Einführung in die Programmierung - C/C++

Linz,WS 2003/04

Gundolf Haase

Inhaltsverzeichnis

Kapitel 1

Das erste Programm

1.1 Was ist ein Programm ??

Eigentlich kennt jeder bereits Programme, jedoch versteht man oft verschiedene Inhalte darunter.

- Parteiprogramm ↔ Ideen
- Theaterprogramm ↔ Ablaufplanung
- Musikpartitur ↔ strikte Anweisungsfolge
- Windowsprogramm ↔ interaktive Aktion mit dem Computer

Programmieren ist das Lösen von Aufgaben auf dem Computer mittels eigener Software und beinhaltet alle vier Teilaspekte in obiger Liste.

Eine typische Übungsaufgabe beinhaltet folgenden Satz:

⋮
Ändern [editieren] Sie das Quellfile [source file] entsprechend der Aufgabenstellung, übersetzen [compilieren] und testen Sie das Programm.

Was (??) soll ich machen ??

Idee	Im Kopf oder auf dem Papier. (Was soll der Computer machen?)	Programmidee
↓		
Idee für Computer aufbereiten	Entwurf. (Wie kann der Computer die Idee realisieren?)	Struktogramm
↓		
Idee in einer Programmiersprache formulieren.	Quelltext/Quellfile editieren. (Welche Ausdrücke darf ich verwenden?)	Programmcode
↓		
Quellfile für den Computer übersetzen	File compilieren [und linken]. (Übersetzung in Prozessorsprache)	ausführbares Programm
↓		
Programmcode ausführen	Programm mit verschiedenen Datensätzen testen	Programmtest

Bemerkungen:

1. Der Lernprozeß beim Programmieren erfolgt typischerweise von unten nach oben in der vorangegangenen Übersicht.
2. Software = ausführbares Programm + Programmcode + Ideen

Warnung : Das Programm auf dem Computer wird **genau das** ausführen, was im Programmcode beschrieben ist!

Typischer Anfängerkommentar: *Aber das habe ich doch ganz anders gemeint.*

Merke : Computer sind strohdumm! Erst die (korrekte und zuverlässige) Software nutzt die Möglichkeiten der Hardware.

1.2 Das "Hello World" - Programm in C

Idee: Das einfachste Programm, welches nur eine Meldung auf den Bildschirm schreibt.

Quelltext (*HelloWorld.c*): HelloWorld.c

```
/* HelloWorld.c */

#include <stdio.h>

main()
{
 printf("Hello World \n");

/* "\n" - new line */
}
```

- Kommentaranfang, -ende
- vordefinierte Funktionen/ Variablen/ Konstanten
- Beginn Hauptprogramm
- einfache Anweisung
- Blockanweisung

Quelltext eingeben und compilieren, Programm ausführen:

0. Computer einschalten, einloggen
   login:
   passwd:

1. Terminal bzw. Dateimanager öffnen und in das Arbeitsverzeichnis wechseln.
   `LINUX> cd progs`

2. Quelltext in das Quellfile eingeben, Editor nach eigener Wahl.
   `LINUX> nedit HelloWorld.c` oder
   `LINUX> xemacs HelloWorld.c`.

3. Quellfile compilieren. `LINUX> gcc HelloWorld.c`

4. Programm ausführen.
   `LINUX> a.out` oder
   `LINUX> ./a.out` oder
   `WIN98> ./a.exe`

Bemerkungen:

- `LINUX> gcc HelloWorld.c`
  erzeugt ein ausführbares Programm mit dem Standardnamen *a.out* .

- Falls das ausführbare Programm, z.B., *myprog* heißen soll:
  `LINUX> gcc -o myprog HelloWorld.c`
  `LINUX> myprog`

- Die konkrete Kommandozeile zum Compilieren ist abhängig vom verwendeten Compiler.

1.3 Das "Hello World" - Programm in C++

Idee und Struktogramm wie im Abschnitt 1.2

HelloWorld.cc

Quelltext (*HelloWorld.cc*):

```
// HelloWorld.cc

#include <iostream.h>

main()
{
 cout << "Hello World" << endl;

// 'endl' - new line
}
```

- Kommentar bis Zeilenende
- vordefinierte Klassen und Methoden
- Beginn Hauptprogramm
- einfache Anweisung
- Blockanweisung

Quelltext eingeben und compilieren, Programm ausführen:

0./1. wie in § 1.2 .

2. Quellfile editieren.
   `LINUX> nedit HelloWorld.cc`

3. Quellfile compilieren.
   `LINUX> g++ HelloWorld.cc`

4. Programm ausführen.
   `LINUX> a.out` oder
   `LINUX> ./a.out` oder
   `WIN98> ./a.exe`

Bemerkungen:

- Der C-Quelltext von *HelloWorld.c* kann auch in C++ übersetzt werden:
  `LINUX> g++ HelloWorld.c`
  Allerdings ist dann die Quelltextzeile `#include <stdio.h>` zwingend notwendig.
- C-Instruktionen sind eine Untermenge der C++ -Instruktionen.
- Der C-Kommentar `/*` `*/` kann auch in C++ verwendet werden.
- Der C++ Kommentar `//` gehört nicht zur Syntax von C und sollte daher nicht in C-Programmen verwendet werden. Ansonsten tritt ein Portabilitätsproblem auf, d.h., nicht jeder C-Compiler kann den Quelltext übersetzen.

Tip zum Programmieren:
Es gibt (fast) immer mehr als eine Möglichkeit eine Idee im Computerprogramm zu realisieren.
$\Longrightarrow$ Finden Sie Ihren eigenen Programmierstil (und verbessern Sie ihn).

1.4 Interne Details beim Programmieren

Der leicht geänderte Aufruf zum Compilieren
`LINUX> g++ -v HelloWorld.cc`
erzeugt eine längere Bildschirmausgabe, welche mehrere Phasen des Compilierens anzeigt. Im folgenden einige Tips, wie man sich diese einzelnen Phasen anschauen kann, um den Ablauf besser zu verstehen:

a) Präprozessing:
Headerfiles (**.hh* und **.h*) werden zum Quellfile hinzugefügt (+ Makrodefinitionen, bedingte Compilierung)
`LINUX> g++ -E HelloWorld.cc > HelloWorld.ii`
Der Zusatz `> HelloWorld.ii` lenkt die Bildschirmausgabe in das File *HelloWorld.ii*. Diese Datei *HelloWorld.ii* kann mit einem Editor angesehen werden und ist ein langes C++ Quelltextfile.

b) Übersetzen in Assemblercode:
Hier wird ein Quelltextfile in der (prozessorspezifischen) Programmiersprache Assembler erzeugt.
`LINUX> g++ -S HelloWorld.cc`
Das entstandene File *HelloWorld.s* kann mit dem Editor angesehen werden.

c) Objektcode erzeugen:
Nunmehr wird ein File erzeugt, welches die direkten Steuerbefehle, d.h., Zahlen, für den Prozessor beinhaltet.
`LINUX> g++ -c HelloWorld.cc`
Das File *HelloWorld.o* kann nicht mehr im normalen Texteditor angesehen werden, sondern mit
`LINUX> hex HelloWorld.o`

d) Linken:
Verbinden aller Objektfiles und notwendigen Bibliotheken zum ausführbaren Programm *a.out* .
`LINUX> g++ HelloWorld.o`

1.5 Bezeichnungen in der Vorlesung

- Kommandos in einer Befehlszeile unter LINUX:
`LINUX> g++ [-o myprog] file_name.cc`
Die eckigen Klammern `[ ]` markieren optionale Teile in Kommandos, Befehlen oder Definitionen. Jeder Filename besteht aus dem frei wählbaren Basisnamen (*file_name*) und dem Suffix (*.cc*) welcher den Filetyp kennzeichnet.

- Einige Filetypen nach dem Suffix:

Suffix	Filetyp
.c	C-Quelltextfile
.h	C-Headerfile (auch C++), Quellfile mit vordefinierten Programmbausteinen
.cc [*.cpp*]	C++ -Quelltextfile
.hh [*.hpp*]	C++ -Headerfile
.o	Objektfile
.a	Bibliotheksfile (Library)

- Ein Angabe wie `··· < typ > ···` bedeutet, daß dieser Platzhalter durch einen Ausdruck des entsprechenden Typs ersetzt werden muß.

1.6 Neuere C++-Compiler

Seit der ersten Version dieses Skriptes sind neuere Versionen der Headerfiles parallel zu den alten Headerfiles verfügbar, wie *iostream* anstelle von *iostream.h*. Teilweise liefern dann Compiler wie der `g++` lästige, mehrzeilige Warnungen beim Compilieren des Quelltextes auf Seite 4. Diese Fehlermeldung kann mittels
`LINUX> g++ -Wno-deprecated HelloWorld.cc`
unterdrückt werden.

HelloWorld_new.cc

Bei der (empfohlenen) Benutzung der neuen Headerfiles ändert sich unser kleines Programm in :

```
//      Include file "iostream" is used instead of  "iostream.h"

#include <iostream>

main()
{
//               Scope operator  ::  is needed

  std::cout << "Hello World" << std::endl;
}
```

Ich will den Scope-Operator, siehe §8 nicht jedesmal mitschreiben müssen, daher bevorzuge ich die Variante:

```
//      Include file "iostream" is used instead of  "iostream.h"

#include <iostream>
//      All methods from class  std  can be used.
using namespace std;

int main()
{
 cout << "Hello World" << endl;
}
```

Kapitel 2

Einfache Datentypen

2.1 Variable

2.1.1 Variablendefinition

Jedes sinnvolle Programm bearbeitet Daten in irgendeiner Form. Die Speicherung dieser Daten erfolgt in Variablen.

Die Variable
i) ist eine symbolische Repräsentation (Bezeichner/Name) für den Speicherplatz von Daten.
ii) wird beschrieben durch Typ und Speicherklasse.
iii) Die Inhalte der Variablen, d.h., die im Speicherplatz befindlichen Daten, ändern sich während des Programmablaufes.

