: wie bei freien Funktionen
- (b) „`::`“: Bereichsauflösungsoperator (*scope operator*)

- (c) Deklaration → innerhalb Klasse
Definition → kann außerhalb sein
- (d) kein Argument: direkter Zugriff auf Datenkomponenten

4. Prinzip der Kapselung (*Geheimnisprinzip*)

- öffentliche Schnittstellen: von außen sichtbar und benutzbar
- private Implementierung: rein interne Bestandteile; für Benutzer der Klasse nicht interessant.

class_menge.cc



9.3 Freunde (friends)

Freunde einer Klasse sind Programmstücke (Klassen oder freie Funktionen), für die die `private`-Beschränkung aufgehoben ist.

9.3.1 Funktionen als Freunde

Idee erkläre freie Funktion als Freund; diese Funktion hat Zugriff auf private Komponenten.

Syntax `friend <Rückgabety> <Funktionsname>(<Parameterliste>);`

Bemerkungen

1. befreundete Funktionen = Alternative zu Methoden
2. auch Operatoren können Freunde einer Klasse sein.
3. mit „friend“ sparsam umgehen, da „private“ außer Kraft gesetzt wird.
4. Klassen als Freund

class_friend.cc

Syntax: `friend class <Name der Klasse>;`

→ alle Methoden des Freundes haben Zugriff auf private Komponenten.

5. *Freundschaften beruhen nicht auf Gegenseitigkeit.*

Bsp.: Überladen von „<<“ und „>>“

- Bisher: class_ueber.cc
Ein- und Ausgabe für Klassenvariablen als Methode definiert
- Jetzt:
Definiere „<<“ und „>>“ als befreundete Operatoren und benutze (z.B.) `cout << vek1;`
⇒ selbst definierte Objekte können ein- und ausgegeben werden wie `int`, `float`, ...
- Beachte:
 1. „<<“ und „>>“ haben 2 Argumente
 - (a) Referenz auf ein `ostream`- bzw. `istream`-Objekt; eine solche Referenz muß auch zurückgegeben werden.
 - (b) auszugebendes Objekt

2. ostream/istream sind vordefinierte Klassen
cout/cin sind vordefinierte Variablen dieser Klassen

Bemerkungen

$$\left. \begin{array}{l} \text{int x;} \\ \text{cout} \ll \text{x;} \end{array} \right\} \hat{=} \left\{ \begin{array}{l} \text{int x;} \\ \text{operator} \ll (\text{cout}, \text{x}); \\ \text{cout} \ll \text{x} \ll \text{y} \end{array} \right.$$
9.4 Konstruktoren und Destruktoren**9.4.1 Initialisierung mit Konstruktoren**1. **separate Methode**

class_menge_2.cc

2. **Konstruktor:**

Initialisieren ist Routineaktion → kann der Compiler übernehmen

Konstruktor: definiere Methode mit gleichem Namen wie Klasse und ohne Rückgabotyp.

class_menge_3.cc

Menge m1, m2;

⇒ Compiler fügt automatisch folgende Zeilen ein:

m1.Menge();

m2.Menge();

Bemerkungen:

- (a) Konstruktoren sind Initialisierungsroutinen für Objekte.
- (b) Sie werden mit der Klasse definiert und bei Variablendefinition automatisch aufgerufen.
- (c) Definition von Konstruktoren erfolgt nicht automatisch

3. **Konstruktoren mit Argumenten**

class_menge_4.cc

Konstruktoren können überladen werden; können Argumente haben.

Bemerkungen:

- (a) Defaultkonstruktor = Konstruktor ohne Argumente; wird vom Compiler automatisch aufgerufen. Beachte:

Menge m1; // ✓

Menge m1(); // Falsch

- (b) Defaultkonstruktor muss nicht vorhanden sein.

Aber:

```
class Menge
{
    public:
        Menge(int);
        //kein Defaultkonstruktor
        ...
};
```

...

Menge m1(1); //OK

Menge m2;

Ist kein Konstruktor definiert → e wird kein Defaultkonstruktor erwartet

Gibt es irgendeinen Konstruktor → Defaultkonstruktor muss bei Bedarf vorhanden sein

- (c) Konversionsoperator

Menge m4 = 42; „implizierte Konversion“

⇒ Menge::Menge(int) wird benutzt.

Menge m5 = Menge(42); „explizite Konversion“

9.4.2 Destruktoren

- sind komplementär zu Konstruktoren
- werden aufgerufen bevor Objekt vernichtet wird.

Beispiel

```
class Menge
{
    Menge(); //Konstruktor
    ~Menge(); //Destruktor
};
void f( int x )
{
    Menge m;
    //Konstruktor wird aufgerufen m.Menge();
    ...
}
```

↘ Ende der Funktion, Ende der Existenz von lokalen Variablen; Compiler ruft automatisch Destruktor auf. (m.~Menge();)

Bemerkungen

1. Destruktor erledigt Aufräumarbeiten (gibt z.B. Ressourcen frei)
2. Destruktor werden automatisch aufgerufen, aber nicht automatisch erzeugt.

9.5 Vererbung

9.5.1 Prinzip

1. **Idee:** 2 Klassen, A, B
B soll alles können, was A kann und etwas mehr.

ver_1.cc

2. **Prinzip:**

ver_form.cc

```
class A //Basisklasse
{
    public:
    void f( int );
    int i;
}

class B : public A // abgeleitete Klasse
{
    float x;
}
...
A a;
B b;
...
a = b; // möglich; b wird reduziert auf Basistyp
a.f( 1 );
b.f( 2 );
b = a; // nicht möglich
```

9.5.2 Sichtbarkeit bei Vererbung

1. öffentliche Komponenten der Basisklasse sind öffentlich in jeder Ableitung
2. private Komponenten der Basisklasse sind in den Ableitungen nicht zugreifbar.
3. `protected`: Sichtbarkeit der Komponenten nur für Basisklasse und ihrem Ableitungen

ver_ppp.cc

9.5.3 Vererbungstypen**1. öffentliche Vererbung**

Bsp:

```
class Ab : public Basis { ... }
```

- `public` bleibt `public`
- kein Zugriff von `Ab` auf `private`-Komponenten von `Basis`

2. private Vererbung:

alle (`private` und `public`) Komponenten der Basisklasse sind in den Ableitungen `private`.

```
Bsp: class Ab : private Basis{ ... }
```

3. protected-Vererbung:

alle öffentlichen Komponenten der Basis-Klasse. Werden zu `protected`-Komponenten in der abgeleiteten Klasse.

```
Bsp: class Ab : protected Basis { ... }
```

4. Mehrfachvererbung:

```
class Messer{};
class Schere{};
class Feile{};

class SchweizerMesser : public Messer,
                        public Schere,
                        public Feile {};
```

9.6 Schablonen**9.6.1 Funktionsschablonen**

1. **Idee:** oft sind Programmstücke sehr ähnlich:

max_fi.cc

- (a)
 - i. Schreibe `float`-Version
 - ii. Kopiere „`float`“ → „`int`“
- (b) Benutze Schablonen; d.h. Schritt 1(a)ii wird vom Compiler gemacht.

2. Struktur einer Schablone

`template <class T>` → Es folgte eine Schablone Parameter `T` \equiv Typ
Rest Funktionsmuster; verwende `T`

temp_max.cc

Bemerkungen

- (a) `Template` = Kopier / Editieranweisung an Compiler:
 - kopiere `Template`
 - ersetze `T` durch `int` oder `float` oder - übersetze Funktionsdefinition
- (b) Am Typ der Parameter erkennt der Compiler welche Version gebraucht wird

3. **Explizite Angabe von Schablonenparameter**; falls temp_max_expl.cc
- (a) man sich nicht auf den Compiler verlassen will
 - (b) Angabe nicht eindeutig
4. **Parameter einer Schablone** temp_sort.cc
- (a) Typ-Parameter; „class“ temp_sort_2.cc
 - (b) Nicht-Typ-Parameter $\hat{=}$ formale Parameter einer Funktion.

9.6.2 Klassenschablonen

- Syntax ähnlich wie bei Funktionsschablonen temp_stack.cc
- Unterschied: Spezialisierungen müssen immer explizit angegeben werden.

9.7 STANDARD TEMPLATE LIBRARY (STL)

C++:

1. Sprachdefinition
2. STL
 - Sammlung von Funktionen und Typendefinitionen, Strings, ...
 - spart Programmierarbeit

STL Funktionen und Klassenschablonen

3 wesentliche Bestandteile

1. **Behälter (Container)** stl_list.cc
Klassentemplates für Listen, Vektoren, Maxtrizen
2. **Iteratoren**
„Durchwandern“ durch Behälter
3. **Algorithmen** stl_complex.cc
Manipulationen angewandt auf Behälter (Sortieren, Permutationen, ...)

10 Pointer (Zeiger)

10.1 Motivation und Notation

Zeiger \equiv Pointer = (Anfangs)Adresse von Größen

1. Warum Pointer?

- ◇ dynamische Speicherverwaltung
- ◇ Manipulation von Fehler u. Strings
- ◇ Standardbibliothek in C und C++ verwenden Pointer
- ◇ komplexe Datenstrukturen (verkettete Liste)

2. Deklaration von Pointer

Syntax: <Typ> * <Name>

Beispiel: `int *ip; //ip ist ein Pointer auf Integer-Variable`

Bemerkung

1. <Name> ist Pointer auf Variable von Typ <Typ>

2. Ein Pointer muss deklariert werden wie jede andere Variable

pointer_1.cc

- | | | |
|----|---------------------------|------------------------|
| * | Verweisoperator | gibt den Inhalt |
| | Inhaltsoperator | der Speicherstelle aus |
| 3. | Dereferenzierungsoperator | |
| & | Adressoperator | gibt Adresse an |

referenziertes Objekt: Größe auf die Pointer zeigt.

pointer_2.cc

4. Zeiger : „Adressvariable“

5. Falsch:

```
int *a;
*a = 5; } a(=Adresse) hat im Allgemeinen keinen vernünftigen Wert
```

Richtig:

```
int b=5;
int *pb = &b;
```

6. Nullpointer

keine gültige Speicheradresse; nützlich als „neutrales Element“.