Allgemeine Form der Variablenvereinbarung:

```
[< speicherklasse >] <typ> <bezeichner1> [, bezeichner2] ;
```

Typ	Speicherbedarf in Byte (g++)	Inhalt	Werte
`char`	1	Character-Zeichen	'H', 'e', '\n'
`bool`	1	Booleanvariable	`false`, `true` [nur C++]
`int`	4		-32767, -2^{31}
`short` [`int`]	2	Ganze Zahlen	-32767
`long` [`int`]	4		-32767, -2^{31}
`float`	4	Gleitkommazahlen	1.1, -1.56e-32
`double`	8		1.1, -1.56e-32, 5.68e+287
`unsigned` [`int`]	4	Natürliche Zahlen	32767, 32769, $2^{31}-1$
`long long` [`int`]	8	Ganze Zahlen	-2^{31}, $2^{63}-1$
`long double`	12	Gleitkommazahlen	5.68e+287, 5.68e+420

Bemerkungen:

- Characterdaten speichern genau ein ASCII- oder Sonderzeichen.
- Der Speicherbedarf von Typen der Integer-Gruppe (`int`) kann von Compiler und Betriebssystem (16/32 bit) abhängen. Es empfiehlt sich daher, die einschlägigen Compilerhinweise zu lesen bzw. mit dem `sizeof`-Operator mittels `sizeof( <typ> )` oder `sizeof( <variable> )` die tatsächliche Anzahl der benötigten Bytes zu ermitteln. Siehe dazu auch das folgende Beispiel :

DataTypes.cc

```
/* Demo fuer sizeof-Operator */

#include <iostream.h>

main()
{
 int i;

 cout << " Size (int) = " << sizeof(int) << endl;
 cout << " Size ( i ) = " << sizeof( i ) << endl;
}
```

- Wir werden meist den Grundtyp `int` für den entsprechenden Teilbereich der ganze Zahlen und `unsigned int` für natürliche Zahlen verwenden. Die Kennzeichnung `unsigned` kann auch in Verbindung mit anderen Integertypen verwendet werden.

2.1.2 Bezeichnung von Variablen

Variablennamen beginnen mit Buchstaben oder Unterstrich, folgende Zeichen dürfen auch Zahlen sein. Nicht erlaubt ist die Verwendung von Leerzeichen und Operatorzeichen (§ 3) im Namen, gleichfalls dürfen Variablennamen keine Schlüsselwörter der C++ -Syntax sein (siehe Lit.).

Ex210.cc

Gültig	Ungültig	Grund
`i`		
`j`		
`ijl`		
`i3`	`3i`	3 ist kein Buchstabe
`_3a`	`_3*a`	* ist Operatorzeichen
`Drei_mal_a`	`b-a`	- ist Operatorzeichen
`auto1`	`auto`	`auto` ist Schlüsselwort

- C/C++ unterscheidet zwischen Groß- und Kleinschreibung, d.h., `ToteHosen` und `toteHosen` sind unterschiedliche Bezeichner!
- Laut originalem C-Standard sind die ersten 8 Zeichen eines Variablenbezeichners signifikant, d.h., `a2345678A` und `a2345678B` würden nicht mehr als verschiedene Bezeichner wahrgenommen. Mittlerweile sehen Compiler mehr Zeichen als signifikant an (C9X Standard: 63 Zeichen).

2.2 Konstanten

Die meisten Programme, auch *HelloWorld.cc*, verwenden im Programmverlauf unveränderliche Werte, sogenannte Konstanten.

2.2.1 Integerkonstanten

Dezimalkonstanten (Basis 10):

```
100        // int;     100
512L       // long;    512
128053     // long; 128053
```

Oktalkonstanten (Basis 8):

```
020        // int;      16
01000L     // long;    512
0177       // int;     127
```

Hexadezimalkonstanten (Basis 16):

```
0x15       // int;      21
0x200      // int;     512
0x1ffffl   // long; 131071
```

2.2.2 Gleitkommakonstanten

Gleitkommakonstanten werden stets als `double` interpretiert.
Einige Beispiele im folgenden:

```
17631.0e-78
1E+10              //   10000000000
1.                 //   1
.78                //   0.78
0.78
-.2e-3             //  -0.0002
-3.25
```

2.2.3 Zeichenkonstanten (Characterkonstanten)

Die Characterkonstante beinhaltet das Zeichen zwischen den beiden ' :

```
'a', 'A', '@', '1' //  ASCII-Zeichen
' '                //    Leerzeichen
'_'                //    Unterstreichung/Underscore
'\''               //    Prime-Zeichen '
'\\'               //    Backslash-Zeichen  \
'\n'               //  neue Zeile
'\0'               //  Nullzeichen NUL
```

2.2.4 Zeichenkettenkonstanten (Stringkonstanten)

Die Zeichenkette beinhaltet die Zeichen zwischen den beiden " :

```
"Hello World\n"
""                 //    leere Zeichenkette
"A"                //    String "A"
```

Jede Zeichenkette wird automatisch mit dem (Character-) Zeichen `'\0'` abgeschlossen (*"Hey, hier hört der String auf!"*). Daher ist `'A'` ungleich `"A"`, welches sich aus den Zeichen `'A'` und `'\0'` zusammensetzt und somit 2 Byte zur Speicherung benötigt.

Ex224.cc

```
/* Demo fuer Char / String - Konstante  */
#include <iostream.h>

main()
{
 cout << "A" << "  string : " << sizeof("A") << endl;
 cout << 'A' << "  char   : " << sizeof('A') << endl;
}
```

2.2.5 Symbolische Konstanten (Macros)

Wird eine der in den vorigen Abschnitten benutzten Konstanten mehrfach benötigt vergibt man für diese Konstante einen symbolischen Namen, z.B.

```
#define  NL     '\n'
#define  N      5
#define  HELLO  "Hello World\n"
```

Ex224.cc

oder allgemein

```
#define  <bezeichner>  <konstante>
```

Bemerkungen:

- Der Präprozessor ersetzt im restlichen Quelltext jedes Auftreten von `<bezeichner>` durch `<konstante>`, d.h., aus
  ```
  cout << HELLO;
  ```
  wird
  ```
  cout << "Hello World\n";
  ```
- Üblicherweise werden in diesen Bezeichnern keine Kleinbuchstaben verwendet, da z.B., `MAX_AUTO` dann sofort als symbolische Konstante erkennbar ist.

2.2.6 Konstante mit Variablennamen

Wird eine Variablenvereinbarung zusätzlich mit dem Schlüsselwort `const` gekennzeichnet, so kann diese Variable nur im Vereinbarungsteil initialisiert werden und danach nie wieder, d.h., sie wirkt als Konstante. Ex226.cc

```
// Constants and variables

#include <iostream.h>

main()
{
 const int  N = 5;     // The only initialization of constant
       int  i, j = 5;  // First    initialization of variable

 cout << "Hello World\n";

 i = j + N;

 cout << endl << i << " " << j << " " << N << endl;
}
```

Unterschied:

`#define  N  5`	Es wird kein Speicherplatz für `N` benötigt, da `N` im gesamten Quelltext durch `5` ersetzt wird.
`const int N = 5;`	Variable `N` wird gespeichert, das Programm arbeitet mit ihr.

Kapitel 3

Ausdrücke, Operatoren und mathematische Funktionen

- **Ausdrücke** bestehen aus Operanden und Operatoren.
- sind Variablen, Konstanten oder wieder Ausdrücke.
- **Operatoren** führen Aktionen mit Operanden aus.

3.1 Zuweisungsoperator

Der Zuweisungsoperator `<operand_A> = <operand_B>` weist dem linken Operanden, welcher eine Variable sein muß, den Wert des rechten Operanden zu.

Zum Beispiel ist im Ergebnis der Anweisungsfolge

```
{
 int x,y;

 x = 0;
 y = x + 4;
}
```

der Wert von `x` gleich 0 und der Wert von `y` gleich 4. Hierbei sind `x`, `y`, `0`, `x+4` Operanden, wobei letzterer gleichzeitig ein Ausdruck, bestehend aus den Operanden `x`, `4` und dem Operator `+`, ist. Sowohl `x = 0` als auch `y = x + 4` sind Ausdrücke. Erst das abschließende Semikolon `;` wandelt diese Ausdrücke in auszuführende Anweisungen!

Ex310.cc

Es können auch Mehrfachzuweisungen auftreten. Die folgenden drei Zuweisungen sind äquivalent.

```
{
 int a,b,c;

 a = b = c = 123;       // 1. Moeglichkeit

 a = (b = (c = 123);    // 2. Moeglichkeit

 c = 123;               // 3. Moeglichkeit (Standard)
 b = c;
 a = b;
}
```

3.2 Arithmetische Operatoren

3.2.1 Unäre Operatoren

Bei unären Operatoren tritt nur ein Operand auf.

Operator	Beschreibung	Beispiel
`-`	Negation	`-a`

3.2.2 Binäre Operatoren

Bei binären Operatoren treten zwei Operanden auf. Der Ergebnistyp der Operation hängt von den Operatoren ab.

Operator	Beschreibung	Beispiel
`+`	Addition	`b + a`
`-`	Subtraktion	`b - a`
`*`	Multiplikation	`b * a`
`/`	Division (! bei Integer-Werten !)	`b / a`
`%`	Rest bei ganzzahliger Division	`b % a`

Ex320.cc

Die Division von Integerzahlen berechnet den ganzzahligen Anteil der Division, d.h., `8 / 3` liefert `2` als Ergebnis. Falls aber der Wert `2.666666` herauskommen soll, muß mindestens einer der Operatoren in eine Gleitkommazahl umgewandelt werden, wie im Beispiel zu sehen ist.

```
{
 int   i,j ;
 float ij_mod, ij_div, float_ij_div;

 i = 8;
 j = 3;

 ij_div = i / j;                    // Attention: result is 2
 ij_mod = i % j;                    // Modulu

                                    // now: result is 2.666666
 float_ij_div = i/(float)j;         // explicit or
 float_ij_div = i/(j+0.);           // implicit type conversion
}
```

Bzgl. der Vorrangregeln für Operatoren sei auf die Literatur verwiesen, die alte Regel "*Punktrechnung geht vor Strichrechnung*" gilt auch in C/C++. Analog zur Schule werden Ausdrücke in runden Klammern `( <ausdruck> )` zuerst berechnet.

Ex320.cc

```
{
 int k;
 double x = 2.1;

 k = 1;                  // k stores 1

 k = 9/8;                // k stores 0, Integer division

 k = 3.14;               // k stores 3, truncated

 k = -3.14;              // k stores -3 or -4, compiler dependent

 k = 2.9e40;             // undefined

 x = 9/10;               // x stores 0

 x = (1+1)/2;            // x stores 1.0

 x = 1 + 1.0/2;          // x stores 1.5

 x = 0.5 + 1/2;          // x stores 0.5
}
```

3.3 Vergleichsoperatoren

Vergleichsoperatoren sind binäre Operatoren. Der Ergebniswert ist immer ein Integerwert, wobei FALSCH/false den Wert `0` liefert und WAHR/true einen Wert ungleich `0` .

Operator	Beschreibung	Beispiel
`>`	größer	`b > a`
`>=`	größer oder gleich	`b >= 3.14`
`<`	kleiner	`a < b/3`
`<=`	kleiner oder gleich	`b*a <= c`
`==`	gleich (! bei Gleitkommazahlen!)	`a == b`
`!=`	ungleich (! bei Gleitkommazahlen!)	`a != 3.14`

Ex330.cc

```
{
 bool  bi,bj;
 int   i;

 bi = ( 3 <= 4 );
 bj = ( 3 >  4 );

 cout << " 3 <= 4    TRUE  = " << bi << endl;
 cout << " 3 >  4    FALSE = " << bj << endl;

//               if - statement will be defined in Sec. 4
 i = 3;
 if (i <= 4)
   {
     cout << "\n i less or equal 4 \n\n";
   }
}
```

Ein **typischer Fehler** tritt beim Test auf Gleichheit auf, indem statt des Vergleichsoperators == der Zuweisungsoperator = geschrieben wird. Der Compiler akzeptiert beide Quelltexte, eventuell (compilerabhängig) wird beim falschen Code eine Warnung ausgegeben.

```
{
//           Incorrect Code
 int i;

 i = 2;
 if ( i = 3 )              // Assignment i=3 is always true !!
   {
     cout << " BB: i =  " << i << endl;    // i is 3
     i = 0;
   }
 cout << " CC: i = " << i << endl;         // i is always 0 !!
}
```

Im inkorrekten Code tritt der unerwünschte Nebeneffekt auf, daß der Wert der Variablen i im Test geändert wird, während folgender, korrekter Code keinerlei Nebeneffekte aufweist.

```
{
//           Correct Code
 int i;

 i = 2;
 if ( i == 3 )                           // Correct comparison
   {
     cout << " BB: i =  " << i << endl;  // i always remains 2
     i = 0;
   }
 cout << " CC: i = " << i << endl;       // i is still 2 !!
}
```

3.4 Logische Operatoren

Es gibt nur einen unären logischen Operator:

Operator	Beschreibung	Beispiel
`!`	logische Negation	`! (3>4)  // TRUE`

und zwei binäre logische Operatoren:

Operator	Beschreibung	Beispiel
`&&`	logisches UND	`(3>4) && (3<=4)  // FALSE`
`||`	logisches ODER	`(3>4) || (3<=4)  // TRUE`

Die Wahrheitswertetafeln für das logische UND und das logische ODER sind aus Algebra bekannt (ansonsten, siehe Literatur). Ex340.cc

```
{
 const int Ne = 5;              // one limit
       int i;

 cout << " i = " ;
 cin  >> i;                     // Input i

 if ( i <= Ne && i >= 0  )      // other limit is 0
  {
    cout << "i between 0 and 5" << endl;
  }
}
```

3.5 Bitorientierte Operatoren

Ein Bit ist die kleinste Informationseinheit mit genau zwei möglichen Zuständen:

$$\begin{Bmatrix}\text{bit gelöscht}\\ \text{bit gesetzt}\end{Bmatrix} \equiv \begin{Bmatrix}0\\ L\end{Bmatrix} \equiv \begin{Bmatrix}0\\ 1\end{Bmatrix} \equiv \begin{Bmatrix}\texttt{false}\\ \texttt{true}\end{Bmatrix}$$

Ein Byte besteht aus 8 Bit und damit ist eine `short int` Zahl 16 Bit lang.

Als Operatoren in Bitoperationen treten normalerweise Integer-Ausdrücke auf.

3.5.1 Unäre bitorientierte Operatoren

Operator	Beschreibung	Beispiel
`~`	Binärkomplement, bitweise Negation des Operanden	`~k`

3.5.2 Binäre bitorientierte Operatoren

Operator	Beschreibung	Beispiel	
`&`	bitweises UND der Operanden	`k & l`	
`|`	bitweises ODER	`k | l`	
`^`	bitweises exklusives ODER	`k ^ l`	
`<<`	Linksverschiebung der Bits von `<op1>` um `<op2>` Stellen	`k << 2`	`// = k*4`
`>>`	Rechtsverschiebung der Bits von `<op1>` um `<op2>` Stellen	`k >> 2`	`// = k/4`

Wahrheitswertetafel:

x	y	x & y	x | y	x ^ y
0	0	0	0	0
0	L	0	L	L
L	0	0	L	L
L	L	L	L	0

Diese Operatoren seien an den folgenden Beispielen demonstriert:

```
// bitwise operators

#include <iostream.h>

main()
{
 short int   k,l;
 short int   n1,n2,n3,n4,n5,n6,n7;

 l = 5;          //                    0..000L0L  = 5
 k = 6;          //                    0..000LL0  = 6

 n1 = ~k;        // Complement         L..LLL00L  = -7 = -6 - 1
 n2 = k & l;     // bit-AND            0..000L00  = 4
 n3 = k | l;     // bit-OR             0..000LLL  = 7
 n4 = k ^ l;     // bit-XOR            0..0000LL  = 3
 n5 = k << 2;    // shift  left by 2   0..0LL000  = 24 = 6 * 2^2
 n6 = k >> 1;    // shift right by 1   0..0000LL  = 3  = 6 / 2^1
 n7 = l >> 1;    // shift right by 1   0..0000L0  = 2  = 5 / 2^1
}
```

Ex350.cc

Die Bitoperationen sind nützlich beim Test, ob eine gerade oder ungerade Intergerzahl vorliegt. Das niederwertigste Bit kann bei Integerzahlen zur Unterscheidung gerade/ungerade Zahl genutzt werden (siehe auch die Bitdarstellung der Zahlen 5 und 6 im obigen Code). Wenn man daher dieses Bit mit einem gesetzten Bit über die ODER-Operation verknüpft, so bleibt das niederwertigste Bit bei ungeraden Zahlen unverändert. Dies wird im nachfolgenden Code ausgenutzt.

```
// mask for odd numbers

#include <iostream.h>

main()
{
 const int   Maske = 1;                //   0..00000L
       int   i;

 cout << "   Zahl: ";
 cin  >> i;                            //   read number

 cout << "  " << i << " ist eine ";

// Check for odd number:
//    Last bit remains unchanged for odd numbers

 if ((i | Maske) == i)
  {
    cout << "ungerade";
  }
 else
  {
    cout << "gerade";
  }
 cout << " Zahl." << endl << endl;
}
```

Ex351.cc

3.6 Operationen mit vordefinierten Funktionen

3.6.1 Mathematische Funktionen

Im Headerfile *math.h* werden u.a. die Definitionen der in Tabelle 3.1 zusammengefaßten mathematischen Funktionen und Konstanten bereitgestellt:

Das Runden einer reellen Zahl `x` erreicht man durch `ceil(x+0.5)` (ohne Beachtung der Rundungsregeln bei z.B., 4.5).

Funktion/Konstante	Beschreibung
`sqrt(x)`	Quadratwurzel von x: $\sqrt[2]{x}$ $(x \geq 0)$
`exp(x)`	e^x
`log(x)`	natürlicher Logarithmus von x: $\log_e x$ $(x > 0)$
`pow(x,y)`	Potenzieren ($x > 0$ falls y nicht ganzzahlig)
`fabs(x)`	Absolutbetrag von x: $\|x\|$
`fmod(x,y)`	realzahliger Rest von x/y $(y \neq 0)$
`ceil(x)`	nächste ganze Zahl $\geq x$
`floor(x)`	nächste ganze Zahl $\leq x$
`sin(x)`, `cos(x)`, `tan(x)`	trigonometrische Funktionen
`asin(x)`, `acos(x)`	trig. Umkehrfunktionen $(x \in [-1, 1])$
`atan(x)`	trig. Umkehrfunktion
`M_E`	Eulersche Zahl e
`M_PI`	π

Tabelle 3.1: Mathematische Funktionen

Für die Zulässigkeit der Operationen, d.h., den Definitionsbereich der Argumente, ist der Programmierer verantwortlich. Ansonsten werden Programmabbrüche oder unsinnige Ergebnisse produziert.