Beispiel:

```
float *ptr;
ptr = NULL; //NULL ist vordefinierte Größe i.A. NULL = 0
//nicht!: *ptr = NULL
```

7. Compiler ist streng bei Typenprüfung

```
int *p;
float f;
ip = &f; //Falsch!
```

8. void-Pointer

offen lassen auf welchen Datentyp ein Pointer zeigt.

Bsp:

```
void *ptr;
ptr = &f; //erlaubt
ip=(int*)ptr; //korrekt
```

Beispiele

```
int a;
int *a_zeig;
a_zeig = &a;    //--> a_zeig zeigt auf a; a und *a_zeig sind identisch
*a_zeig = 5;    //--> a=5;
```

10.2 Zeiger, Arrays, Strukturen und Funktionen**10.2.1 Zeiger und Arrays**

1. betrachte `<Typ> a[N];`
dann ist automatisch Zeigerkonstante `a` definiert mit `a=&a[0]`
Es gilt `a + n = &a[n]` „Pointerarithmetik“
Gilt auch für Inhalte `*(a + n) = a[n]`

pointer_an_na.cc

- (a) Intern erfolgt Umwandlung von `n` in einem Zeiger von jeweiligen Typ. Tatsächlich wird folgendes berechnet:
`a+n*sizeof(*a)`
- (b) Es gilt:
`a[n] = *(a + n) = *(n + a) = n[a]`

2. betrachte `<Typ> *b;`
`*(b + n) = b[n]`

pointer_3.cc

Beispiel

pointer_string.cc

1. 2x3 Matrix:
`float a[2][3];`

```
a[i][j] = ( a[i][j] )           //i = 0,1
         = (( a[i] ) + j )     //j = 0,1,2
         = (( *( a + i ) + j )
```

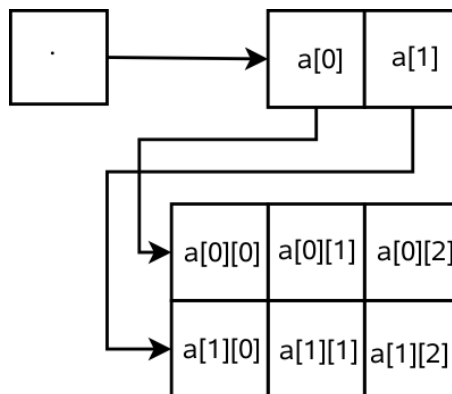


Abbildung 6: Arrays realisiert mit Pointern

`a[i] = *(a + i)`: Pointer auf Zeilenanfänge
`*(a + i) + j`: Pointer auf das `j`-te Element der `i`-ten Zeile

10.2.2 Übergabe von Arrays an Funktionen

Beispiel

```
void fkt( float v[] );
wird automatisch umgewandelt in:
void fkt( float *v );
Hauptprogramm: float v[10]; fkt(v);
```

pointer_4.cc

10.2.3 Zeiger auf Funktionen

Auch Funktionen haben Anfangsadressen; Name (ohne Klammern) hat Adresse als Wert.

1. Deklaration von Funktionspointern

```
int fct(float, int);
int main()
{
    ...
    int (*pfct)(float , int); //Rückgabewert: int; Parameter float, int;
    //pfct ist Pointer; (*pfct) ist eine Funktion
    pfct = fct;
    pfct( 3.5 , 4 );
}
```

2. Übergabe von Funktionen als Parametern

Beispiel:

```
float integrate( float(*f)( float x )
{...}
int main()
{
    cout << integrate( sin );
}
```

pointer_array_fkt.cc

3. Funktionen, die Pointer zurückgeben

Beispiele:

- (a) `int *f();`
Funktion ohne Parameter; Rückgabety: Pointer auf `int`
- (b) `int *f(*f)();`
`f` ist Pointervariable auf eine Funktion ohne Parameter mit Rückgabety: Pointer auf `int`

10.2.4 Zeiger auf Strukturen

Syntax für Zugriff auf Komponenten einer Struktur:
<Zeiger auf Struktur> -> <Elementname>

pointer_struct.cc

10.3 Dynamische Speicherverwaltung

Idee Belegung von Speicherplatz variabler Größe zur Laufzeit.

1. Operator `new` Speicherplatzbelegung

Syntax `new <Typ>;`
`new` erzeugt einen Pointer auf `<Typ>`.

Beispiel:

```
p = new int;
```

⇒ p ist Pointer auf einem int-Speicherplatz. „new int“ hat 2 Funktionen:

- (a) reserviere Platz für einen int-Wert
- (b) liefere Pointer zurück

Bemerkungen:

Falls Speicherplatzbeschaffung fehlgeschlagen

⇒ int gibt Zeiger NULL zurück; (kann abgefragt werden)

2. Operator `delete <Variablenname>;`
gibt den Speicherplatz, auf der `<Variablenname>` zeigt wieder frei.
Bei Programmen mit wenig dynamischen Speicherplatzbedarf ist `delete` nicht notwendig.

3. Beispiele

- (a) verkettete Liste

Ziel: Eingabe von Zahlen

Zahlen werden in einem `struct` abgelegt.

```
s struct
```

pointer_liste.cc

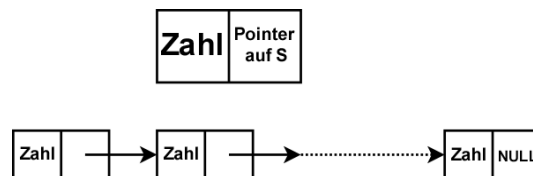


Abbildung 7: verkettete Liste mit Pointern

Auch Pointerverwendung in beiden Richtungen möglich.

Teil II

Mathematische Einschübe

1 Lineare Gleichungssysteme (LGS)

1.1 Motivation

geg. Matrix $A \in R^{n \times n}$
$$\begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix}$$

Vektor $b \in R^n$
$$\begin{pmatrix} b_{11} \\ \vdots \\ b_{n1} \end{pmatrix}$$

Schreibe $(A, b) = \left(\begin{array}{ccc|c} a_{11} & \dots & a_{1n} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{n1} & \dots & a_{nn} & b_n \end{array} \right)$

Bemerkungen: Ohne das Ergebnis zu verändern, kann man ...

1. Vielfache einer Zeile zu einer anderen addieren.
2. Zeilen in (A,b) vertauschen

1.2 Schritte

1. Bringe A in Dreiecksform:
$$\left(\begin{array}{cccc|c} x & * & \dots & \dots & \alpha_1 \\ 0 & x & * & \vdots & \vdots \\ \vdots & 0 & x & * & \vdots \\ 0 & \dots & 0 & x & \alpha_n \end{array} \right)$$

2. Auflösen der Gleichungen durch Rückwärtseinsetzen.

Zu (1): Betrachte (A,b); n Gleichungen

Algorithmus:

1. Bestimme $a_{r1} \neq 0$ fahre mit (2) fort, falls kein r existiert: A ist singulär; stopp;
2. Vertausche Zeile 1 mit Zeile r: $(A, b) \rightarrow (\bar{A}, \bar{b})$
3. Subtrahiere geeignetes Vielfaches $l_{i1} := \frac{\bar{a}_{i1}}{\bar{a}_{11}}$ von dem übrigen Gleichungen, sodass $\bar{a}_{i1} = 0$ mit $i = 2, \dots, n$: $(\bar{A}, \bar{b}) \rightarrow (A', b')$
4. Verfahre analog für Spalte 2, ..., n

Bemerkungen

1. LGS $Ax = b$ und LGS $A'x = b'$ haben gleiche Lösung
2. Element a_{r1} in (1) heißt *Pivot-Element (Leitelement)*;
Schritt (1): Pivotsuche
gutartiges (numerisch) Verhalten für $\|a_{r1}\| = \max_{i=1}^n \|a_{i1}\|$
Spaltenpivotsuche
3. Totalpivotsuche: Vertauschen von Zeilen **und** Spalten um Maximum zu finden.

zu (2): Ergebnis von (1)

$$\left(\begin{array}{cccc|c} x & * & \dots & \dots & \alpha_1 \\ 0 & x & * & \vdots & \vdots \\ \vdots & 0 & x & * & \vdots \\ 0 & \dots & 0 & x & \alpha_n \end{array} \right)$$

$$x_n = \frac{c_n}{r_{nn}}$$

$$x_{n-1} = \frac{1}{r_{n-1,n-1}}(c_{n-1} - r_{n-1,n}x_n)$$

\vdots

$$x_1 = \frac{1}{r_{11}}(c_1 - r_{12}x_2)$$

1.3 Verfahren von Gauss-Jordan

Im k-ten Teilschritt werden alle Elemente der k-ten Spalte zu 0 gemacht, außer a_{kl}

Bsp: Nach 3. Schritt

$$\left(\begin{array}{cccc|c} 1 & 0 & 0 & \dots & b_1 \\ 0 & 1 & 0 & \vdots & b_2 \\ 0 & 0 & 1 & * & \vdots \\ 0 & 0 & 0 & x & b_n \end{array} \right)$$

1. GJ-Algorithmus kann zur Bestimmung der inversen Matrix verwendet werden
2. ...

2 Interpolation

2.1 Motivation

1. Datenpunkte aus Experiment
 $(t, x(t))$
(Schaubild aus verschiedenen Punkten)
 Frage wie sieht $x(t)$ zwischen den Punkte des Schaubilds aus?
 Interpolation = Verfahren, um $x(t)$ für alle t zu bestimmen.
2. geg: $f[a, b] \rightarrow R$
 Approximiere f durch einfachere Funktionen.
 Methoden:
 - Polynom-Interpolation: $f(t) = \sum_k a_k \cdot t^k$
 - Trigonometrische-Interpolation: $f(t) = \sum_k a_k \cdot e^{i k t}$
 - Spline-Interpolation: $f(t) = \sum_k a_k f_k(t)$
 kubische Splines :
 - f_k : Polynome 3. Ordnung
 - $f(t)$ 2 mal stetig differenzierbar
 - Interpolation mit rationalen Funktionen $f(t) = \frac{\sum_k a_k \cdot t^k}{\sum_k b_k \cdot t^k}$