Ex361.cc

```
// Math. functions
#include <iostream.h>
#include <math.h>

main()
{
 double x,y,z;

 x = -1;                //   x < 0 !!
 y = sqrt(x);           //   Square root with wrong argument

 cout << "x = " << x << " ,  y  = " << y << endl;

                        //   Absolut value
 z = fabs(x);
 cout << "x = " << x << " , |x| = " << z << endl;

                        //   Power function
 y = 3.0;               //   try 2.0 , 3.0 and 2.5
 z = pow(x,y);
 cout << "(x,y) = " << x << " , " << y
                         << " , x^y = " << z << endl;
}
```

Die Funktionen aus *math.h* werden in einer speziellen mathematischen Bibliothek gespeichert, sodaß der Befehl zum Compilieren und Linken diese Bibliothek *libm.a* berücksichtigen muß, d.h.

```
LINUX> g++ Ex361.cc [-lm]
```

3.6.2 Funktionen für Zeichenketten (Strings)

Im Headerfile *string.h* werden u.a. die Definitionen der folgenden Funktionen für Strings bereitgestellt: Ex362.cc

Funktion	Beschreibung
`strcat(s1,s2)`	Anhängen von `s2` an `s1`
`strcmp(s1,s2)`	Lexikographischer Vergleich der Strings `s1` und `s2`
`strcpy(s1,s2)`	Kopiert `s2` auf `s1`
`strlen(s)`	Anzahl der Zeichen in String `s` (= `sizeof(s1)-1`)
`strchr(s,c)`	Sucht Character `c` in String `s`

Tabelle 3.2: Klassische Funktionen für Strings

```
// String functions
#include <iostream.h>
#include <string.h>              // definitions
main()
{
//    Definition and initialization of string variables
//        --> Sec. 5.1

 char s[30], s1[30] = "Hello", s2[] = "World";
 int  i;

 cout << "s1 = " << s1 << endl;
 cout << "s2 = " << s2 << endl;

 i = strcmp(s1,s2);             // lex. comparison

 cout << "cmp : " << i << endl;

 strcpy(s,s1);                  // copy s1 on s

 cout << "s  : " << s << endl;

 strcat(s,s2);                  // Appends s2 on s

 cout << "s  : " << s << endl;

 i = strlen(s);                 // length of string s

 cout << "Length of s : " << i << endl;
}
```

Details über diese Funktionen (und weitere) können mittels
`LINUX> man 3 string`
`LINUX> man strcmp`
erhalten werden.

3.7 Inkrement- und Dekrementoperatoren

3.7.1 Präfixnotation

```
++<lvalue>              // <lvalue> = <lvalue> + 1
--<lvalue>              // <lvalue> = <lvalue> - 1
```

```
//     Example: prefix notation
{
 int i=3, j;

 ++i;             // i = 4

 j = ++i;         // i = 5, j = 5
                  // above prefix notation is equivalent to
 i = i + 1;
 j = i;
}
```

3.7.2 Postfixnotation

```
<lvalue>++              // <lvalue> = <lvalue> + 1
<lvalue>--              // <lvalue> = <lvalue> - 1
```

```
//     Example: postfix notation
{
 int i=3, j;

 i++;             // i = 4

 j = i++;         // i = 5, j = 4
                  // above postfix notation is equivalent to
 j = i;
 i = i + 1;
}
```

Prä- und Postfixnotation sollten sparsam verwendet werden, meist benutzt man diese für eine Indexvariable in Zyklen (§ 4).

3.8 Zusammengesetzte Zuweisungen

Wertzuweisungen der Form

`<lvalue> = <lvalue> <operator> <ausdruck>`

können zu

`<lvalue> <operator>= <ausdruck>`

verkürzt werden.
Hierbei ist `<operator>` ∈ `{+,-,*,/,%,&,|,^,<<,>>}` aus § 3.2 und § 3.5 .

```
{
 int   i,j,w;
 float x,y;

 i += j           // i = i+j
 w >>= 1;         // w = w >> 1 (= w/2)
 x *=y;           // x = x*y
}
```

3.9 Weitere nützliche Konstanten

Für systemabhängige Zahlbereiche, Genauigkeiten usw. ist die Auswahl der folgenden Konstanten recht hilfreich.

Funktion	Beschreibung
`FLT_DIG`	Anzahl gültiger Dezimalstellen für `float`
`FLT_MIN`	Kleinste, darstellbare positive Zahl
`FLT_MAX`	Größte, darstellbare positive Zahl
`FLT_EPSILON`	Kleinste positive Zahl mit $1.0 + \varepsilon \neq 1.0$ (Stellenauslöschung)
`DBL_`	wie oben für `double`
`LDBL_`	wie oben für `long double`

Tabelle 3.3: Einige, wenige Konstanten aus *float.h*

Funktion	Beschreibung
`INT_MIN`	Kleinste, darstellbare Integerzahl
`INT_MAX`	Größte, darstellbare positive Integerzahl
`SHRT_`	wie oben für `short int`
`LONG_`	wie oben für `long int`
`LLONG_`	wie oben für `long long int`

Tabelle 3.4: Einige, wenige Konstanten aus *limits.h*

Weitere Konstanten können unter den gängigen Linuxdistributionen direkt in den Files */usr/lib/gcc-lib/i686-pc-linux-gnu/3.2.3/include/float.h* und */usr/include/limits.h* nachgeschaut werden. Die entspechenden Headerfiles können auch mit dem Befehl

```
LINUX> find /usr -name float.h -print
```

gesucht werden.

Kapitel 4

Kontrollstrukturen

4.1 Einfache Anweisung

Eine einfache Anweisung setzt sich aus einem Ausdruck und dem Semikolon als Abschluß einer Anweisung zusammen:

```
<ausdruck> ;
```

Beispiele:
```
cout << "Hello World" << endl;
i = 1 ;
```

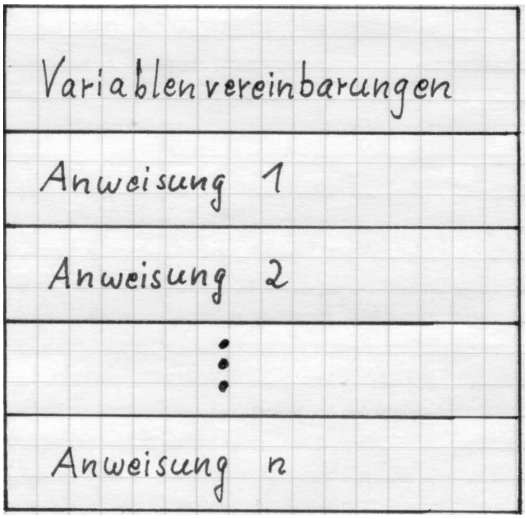
4.2 Block

Der Block (auch Verbundanweisung) ist eine Zusammenfassung von Vereinbarungen und Anweisungen mittels geschweifter Klammern:

```
{
  <anweisung_1>
       ...
  <anweisung_n>
}
```

```
//        Example block
{                  // Blockanfang
  int i,n;              // Vereinbarung

  i = 0;                // Anweisung
  n = i+1;              // Anweisung
}                  // Blockende
```

Struktogramm:



- In C muß der Vereinbarungsteil dem Blockanfang direkt folgen. In C++ können mehrere Vereinbarungsteile im Block existieren, sie müssen nur vor der jeweiligen Erstbenutzung der Variablennamen stehen. Aus Gründen der Übersichtlichkeit im Programm sollte dies aber nicht ausgenutzt werden.

- Der schließenden Klammer des Blockendes "}" folgt kein Semikolon.

- Ein Block kann stets anstelle einer Anweisung verwendet werden.

- Blöcke können beliebig ineinander geschachtelt werden.

- Die in einem Block vereinbarten Variablen sind nur dort sichtbar, d.h., außerhalb des Blocks ist die Variable nicht existent (Lokalität). Umgekehrt kann auf Variablen des übergeordneten Blocks zugegriffen werden.

Ex420.cc

```
// Block
#include <iostream.h>
main()
{
 int   i,j;                      // outer i

 i = j = 1;
 {                               // Begin inner block
   int k;
   int i;                        // inner i

   i = k = 3;
   cout << "  inner i = " << i << endl;
   cout << "i_outer j = " << j << endl;
 }                               // End inner block
 cout << "  outer i = " << i << endl;
 cout << "  outer j = " << j << endl;

// j = i+k;                     // k undeclared !!
}
```

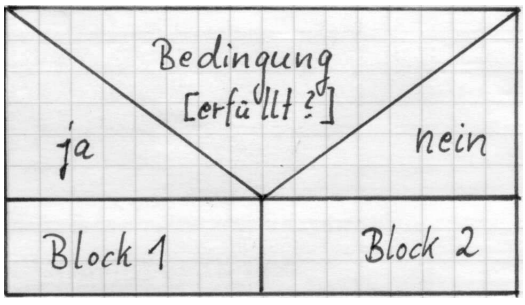
4.3 Verzweigungen

Die allgemeine Form der Verzweigungen (auch Alternative) ist

```
if ( <logischer ausdruck> )
  <anweisung_A>
else
  <anweisung_B>
```

und zählt ihrerseits wiederum als Anweisung. Der `else` -Zweig kann weggelassen werden (einfache Alternative).

Struktogramm:



Wie so oft kann ein konkretes Problem auf verschiedene Weise programmiert werden.

Beispiel: Wir betrachten dazu die Berechnung der Heaviside-Funktion

$$y(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

Ex431.cc

und stellen vier Varianten der Implementierung vor.

```
//            Rahmenprogramm
#include <iostream.h>
main()
{
 double x,y;

 cout << endl << " Input Argument      : ";
 cin  >> x;
//                            Version a

//                            Version b

//                            Version c

//                            Version d

}
```

Variante a: einfache Alternative

```
//            Version a
{
 y = 0.0 ;
 if ( x >= 0. )
     y = 1.0 ;        // genau eine Anweisung im if-Zweig

 cout << " Result of version a) : " << y << endl;
}
```

Variante b: zweifache Alternative

```
//            Version b
{
 if ( x >= 0.0 )
     y = 1.0 ;
 else
     y = 0.0 ;

 cout << " Result of version b) : " << y << endl;
}
```

Variante c: zweifache Alternative mit Blöcken

```
//            Version c
{
 if ( x >= 0.0 )
   {
     y = 1.0 ;
   }
 else
   {
     y = 0.0 ;
   }

 cout << " Result of version c) : " << y << endl;
}
```

Variante d: Entscheidungsoperator.
Treten in einer zweifachen Alternative in jedem Zweig nur je eine Wertzuweisung zur selben Variablen auf (wie in Versionen b) und c)), dann kann der Entscheidungsoperator

```
<log. ausdruck> ? <ausdruck_A> : <ausdruck_B>
```

verwendet werden.

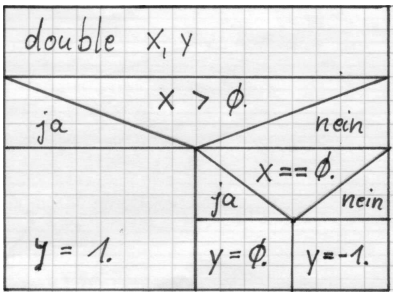
```
//            Version d
{
 y = (x >= 0) ? 1.0 : 0.0 ;

 cout << " Result of version d) : " << y << endl;
}
```

Beispiel: Ein weiteres Beispiel ist die Berechnung der Signum-Funktion (Vorzeichenfunktion)

$$y(x) = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$$

und wir stellen mehrere Varianten der Implementierung vor. Ex432.cc

Struktogramm:



Wir betrachten zwei Implementierungsvarianten, das Rahmenprogramm ist identisch zum Rahmenpogramm auf Seite 28.

Variante a: Schachtelung der Alternativen

```
//            Version a
{
 if ( x > 0.0 )
   {
     y = 1.0 ;
   }
 else
   {
     if ( x == 0.0 )
       {
         y = 0.0 ;
       }
     else
       {
         y = -1.0 ;
       }
   }
 cout << " Result of version a) : " << y << endl;
}
```

Variante b: Falls der `else`-Zweig nur aus einer weiteren `if-else`-Anweisung besteht, kann Variante a leicht modifiziert werden.

```
//            Version b
{
 if ( x > 0.0 )
   {
     y = 1.0 ;
   }
 else if ( x == 0.0 )
   {
     y = 0.0 ;
   }
 else
   {
     y = -1.0 ;
   }

 cout << " Result of version b) : " << y << endl;
}
```

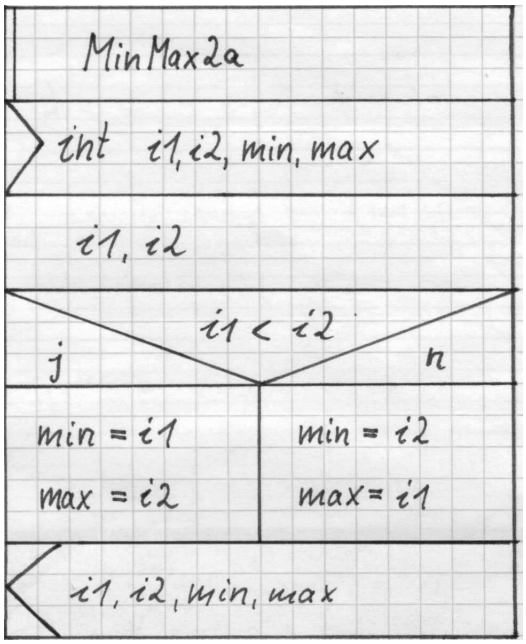
Allgemein kann eine solche Mehrwegentscheidung als

```
if ( <logischer ausdruck_1> )
  <anweisung_1>
else if ( <logischer ausdruck_2> )
  <anweisung_2>
        ...
else if ( <logischer ausdruck_(n-1)> )
  <anweisung_(n-1)>
else
  <anweisung_n>
```

geschrieben werden, wobei der `else`-Zweig wiederum optional ist.

Beispiel: Bestimmung von Minimum und Maximum zweier einzugebender Zahlen. Ex433.cc

Struktogramm:

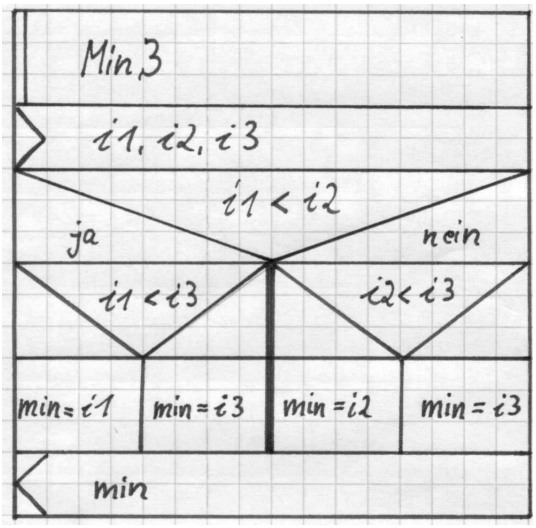


```
// Example: Maximum and Minimum of two numbers

#include <iostream.h>

main()
{
 int i1,i2,min,max;

 cout << endl << " Input Arguments   i1 i2 : ";
 cin  >> i1 >> i2 ;

 if ( i1 < i2 )
   {
     min = i1 ;
     max = i2 ;
   }
 else
   {
     min = i2 ;
     max = i1 ;
   }

 cout << " Min,Max  (a) : " << min << " , " << max << endl;
}
```

Ex434.cc

Beispiel: Bestimmung des Minimums dreier einzugebender Zahlen.

Struktogramm:



```
// Example: Minimum of three numbers

#include <iostream.h>

main()
{
 int i1,i2,i3,min;

 cout << endl << " Input Arguments   i1 i2 i3  : ";
 cin  >> i1 >> i2 >> i3;

 if ( i1 < i2 )
   {
     if ( i1 < i3 )
       {
         min = i1;
       }
     else
       {
         min = i3;
       }
   }
 else
   {
     if ( i2 < i3 )
       {
         min = i2;
       }
     else
       {
         min = i3;
       }
   }
 cout << " Min  (a) : " << min  << endl;
}
```

4.4 Der Zählzyklus (`for`-Schleife)

Beim Zählzyklus steht die Anzahl der Zyklendurchläufe a-priori fest, der Abbruchtest erfolgt vor dem Durchlauf eines Zyklus. Die allgemeine Form ist

```
for (<ausdruck_1>; <ausdruck_2>; <ausdruck_3>)
  <anweisung>
```

Am besten sei der Zählzyklus an einem Beipiel erläutert.

Beispiel: Es ist die Summe der ersten 5 natürlichen Zahlen zu berechnen. Ex440.cc

```
// Example : sum of natural numbers
#include <iostream.h>

main()
{
 int i,isum,n;                    // loop index, sum, last index

 n = 5;                           // initialize last index

 isum = 0;                        // initialize sum (integer !)
 for ( i = 1; i <= n; i=i+1)
   {
     isum = isum + i;
   }
 cout << endl << "Sum of first " << n
      << " natural numbers = " << isum << endl;
}
```

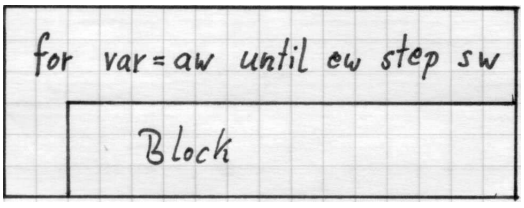
Im obigen Programmbeispiel ist `i` die Laufvariable des Zählzyklus, welche mit `i = 1` (`<ausdruck_1>`) initialisiert, mit `i = i+1` (`<ausdruck_3>`) weitergezählt und in `i <= n` (`<ausdruck_2>`) bzgl. der oberen Grenze der Schleifendurchläufe getestet wird. Im Schleifeninneren `sum = sum + i;` (`anweisung`) erfolgen die eigentlichen Berechnungsschritte des Zyklus. Die Summationsvariable `sum` muß vor dem Eintritt in den Zyklus initialisiert werden.

Eine kompakte Version dieser Summationsschleife (korrekt, aber sehr schlecht lesbar) wäre :
`for (isum = 0, i = 1; i <= n; isum += i, i++)`
Man unterscheide dabei zwischen dem Abschluß einer Anweisung "`;`" und dem Trennzeichen "`,`" in einer Liste von Ausdrücken. Diese Listen werden von links nach rechts abgearbeitet.

Der `<ausdruck_2>` ist stets ein logischer Ausdruck (§ 3.3-3.4) und `<ausdruck_3>` ist ein arithmetischer Ausdruck zur Manipulation der Laufvariablen, z.B.

```
i++
j = j-2
j += 2
x = x+h          // float-Typ
k = 2*k          // Verdoppelung
l = l/4          // Viertelung - Vorsicht bei Integer
```

Struktogramm:



- Die Laufvariable kann eine einfache Variable aus § 2.1 sein, z.B., `int` oder `double` .

- Vorsicht bei Verwendung von Gleitkommazahlen (`float`, `double`) als Laufvariable. Dort ist der korrekte Abbruchtest wegen der internen Zahldarstellung u.U. nicht einfach zu realisieren.

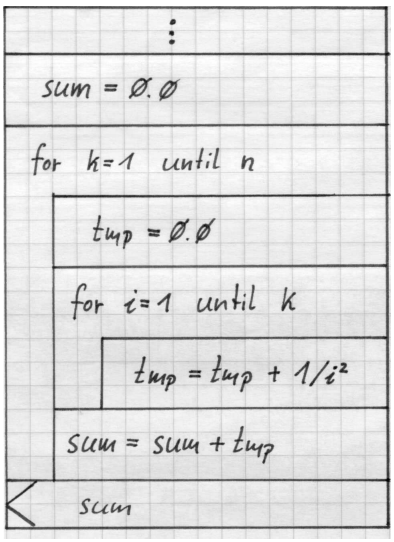
Loop_Float.cc

Beispiel: Es sei die Doppelsumme

$$\text{sum} = \sum_{k=1}^{n} \underbrace{\sum_{i=1}^{k} \frac{1}{i^2}}_{t_k} = \sum_{k=1}^{n} t_k$$

für einzugebende n zu berechnen.

Struktogramm:



Ex442.cc

```
// Example: double sum
#include <iostream.h>
main()
{
 int    i,k,n;                  // loop index, sum, last index
 double sum_i,sum_k;            // outer and inner sum

 cout << "  Input n : "; cin  >> n;     // read n

 sum_k = 0.0;                           // initialize outer sum

 for ( k = 1; k <= n; k++)
  {
    sum_i = 0.0;                        // initialize inner sum
    for ( i = 1; i <= k; i++)           // last index depends on k !!
     {
        sum_i = sum_i + 1.0/i/i;
     }
    cout << "  Sum (" << k << ") = " << sum_i << endl;
    sum_k = sum_k + sum_i;              // sum_k grows unbounded
  }
 cout << "  Double-Sum (" << n << ") = " << sum_k << endl;
}
```

Weitere einfache **Beispiele** berechnen die Summe der ersten geraden natürlichen Zahlen und das Zählen eines CountDowns.