2.2 Polynom-Interpolation

1. **LAGRANGE-Darstellung**
 Satz: $f[a, b] \rightarrow R$; $n+1$ Stützstellen
 $x_0, \dots, x_n \in [a, b]$
 Funktionswerte $f(x_0), \dots, f(x_n)$
 Dann existiert ein eindeutig bestimmtes Polynom vom Grad n mit $P_n(x_k) = f_k$

$$P_n(x) = \sum_k f_k \cdot l_k(x); l_k(x) = \prod_{i=0; i \neq k}^n \frac{x - x_i}{x_k - x_i}$$

l_k : LAGRANGE'sche Grundpolynome

$$l_k(x_i) = \delta_{ik}$$

Bemerkungen

- (a) Konstruktion von $P_n(x)$ klar
 - (b) Nachteil: Hinzunahme von neuem Wertepaar $(x_{n+1}, f(x_{n+1})) \rightarrow$ starte Berechnung von vorne.
2. **Der Algorithmus von NEVILLE**
 Ziel: Berechne Wert des Interpolationspolynom an fester Stelle x .
 Satz: geg. Stützpunkte $(x_i, f_i) \quad i = 0, 1, \dots$
 $P_{i_0, \dots, i_k}(x)$ sei das Polynom mit $P_{i_0, \dots, i_k}(x_{i_j}) = f_{i_j} \quad j = 0, \dots, k$
 Dann gilt die Rekursionsformel:

$$(a) P_i(x) = f_i$$

$$(b) P_{i_0, \dots, i_k} = \frac{(x-x_{i_0}) \cdot P_{i_0, \dots, i_k} - (x-x_{i_k}) \cdot P_{i_0, \dots, i_{k-1}}(x)}{x_{i_k} - x_{i_0}} \quad \text{kompakte Schreibweise } P_{i, i+1, \dots, i+k}(x) = P_{i+k, k}(x) = P_{i+k, k}$$

$P_{i+k, k}$: Interpolationspolynom vom Grad k mit Stützstellen x_i, \dots, x_{i+k}

$$i. P_{j, 0} = f_j$$

ii.

$$\begin{aligned}
 P_{j,k} &= \frac{(x - x_{j-k}) \cdot P_{j,k-1} - (x - x_j) \cdot P_{j-1,k-1}(x)(x)}{x_j - x_{j-k}} \quad 1 \leq k \leq j = 0, 1, 2, 3, \dots \\
 &= P_{j,k-1} + \frac{P_{j,k-1} - P_{j-1,k-1}}{\frac{x - x_{j-k}}{x - x_j} - 1}
 \end{aligned}$$

3. Algorithmus von NEVILLE

Benutze (a) und (b), um folgende Tabelle aus Werten der interpolierenden Polynome $P_{j,k}$ an einer festen Stelle x zu konstruieren.

$$\begin{array}{l|l}
 x_0 & f_0 = P_{00} \\
 x_1 & f_1 = P_{10} > P_{11} \\
 x_2 & f_2 = P_{20} > P_{21} > P_{22} \\
 x_3 & f_3 = P_{30} > P_{31} > P_{32} > \boxed{P_{33}}
 \end{array}$$

4. Verbesserung der Genauigkeit

Def. $Q_{i0} = D_{i0} = f_i$

$Q_{i,k} = P_{i,k} - P_{i,k-1}$

$D_{i,k} = P_{i,k} - P_{i-1,k-1}$

Man erhält folgende Rekursionsformeln

$$Q_{ik} = (D_{i,k-1} - Q_{i-1,k-1}) \frac{x_i - x}{x_{i-k} - x_i}$$

$$D_{ik} = (D_{i,k-1} - Q_{i-1,k-1}) \frac{x_{i-k} - x}{x_{i-k} - x_i}$$

$$1 \leq k \leq i; \quad i = 0, 1, \dots$$

Berechne Tabelle für $Q_{ik}, D_{ik} \rightarrow$

$$P_{nn} = f_n + \sum_{k=1}^n Q_{n,k}$$

Vorteil: Falls Werte $f_0, f_1, f_2, \dots \approx$ gleich $\rightarrow Q_{ik}$ klein gegen f_i

\rightarrow Summiere im ersten Schritt Q_{nk} auf; vermeide Rundungsfehler.

Bemerkungen:

- (a) Arthen's Methode: selbe Rekursion, selbes Ergebnis, Zwischenschritte leicht unterschiedlich
- (b) Hinzunahme von neuem Punkt (x_k, f_k) kein Problem.

3 Numerische Integration

Ziel: Näherung für $F = \int_a^b f(t) dt$ mit f ist integrierbar

gesucht: Formeln der Art

$$F = \sum_{k=0}^n c_k f(x_k) + R_n =: F_n + R_n$$

c_k : Koeffizienten oder Gewichte der Integrationsformel

x_k : Stützstellen

R_n : Restglied

3.1 NEWTON-COTES-Formeln

3.1.1 Methode zur Gewinnung der Integrationsformeln

- Idee:** Interpolation von f und Integration des Interpolationspolynoms.
NEWTON-COTES: äquidistante Stützstellen:

$$x_0 = a$$

$$x_n = b$$

Schreibe $f(x) = p_n(x) + r_n(x)$

$$\begin{aligned} \Rightarrow F = \int_a^b f(x) dx &= \int_a^b p_n(x) dx + \int_a^b r_n(x) dx \\ &= F_n + R_n \end{aligned}$$

F_n geschlossen auswertbar

R_n i.a. nur abschätzbar

Ziel: Direkte Berechnung der Gewichte ohne explizite Aufstellung von $p_n(x)$

2. Gewichte

$h = \frac{b-a}{n}$; Stützstellen: $x_0 = a$; $x_k = x_0 + k \cdot h$; $k = 0, \dots, n$

$$f_k = f(x_k);$$

p_n sei in der Form von LAGRANGE gegeben

$$\begin{aligned} f(x) &= p_n(x) + r_n(x) \\ &= \sum_{k=0}^n f_k \cdot l_k + r_n(x); \quad l_k(x) = \prod_{i=0; i \neq k}^n \frac{x - x_i}{x_k - x_i} \\ \Rightarrow F &= \int_{x_0}^{x_0+n \cdot h} p_n(x) dx + \int_{x_0}^{x_0+n \cdot h} r_n(x) dx \\ &= F_n + R_n \\ &= \sum_{k=0}^n f_k \int_{x_0}^{x_0+n \cdot h} l_k(x) dx = (b-a) \cdot \sum_{k=0}^n f_k c_k^{(n)} \end{aligned}$$

mit $c_k^{(n)} = \frac{1}{n} \int_0^n \tilde{l}_k(s) ds$ COTES-Zahlen

$$\tilde{l}_k(s) = \prod_{i=0; i \neq k}^n \frac{s-i}{k-i}$$

3. Eigenschaften der COTES-Zahlen

(a) wähle $n = 0$; $b = 1$; $f = 1$

$$1 = \int_0^1 f(x) dx = \int_0^1 p_n(x) dx = F_n = \sum_{k=0}^n c_k^{(n)}$$

(b)

$$\tilde{l}_k(s) = \tilde{l}_{n-k}(n-s)$$

$$\begin{aligned} c_k^{(n)} &= \frac{1}{n} \int_0^n \tilde{l}_k(s) ds = \frac{1}{n} \int_0^n \tilde{l}_{n-k}(n-s) ds \\ &= \frac{1}{n} \int_0^n \tilde{l}_{n-k}(t) dt = c_{n-k}^{(n)} \end{aligned}$$

$$\text{mit } t = n - s \text{ und } \int_0^n ds = \int_0^n dt$$

3.1.2 Beispiele

1. $n = 1$; $x_0 = a$; $x_1 = b$

$$\left. \begin{aligned} c_0^{(1)} + c_1^{(1)} &= 1 \\ c_0^{(1)} &= c_1^{(1)} \end{aligned} \right\} \implies c_0^{(1)} = c_1^{(1)} = \frac{1}{2}$$

$$\implies \boxed{F_1 = \frac{b-a}{2} (f(a) + f(b))}$$

„Trapezregel“

2. $n = 2$; $x_0 = a$; $x_1 = \frac{a+b}{2}$; $x_2 = b$; $h = \frac{b-a}{2}$

$$\tilde{l}_0(s) = \frac{s-1}{0-1} \cdot \frac{s-2}{0-2} = \frac{s^2 - 3s + 2}{2}$$

$$c_0^{(2)} = \frac{1}{6}$$

$$c_2^{(2)} = c_0^{(2)} = \frac{1}{6}$$

$$c_1^{(2)} = 1 - \frac{1}{3} = \frac{2}{3}$$

$$\implies F_2 = (b-a) \left(\frac{f(a)}{6} + \frac{2f(\frac{a+b}{2})}{3} + \frac{f(b)}{6} \right)$$

$$\implies \boxed{F_2 = \frac{(b-a)}{6} (f(a) + 4f(\frac{a+b}{2}) + f(b))}$$

„SIMPSON-Regel“ oder „KEPLER'sche-Fassregel“

3. $n = 3, 4, 5, 6 :$

n											$C_K^{(n)}$	
1											$\frac{1}{2}$	
2											$\frac{2}{3}$	
3											$\frac{3}{8}$	
4			$\frac{7}{90}$	$\frac{1}{8}$	$\frac{32}{90}$	$\frac{3}{8}$	$\frac{12}{90}$	$\frac{3}{8}$	$\frac{32}{90}$	$\frac{1}{6}$	$\frac{7}{90}$	$\frac{3}{8}$ -Regel
5	$\frac{19}{288}$	$\frac{75}{288}$	$\frac{50}{288}$	$\frac{50}{288}$	$\frac{272}{840}$	$\frac{50}{288}$	$\frac{27}{840}$	$\frac{75}{288}$	$\frac{7}{90}$	$\frac{19}{288}$		
6	$\frac{41}{840}$	$\frac{216}{840}$	$\frac{27}{840}$	$\frac{27}{840}$	$\frac{272}{840}$	$\frac{50}{288}$	$\frac{27}{840}$	$\frac{216}{840}$	$\frac{7}{90}$	$\frac{41}{840}$		

Bemerkung Für $n = 8$ und $n \geq 10$ treten negative Gewichte auf ← nicht erwünscht, numerisch instabil.

3.1.3 Fehlerabschätzung bei NC-Formeln

Trapezregel:

$$|R_1| \geq \frac{M_2}{12} (b - a)^3; \quad |f''(x)| \geq M_2 \quad (x \in [a, b])$$

Simpsonregel:

$$|R_2| \geq \frac{M_4}{90} \left(\frac{b - a}{2}\right)^5; \quad |f''(x)| \geq M_4 \quad (x \in [a, b])$$

3.1.4 Allgemeine Trapez- und Simpsonregel

Idee:

Man verwendet keine NC-Formeln hoher Ordnung;

Deshalb: Bei großen Integrationsintervallen: unterteile Intervall;

verwende NC-Formel kleiner Ordnung für Teilintervall.