Ex443.cc

Ex444.cc

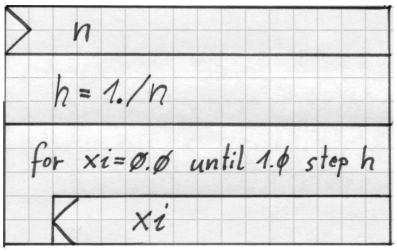
Die folgenden Beispiele verdeutlichen die Problematik der begrenzten Genauigkeit von Gleitkommazahlen in Verbindung mit Zyklen und einige Tips zu deren Umgehung.

Loop_Float.cc

Beispiel: Ausgabe der diskreten Knoten x_i des Intervalls $[0, 1]$, welches in n gleichgroße Teilintervalle zerlegt wird, d.h.,

$$x_i = i \cdot h \quad , \; i = 0, \ldots, n \qquad \text{mit } h = \frac{1-0}{n}$$

Struktogramm:




```
main()
{
 float xa,xe,xi,h;
 int   n;

 cin  >> n;          // # subintervals
 xa = 0.0e0;         // # start interval
 xe = 1.0e0;         // # end interval
 h   = (xe-xa)/n;    // length subinterval

 for (xi = xa; xi <= xe; xi += h)
   {
    cout << xi << endl;
   }
}
```

Da Gleitkommazahlen nur eine limitierte Anzahl gültiger Ziffern besitzen, kann es (meistens) passieren, daß der letzte Knoten x_n nicht ausgegeben wird. Nur für $n = 2^k$, $k \in \mathbb{N}$ kann in unserem Beispiel eine korrekte Abarbeitung des Zählzyklus garantiert werden. Auswege sind

1. Änderung des Abbruchtests in `xi <= xe + h/2.0` , jedoch ist x_n immer noch fehlerbehaftet.

```
for (xi = xa; xi <= xe + h/2.0; xi += h)
  {
   cout << xi << endl;
  }
```

2. Zyklus mit `int`-Laufvariable

```
for (i = 0; i <= n; i++)
  {
   xi = xa + i*h;
   cout << xi << endl;
  }
```

Die gemeinsame Summation kleinerer und größerer Zahlen kann ebenfalls zu Ungenauigkeiten führen. Im **Beispiel** wird die Summe $s1 := \sum_{i=1}^{n} 1/i^2$ mit der (theoretisch identischen) Summe $s2 := \sum_{i=n}^{1} 1/i^2$ für große n (65.000, 650.000) verglichen.

Reihe.cc

```
#include <iostream.h>
#include <math.h>
#include <float.h>

main()
{
 float s1,s2;
 int i,n ;

 cout << "The first sum will be rather precise until  n = "
     << ceil(sqrt(1./FLT_EPSILON)) << endl;
 cin >> n;

 s1 = 0.0;
 for (i=1; i<=n; i++)
   {
     s1 += 1.0/i/i;
   }
 cout << s1 << endl;

 s2 = 0.0;
 for (i=n; i>=1; i--)
   {
     s2 += 1.0/i/i;
//   s2 += 1.0/(i*i);     results in inf
//                        since i*i is longer than int supports
   }
 cout << s2 << endl;
 cout << s2-s1 << endl;
}
```

Das numerische Resultat in $s2$ ist genauer, da dort zuerst alle kleinen Zahlen addiert werden, welche bei $s1$ wegen der beschränkten Anzahl gültiger Ziffern keinen Beitrag zur Summation mehr liefern können. Gleichzeitig ist zu beachten, daß die Berechnung von `1.0/(i*i)` in einem Überlauf endet, da `i*i` nicht mehr in `int`-Zahlen darstellbar ist. Dagegen erfolgt die Berechnung von `1.0/i/i` vollständig im Bereich der Gleitkommazahlen.

4.5 Abweisender Zyklus (`while`-Schleife)

Beim abweisenden Zyklus steht die Anzahl der Durchläufe nicht a-priori fest, der Abbruchtest erfolgt **vor** dem Durchlauf eines Zyklus.

Die allgemeine Form ist

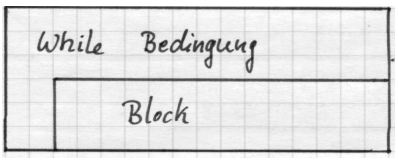
```
while (<logischer ausdruck>)
  <anweisung>
```

Ex450.cc

Beispiel: Bestimme den aufgerundeten Binärlogarithmus (Basis 2) einer einzulesenden Zahl.

```
//         Example : Binary log. of a number
#include <iostream.h>
main()
{
 double x,xsave;
 int    cnt;

 cout << endl << " Eingabe x : " ;
 cin  >> x;
 xsave = x;                              // Save to restore x

 cnt = 0;                                // Initialize  cnt
 while ( x > 1.0 )
   {
     x   = x/2.0 ;
     cnt = cnt + 1;
   }
 cout << endl << "Binary log. of " << xsave
                                   << " = " << cnt << endl;
}
```

Struktogramm:



Bemerkung: Falls der allererste Test im abweisenden Zyklus FALSE ergibt, dann wird der Anweisungsblock im Zyklusinneren nie ausgeführt (der Zyklus wird abgewiesen).

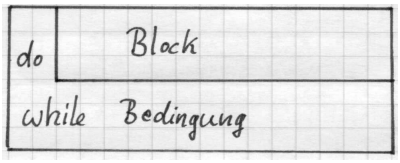
4.6 Nichtabweisender Zyklus (`do-while`-Schleife)

Beim nichtabweisenden Zyklus steht die Anzahl der Durchläufe nicht a-priori fest, der Abbruchtest erfolgt **nach** dem Durchlauf eines Zyklus. Somit durchläuft der nichtabweisende Zyklus mindestens einmal die Anweisungen im Zyklusinneren.

Die allgemeine Form ist

```
do
  <anweisung>
while (<logischer ausdruck>) ;
```

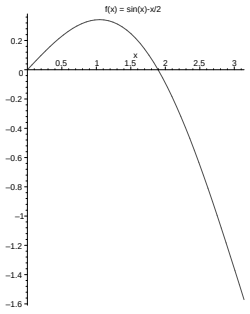
Struktogramm:

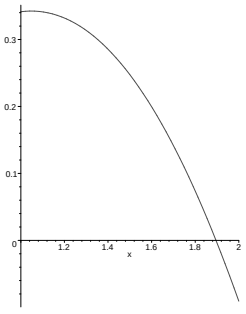


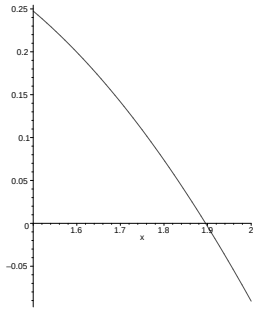
Beispiel: Es wird solange ein Zeichen von der Tastatur eingelesen, bis ein x eingegeben wird. Ex460.cc

```
// Example : Input of a character until 'x'
#include <iostream.h>
main()
{
 char ch;

 do
   {
     cout << endl << "Input command (x = exit, ...)  ";
     cin  >> ch;
   }
 while ( ch != 'x' );

 cout << endl << "          Exit program"
      << endl << endl;
}
```

Betrachten wir ein etwas anspruchsvolleres **Beispiel**, und zwar soll die Lösung von $\sin(x) = x/2$ mit $x \in (0, \pi)$ bestimmt werden. Hierzu betrachtet man die äquivalente Nullstellenaufgabe: Bestimme die Nullstelle $x_0 \in (0, \pi)$ der Funktion $f(x) := \sin(x) - x/2 = 0$.
Analytisch: Kein praktikabler Lösungsweg vorhanden.
Graphisch: Die Funktion $f(x)$ wir graphisch dargestellt und das Lösungsintervall manuell verkleinert (halbiert). Diesen Prozeß setzt man so lange fort, bis x_0 genau genug, d.h., auf eine vorbestimmte Anzahl von Stellen genau, bestimmt werden kann.





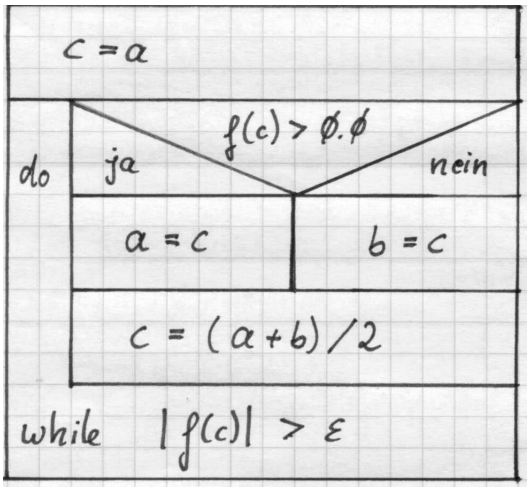


Ex462.mws

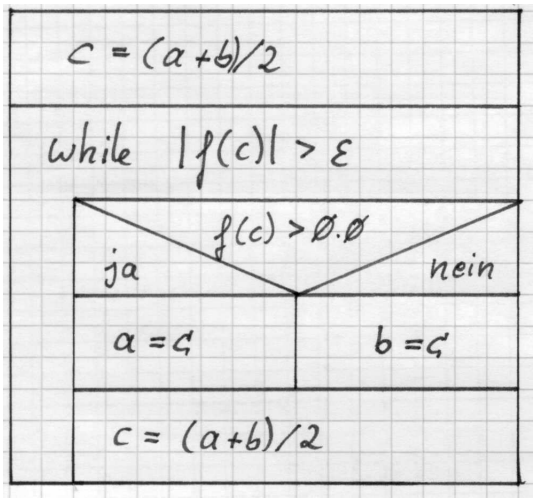
Numerisch: Obiges, graphisches Verfahren kann auf ein rein numerisches Verfahren im Computer übertragen werden (der *MAPLE*-Aufruf `fsolve(sin(x)=x/2,x=0.1..3` liefert als Näherungsergebnis $x_0 = 1.895494267$). Wir entwickeln ein Programm zur Bestimmung der Nullstelle von $f(x) := \sin(x) - x/2$ im Intervall $[a, b]$ mittels Intervallhalbierung, wobei zur Vereinfachung angenommen wird, daß $f(a) > 0$ und $f(b) < 0$ ist. Der Mittelpunkt des Intervalls sei mit $c := (a + b)/2$ bezeichnet. Dann können wir über die Lösung folgendes aussagen: $\begin{cases} x_0 := c & \text{falls } f(c) = 0 \\ x_0 \in [c, b] & \text{falls } f(c) > 0 \\ x_0 \in [a, c] & \text{falls } f(c) < 0 \end{cases}$.
Durch Redefinition der Intervallgrenzen a und b kann die Nullstellensuche auf das kleinere (halbierte) Intervall reduziert werden. Wir demonstrieren die Umsetzung mittels eines nichtabweisenden Zyklus.

Ex462.cc

Struktogramm:



Obige Bisektion kann auch mittels eines abweisenden Zyklus realisiert werden.



```
//   Nullstellenberechnung durch Bisektion in [a,b]
#include <iostream.h>
#include <math.h>

main()
{
 const double Eps = 1e-6;
 double a,b,c,fc;

 ...
 cin >> a; cin >> b;
//      Check that f(a) > 0,  f(b) < 0 !!
 ...
//              Do-While loop
  c  = a;             // since f(a) > 0
  fc = sin(c)-c/2;
  do
   {
    if ( fc > 0.0 )
     {
       a = c;
     }
    else
     {
       b = c;
     }
    c  = (a+b)/2.0;
    fc = sin(c)-c/2;
   }
  while ( fabs(fc) > Eps);
//  while ( fabs(fc) != 0.0);   // endless!! Why ?

  cout << " Nullstelle = " << c << endl;
}
```

Da Gleitkommazahlen nur mit limitierter Genauigkeit arbeiten resultiert ein Abbruchtest $f(c) = 0$ meist in einem endlosen Programm. Dem ist ein Abbruchtest wie $|f(c)| < \varepsilon$ mit einer vorgegebenen Genauigkeit $0 < \varepsilon \ll 1$ vorzuziehen.

Bemerkung: Zählzyklen (`for`), welche mindestens einen Zyklus ausführen, können sowohl durch abweisende (`while`) als auch durch nichtabweisende Zyklen (`do while`) äquivalent ausgedrückt werden. Diese Äquivalenz kann bei Verwendung der Anweisungen in § 4.8 verloren gehen. Falls in einem Zählzyklus der Abbruchtest stets FALSE ergibt, d.h. der Schleifenkörper wird nie ausgeführt, dann ist der entsprechende abweisende Zyklus nach wie vor äquivalent. Jedoch ist der nichtabweisende Zyklus nicht mehr äquivalent, da der dortige Schleifenkörper auch in diesem Fall einmal abgearbeitet wird. Siehe das Beispielfile *Loops.cc*.

Loops.cc

4.7 Mehrwegauswahl (`switch`-Anweisung)

Die Mehrwegauswahl ermöglicht ein individuelles Reagieren auf spezielle Werte einer Variablen.

```
switch (<ausdruck>)
 {
   case <konst_ausdruck_1> :
        <anweisung_1>
[break;]
   ...
   case <konst_ausdruck_n> :
        <anweisung_n>
[break;]
   default:
        <anweisung_default>
 }
```

Ex470.cc

Beispiel: Ausgabe der Zahlwörter für die ganzzahlige Eingaben $\{1, 2, 3\}$.

```
// Demonstration of Switch statement (break !!)
#include <iostream.h>
main()
{
 int number;

 number = 2,

 cout << endl << "  Names of numbers in [1,3]" << endl;

 switch(number)
  {
    case 1:
        cout << "  One = " << number << endl;
        break;
    case 2:
        cout << "  Two = "  << number << endl;
        break;              // Comment this line
    case 3:
        cout << "  Three = "  << number << endl;
        break;
    default:
        cout << "  Number " << number
             << " not in interval" << endl;
        break;              // not necessary
  }
 cout << endl;
}
```

Obige `switch`-Anweisung könnte auch mit einer Mehrfachverzweigung (Seite 31) implementiert werden, jedoch werden in der `switch`-Anweisung die einzelnen

Zweige explizit über die `break;`-Anweisung verlassen. Ohne `break;` wird zusätzlich der zum nachfolgenden Zweig gehörige Block abgearbeitet.

4.8 Anweisungen zur unbedingten Steuerungsübergabe

`break` Es erfolgt der sofortige Abbruch der nächstäußeren `switch`, `while`, `do-while`, `for` Anweisung.

`continue` Abbruch des aktuellen und Start des nächsten Zyklus einer `while`, `do-while`, `for` Schleife. Ex480.cc

`goto <marke>` Fortsetzung des Programmes an der mit
`<marke> : <anweisung>`
markierten Stelle.

Bemerkung : Bis auf `break` in der `switch`-Anweisung sollten obige Anweisungen sehr sparsam (besser gar nicht) verwendet werden, da sie dem strukturierten Programmieren zuwiderlaufen und den gefürchteteten Spaghetticode erzeugen. Im Praktikum sind obige Anweisungen zur Lösung von Übungsaufgaben etc. nicht erlaubt.

Kapitel 5

Strukturierte Datentypen

Wir werden in diesem Kapitel neue Möglichkeiten der Datenspeicherung einführen.

- **Feld (array)**:
  Zusammenfassung von Elementen gleichen Typs.

- **Struktur (struct)**:
  Zusammenfassung von Komponenten verschiedenen Typs.

- **Union (union)**:
  Überlagerung mehrerer Komponenten verschiedenen Typs auf dem gleichen Speicherplatz.

- **Aufzählungstyp (enum)**
  Grunddatentyp mit frei wählbarem Wertebereich.

5.1 Felder (Arrays)

5.1.1 Eindimensionale Felder

In einem Feld werden Daten (Elemente) gleichen Typs zusammengefaßt. Die allgemeine Vereinbarung eines statischen Feldes ist

```
<typ> <bezeichner>[dimension];
```

wobei die eckigen Klammern "[" und "]" unabdingbarer Bestandteil der Vereinbarung sind. Ein eindimensionales Feld entspricht mathematisch einem Vektor.

Ex510.cc

```
//        Example array
{
 const int N=5;
 double x[N], y[10];       // Declaration

 x[0] = 1.0;               //  Manipulations
 x[1] = -2;
 x[2] = -x[1];
 x[3] = x[1]+x[2];
 x[4] = x[1]*x[2];

// access to x[5] , i.e., x[N] is not permitted

}
```

Die eckigen Klammern dienen im Vereinbarungsteil der Dimensionsvereinbarung `x[N]` und im Anweisungsteil dem Zugriff auf einzelne Feldelemente `x[3]` . Das Feld kann schon bei Deklaration initialisiert werden:
`double x[N] = {9,7,6,5,7}`

Achtung : Die Numerierung der Feldelemente beginnt mit 0. Daher darf nur auf Feldelemente x_i, $i = 0, \ldots, N-1$ zugegriffen werden. Andernfalls sind mysteriöses Programmverhalten, unerklärliche Fehlberechnungen und plötzliche Programmabstürze zu erwarten, deren Ursache nicht offensichtlich ist da sie eventuell erst in weit entfernten Programmteilen auftreten können.

Typischer Fehler

```
//        Typical error
{
 const int N = 123;
       int ij[N] , i;
 ...
 for (i = 1; i <= N; i++)    // !! WRONG !!
   {
     cout << ij[i] << endl;
   }
}
```

Es werden die Feldelemente ij_1, ij_2, ij_3, ij_4 und der unsinnige Wert von ij_5 ausgegeben, jedoch nicht das allererste Feldelement ij_0 .

Die Dimension eines statischen Feldes muß zum Zeitpunkt der Compilierung bekannt sein, daher dürfen nur Konstanten oder aus Konstanten bestehende Ausdrücke als Dimension auftreten.

```
{
 const int N = 5, M = 1;
       int size;
 float x[5];            //  Correct
 short i[N];            //  Correct
 char  c[N-M+1];        //  Correct
 int   ij[size];        //  !! WRONG !!
}
```

Beispiel: Ein interessanter Spezialfall des Feldes ist die Zeichenkette (String). Wir initialisieren den String mit dem Wort "Mathematik" und geben ihn in Normalschrift und zeichenweise aus.

```
//                   String variables
#include <iostream.h>
#include <string.h>
main()
{
 const int L=11;                // 10+1
 char word[L];
 int  i;

 strcpy(word,"Mathematik");     // initialization

 cout << endl << word << endl;

 for (i = 0; i < L; i++)
  {
    cout << word[i] << " " ;
  }
 cout << endl;
}
```

Ex511.cc

Die Zeichenkette hätte auch mit
`char word[L] = "Mathematik";`
oder
`char word[] = "Mathematik";`
initialisiert werden können, wobei in letzterem Fall die Länge des Feldes `word` aus der Länge der Zeichenkettenkonstante bestimmt wird.