1. Allgemeine Trapezregel

Betrachte N Teilintervalle; $h = \frac{b-a}{N}$

$$x_0 = a, \quad x_i = x_0 + i h, \quad x_N = b, \quad i = 0, \dots, N$$

$$F^{(i)} = \int_{x_i}^{x_{i+1}} f(t) dt = F_1^{(i)} + R_1^{(i)} \tag{1}$$

$$F_1^{(i)} = \frac{h}{2} (f_i + f_{i+1}) \tag{2}$$

$$F = \int_a^b f(x) dx = \underbrace{\sum_{i=0}^{N-1} F_1^{(i)}}_{F_{TR}^{(N)}} + \underbrace{\sum_{i=0}^{N-1} R_1^{(i)}}_{R_{TR}^{(N)}} \tag{3}$$

Man erhält:

$$F_{TR}^{(N)} = \frac{h}{2} (f_0 + 2 f_1 + 2 f_2 + \dots + 2 f_{N-1} + f_N) \tag{4}$$

$$|R_{TR}^{(N)}| \leq \sum_{i=0}^{N-1} |R_1^{(i)}| \tag{5}$$

$$\leq N \frac{M_2}{R} h^3, \quad |f''(x)| \leq M_2 \quad (x \in [a, b]) \tag{6}$$

2. Allgemeine Simpsonregel

$$h = \frac{b-a}{2N} \quad (7)$$

$$F_3^{(N)} = \frac{h}{3} [f_0 + 4f_1 + 2f_2 + 4f_3 + \dots + 2f_{2N-2} + 4f_{2N-1} + 2f_{2N}] \quad (8)$$

$$\left| R_S^{(N)} \right| \leq N \frac{h^5}{90} M_4 \quad (9)$$

3. Praktische Fehlerabschätzung

Kriterium: $\varepsilon > 0$ gegeben.

$$N \rightarrow 2N; \quad h \rightarrow \frac{h}{2}$$

$$\left| F_{TR}^{(N)} - F_{TR}^{(2N)} \right| < \varepsilon \left| F_{TR}^{(2N)} \right|$$

ja: fertig. **nein**: weitere Integration

4. Bemerkung zum Algorithmus

Implementierung der allgemeinen Trapezregel

(a) Verdopplung der Intervalle:

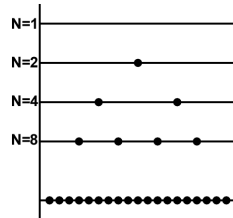


Abbildung 8: Verdopplung der Intervalle

vorteilhaft: schreibe Routine, die jeden Funktionswert nur einmal ausrechnet.

- (b) Integrationsroutine (z.B. Trapez) wird so lange aufgerufen, bis Genauigkeit bzw. maximale Intervallzahl erreicht ist.

```
for( i=1; i <= IMAX; i++ )
{
s= trapez( i );
if( fabs( s - salt ) < EPS * fabs( s ) ) return( s );
salt = s;
} → „Genauigkeit kann nicht erreicht werden“.
```

5. Romberg-Integration

Idee

- (a) Betrachte Schrittweiten:

$$h_0 = b-a; \quad h_1 = \frac{b-a}{2} = \frac{h_0}{2}; \quad h_2 = \frac{h_1}{2} = \frac{h_0}{2^2}; \quad \dots \quad h_i = \frac{h_0}{2^i}$$

- (b) Berechne zugehörige Trapezsummen

$$F_{TR}^{(2^i)} =: T_{i0} \quad i = 0, 1, 2, \dots, i_{max}$$

- (c) Berechne Interpolationspolynom
Extrapolierung zu $h = 0$

- (d) NEVILLE-Tableau

$$\begin{array}{ccccccc}
 T_{00} & & & & & & \\
 T_{10} & > & T_{11} & & & & \\
 T_{20} & > & T_{21} & > & T_{22} & & \\
 T_{30} & > & T_{31} & > & T_{32} & > & \boxed{T_{33}} \\
 \vdots & & \vdots & & & &
 \end{array}$$

$$T_{ik} = T_{i,k-1} + \frac{T_{i,k-1} - T_{i-1,k-1}}{4^k - 1}$$

Bemerkung

1. Spalte: allgemeine Trapezformel
Rest: nur Linearkombinationen

4 Gewöhnliche Differentialgleichungen

4.1 Problemstellung

1. **Ziel**

gesucht: $y(x)$ mit $y(x_0) = y_0$

gegeben: $y'(x) = f(x, y(x))$ gewöhnliche DGL 1. Ordnung.

2. **Allgemein: Systeme von n gewöhnlichen DGL**

$$y_1' = f_1(x, y_1, \dots, y_n) \quad (10)$$

$$\vdots \quad (11)$$

$$y_n' = f_n(x, y_1, \dots, y_n) \quad (12)$$

$$(13)$$

gesucht n Funktionen $y_i(x)$ $i = 1, \dots, n$ mit $\underbrace{y_i(x_0) = y_0}_{\text{Anfangsbedingungen}}$

3. **Gewöhnliche DGL m -ter Ordnung:**

$$y^m(x) = f(x, y, y', y'', \dots, y^{(m-1)})$$

kann in ein System von DGLn 1. Ordnung übergeführt werden:

$$\left. \begin{array}{l} z_1(x) = y(x) \\ z_2(x) = y'(x) \\ \vdots \\ z_m(x) = y^{(m-1)}(x) \end{array} \right\} \implies \begin{array}{l} z_1'(x) = y' = z_2 \\ z_2'(x) = z_3 \\ \vdots \\ z_m'(x) = f(x, z_1, z_2, \dots, z_m) \end{array}$$

Im allgemeinen ist geschlossene analytische Lösung nicht möglich.

Ziel: numerische Lösung

4.2 Polynomzug-Verfahren (EULER)

Näherung

$$y'(x) \approx \frac{y(x+h) - y(x)}{h}$$

$$\implies y(x+h) \approx y(x) + h \cdot f(x, y)$$

Startwerte $x_0, 0$; $x_i = x_0 + h \cdot i$ $i = 1, 2, 3, \dots$

Näherungswerte η_i für $y_i = y(x_i)$:

$$\eta_0 := y_0; \quad \eta_{i+1} = \eta_i + h \cdot f(x_i, \eta_i)$$

EULER-VERFAHREN

Bemerkungen:

1. η_i hängen von h ab. $\eta_i = \eta(x_i, h)$

2. Schreibe allgemein:

$$f(x, y) \rightarrow \Phi(x, y; h, f)$$

$$\eta_{i+1} = \eta_i + h \cdot \Phi(x_i, \eta_i; h, f)$$

3. EULER-Verfahren „ist von der Ordnung h^2 “;

1. Ordnung in h

Verfahren 2.Ordnung betrachte: $\tau = \frac{y(x+h)-y(x)}{h} - f(x, y)$

$$y(x+h) = y(x) + h \cdot y'(x) + \frac{h^2}{2} y''(x) + o(h^3)$$

$$\implies \tau = \frac{y'(x)}{1} + \frac{h}{2} \underbrace{y''(x)}_{= \frac{d}{dx}(y') = \frac{d}{dx}(f(x, y(x))) = f_x + f_y \cdot f} + o(h^2) - f(x, y)$$

$$\implies \tau = \frac{h}{2} (f_x + f_y \cdot f) + o(h^2)$$

Definition: $\Phi(x, y, h, f) = f(x, y) + \frac{h}{2} [f_x(x, y) + f_y(x, y) \cdot f(x, y)]$

Nach Konstruktion Verfahren 2.Ordnung.

Nachteil: Ableitungen schwierig auswertbar.

4.3 RUNGE-KUTTA-VERFAHREN

Ziel Vermeide Ableitungen (aber hohe Ordnungen)

1. Ansatz für RUNGE-KUTTA der Ordnung 2

$$\Phi(x, y, h, f) = a_1 \cdot f(x, y) + a_2 \cdot f(x + p_1 \cdot h, y + p_2 \cdot h \cdot f(x, y))$$

Bestimme a_1, a_2, p_1, p_2 so, dass $\Phi(x, y; h, f)$ mit $f(x, y) + I(x, y) = \frac{y(x+h)-y(x)}{h}$ zu möglichst hoher Ordnung übereinstimmt.

(Entwickle Φ in h und vergleiche)

2 Lösungen

(a) $\Phi(x, y; h, f) = \frac{1}{2} [f(x, y) + f(x + h, y + h \cdot f(x, y))]$ „Trapezregel“

(b) $\Phi(x, y; h, f) = f(x + \frac{h}{2}, y + \frac{h}{2} \cdot f(x, y))$ „Mittelpunkts-Formel“

2. RUNGE-KUTTA-VERFAHREN der Ordnung 4:

$$\Phi(x, y; h, f) = \frac{1}{6} (k_1 + 2 \cdot k_2 + 2 \cdot k_3 + k_4) \quad (14)$$

$$k_1 = f(x, y) \quad (15)$$

$$k_2 = f(x + \frac{h}{2}, y + \frac{h}{2} \cdot k_1) \quad (16)$$

$$k_3 = f(x + \frac{h}{2}, y + \frac{h}{2} \cdot k_2) \quad (17)$$

$$k_4 = f(x + h, y + h \cdot k_3) \quad (18)$$

Verfahren 4.Ordnung

Beispiel

$$y'' + y = 0; \quad y(0) = 0; \quad y'(0) = 1$$

(Analytische Lösung : $y(x) = \sin x$)

Überführen in DGL 1.Ordnung:

$$\left. \begin{aligned} z_1 &= y \\ z_2 &= y' \end{aligned} \right\} \begin{aligned} z_1' &= z_2 \equiv f_1(x, z_1, z_2) \\ z_2' &= -z_1 \equiv f_2(x, z_1, z_2) \end{aligned}$$

$$\begin{aligned} z_1(0) &= 0 \\ z_2(0) &= 1 \end{aligned}$$

EULER:

$$z_1(x+h) = z_1(x) + h \cdot z_2(x) \quad (19)$$

$$z_2(x+h) = z_2(x) - h \cdot z_1(x) \quad (20)$$