Beispiel: Berechnung der L_2-Norm eines Vektors, d.h., $\| \underline{x} \|_{L_2} := \sqrt{\sum_{i=0}^{N-1} x_i^2}$

Ex512.cc

```
// Array: L_2 n
#include <iostream.h
#include <math.h>
main()
{
 const int N=10;
 double x[N], norm;
//                          Initialize x
 for (i = 0; i < N ; i++)
   {
     x[i] = sqrt(i+1.0);
   }
//                          L_2 norm calculation
 norm = 0.0;
 for (i = 0; i < N; i++)
   {
     norm += x[i]*x[i];
   }
 norm = sqrt(norm);

 cout << "  L2-norm : " << norm << endl;
}
```

Als kleines **Beispiel** diene uns die Fibonacci Zahlenfolge, welche über die zweistufige Rekursion

$$f(n)_{:} = f(n-1) + f(n-2) \qquad n = 2, \ldots$$

mit den Anfangsbedingungen $f(0) = 0$, $f(1) = 1$ definiert ist. Zur Kontrolle können wir die Formel von Binet bzw. de Moivre[1] verwenden. Fibo1.cc

$$f(n) = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$$

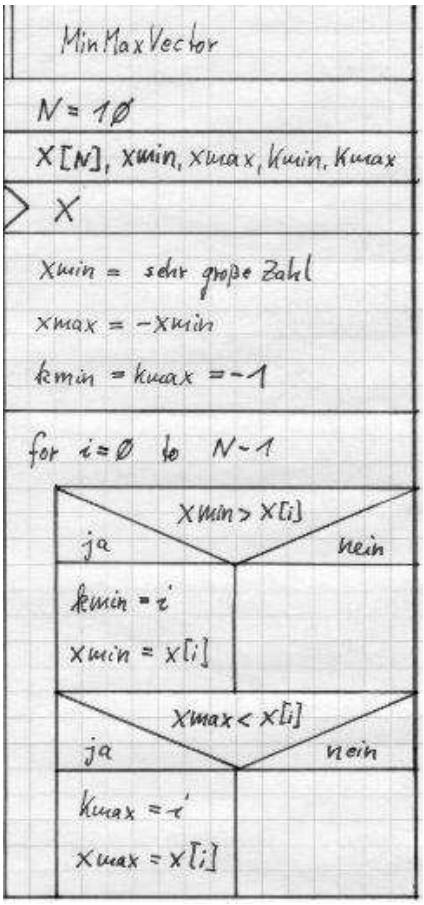
```
//                      Demo of Fibonacci numbers
#include <iostream.h>
#include <math.h>
main()
{
  const int N = 20;
        int i;
        int x[N+1];     //  !!   N+1   !!
     double fib;
//                      Calculate Fibonacci numbers
  x[0] = 0; x[1] = 1;
  for ( i = 2; i <= N; i++ )
     x[i] = x[i-1] + x[i-2];
//                      Output x
  ...
///                     Check last Fibonacci number
  fib = ( pow(0.5*(1.0+sqrt(5.0)),N)
         -pow(0.5*(1.0-sqrt(5.0)),N) )/sqrt(5.0);

  cout << "fib(" << N << ") = " << fib << endl;
}
```

[1] http://www.ee.surrey.ac.uk/Personal/R.Knott/Fibonacci/fibFormula.html

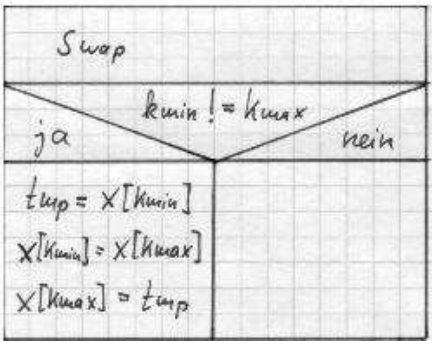
Ex513.cc

Als weiteres **Beispiel** sollen Minimum und Maximum eines Vektors bestimmt und die entsprechenden Vektorelemente miteinander vertauscht werden (analog zu Pivotisierung). Dies beinhaltet die beiden Teilaufgaben:

a) Bestimme Minimum und Maximum (und markiere die Positionen).
Struktogramm:



b) Vertausche Min/Max-Einträge. Bei Vektorlänge 0 oder bei identischen Vektorelementen ist kein Vertauschen notwendig.
Struktogramm:



Beim Vertauschen führt die naheliegende, erste Idee

```
x[kmin] = x[kmax]
x[kmax] = x[kmin]
```

nicht zum Erfolg. Warum?

```
// Pivot for a vector
#include <iostream.h>
#include <float.h>
main()
{
 const int N=10;
 double x[N], xmin, xmax, tmp;
 int    kmin, kmax, i;
//                                Initialize x
 for (i = 0; i < N ; i++)
   {
     cin >> x[i];
   }
//                                Initialize min/max
 xmin =  DBL_MAX; // in floats.h
 xmax = -DBL_MAX;
//  Initialize indices
 kmin = kmax = -1;
//  Determine min/max
 for (i = 0; i < N; i++)
   {
     if ( xmin > x[i] )
      {
       xmin = x[i];
       kmin = i;
      }
     if ( xmax < x[i] )
      {
       xmax = x[i];
       kmax = i;
      }
   }
// Swap Pivot elements
// Do nothing for N=0 or constant vector
 if ( kmax != kmin )
  {
   tmp     = x[kmin];
   x[kmin] = x[kmax];
   x[kmax] = tmp;
  }
//  Print Pivot vector
  ...
}
```

5.1.2 Mehrdimensionale Felder

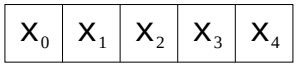
Die Einträge der bisher betrachteten 1D-Felder sind im Speicher hintereinander gespeichert (Modell des linearen Speichers), z.B, wird der Zeilenvektor

$$\begin{pmatrix} x_0 & x_1 & x_2 & x_3 & x_4 \end{pmatrix}$$

als

```
double x[5];
```

vereinbart und gespeichert als

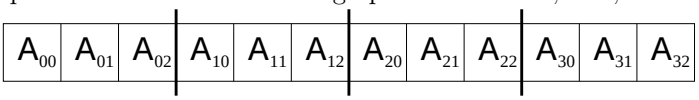


wobei jede Zelle 8 Byte lang ist.

Ein zweidimensionales (statisches) Feld, z.B., eine Matrix A mit $N = 4$ Zeilen und $M = 3$ Spalten

$$A_{N\times M} := \begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \\ A_{30} & A_{31} & A_{32} \end{pmatrix}$$

kann im Speicher ebenfalls nur linear gespeichert werden, d.h.,



Daraus ergeben sich zwei Möglichkeiten der 2D-Feldvereinbarung:

- Variante 1 : Als 2D-Array.
```
double A[N][M];          // Declaration
A[3][1] = 5.0;           // Initialize A(3,1)
```
- Variante 2 : Als 1D-Array.
```
double A[N*M];           // Declaration
A[3*M+1] = 5.0;          // Initialize A(3,1)
```

Beispiel: Als Beispiel betrachten wir die Multiplikation der Matrix $A_{N\times M}$ bestehend aus $N = 4$ Zeilen und $M = 3$ Spalten mit einem Zeilenvektor $\underline{u}_M$ der Länge M. Das Ergebnis ist ein Zeilenvektor $\underline{f}_N$ der Länge N, d.h., $\underline{f}_N := A_{N\times M} \cdot \underline{u}_M$. Die Komponenten von $\underline{f} = [f_0, f_1, \ldots, f_{N-1}]^T$ berechnen sich zu

Ex514.cc

$$f_i := \sum_{j=0}^{M-1} A_{i,j} \cdot u_j \qquad \forall i = 0, \ldots, N-1 \ .$$

Höherdimensionale Felder können analog zu Version 1 deklariert und benutzt werden. In Variante 2 muß auf ein Element $B(i, j, k)$ eines dreidimensionalen Feldes `double B[L,N,M];` mittels `B[i*M*N+j*M+k]` zugegriffen werden.

5.2 Strukturen

Die Struktur definiert einen neuen Datentyp welcher Komponenten unterschiedlichen Typs vereint. Die Typdeklaration

```
struct <struct_bezeichner>
 {
   <Datendeklaration>
 };
```

erlaubt die Deklaration von Variablen diesen Typs

```
<struct_bezeichner> <var_bezeichner>;
```

Beispiel: Wir deklarieren einen Datentyp zur Speicherung der persönlichen Daten eines Studenten. Ex520.cc

```
// Structure
{
// new structure
 struct Student
  {
    long long int matrikel;
                int skz;
                char name[30], vorname[20];
  };
// Variable of type Student
 Student arni,robbi;
// Data input
 cout << endl << " Vorname : ";

 cin >> arni.vorname;
...
 robbi = arni;         // complete copy
 cout << robbi.vorname << endl;
}
```

Die Zuweisung `robbi = arni;` kopiert den kompletten Datensatz von einer Variablen zur anderen. Der Zugriff auf die Komponente `vorname` der Variablen `arni` (des Typs `Student`) erfolgt über
`arni.vorname`
Abgespeichert werden die Daten in der Form

matrikel	skz	name	vorname

Abhängig von Compilereinstellungen bzw. -optionen können kleinere ungenutzte Speicherlücken zwischen den Komponenten im Speicher auftreten (Data Alignment für schnelleren Datenzugriff).

Die Struktur `Student` kann leicht für Studenten, welche mehrere Studienrichtungen belegen, erweitert werden. Ex520b.cc

```
{
 const int MAX_SKZ=5;

 struct Student_Mult
  {
    long long int matrikel;
              int skz[MAX_SKZ];
      int nskz; // number of studies
              char name[30], vorname[20];
  };
// Variable of type Student
 Student arni,robbi;
// Data input
 cout << endl << " Vorname : ";

 cin >> arni.vorname;
...
 robbi = arni;         // complete copy
 cout << robbi.vorname << endl;
}
```

Ex522.cc

Die Struktur `Student` enthält bereits Felder als Komponenten. Andererseits können diese Datentypen wiederum zu Feldern arrangiert werden.

```
// Array of structures
{
 struct Student               // new structure
  {
    ...
  };
 const int N = 20;
       int i;
 Student gruppe[N];          // Array
                             // Init
 for (i = 0; i < N; i++)
  {
   cin >> gruppe[i].vorname;
   ...
  }
 ...
}
```

Ex523.cc

Strukturen können wiederum andere strukturierte Datentypen als Komponenten enthalten.

```
// Structures in structures
{
 struct Point3D               // simple structure
  {
    double x,y,z;
  };

 struct Line3D               // structure uses Point3D
  {
    Point3D p1,p2;
  };

 Line3D line;                // Declare variable

                             // Init
 cout << "Anfangspkt.: ";
 cin  >> line.p1.x >> line.p1.y >>line.p1.z;
 cout << "    Endpkt.: ";
 cin  >> line.p2.x >> line.p2.y >>line.p2.z;
 ...
}
```

In obigem Beispiel ist `line.p2` eine Variable vom Typ `Point3D` , auf deren Daten wiederum mittels des `.` Operators zugegriffen werden kann.

5.3 Union

Alle Komponenten der Union werden auf dem gleichen Speicherbereich überlappend abgebildet. Die Typdeklaration

```
union <union_bezeichner>
 {
   <Datendeklaration>
 };
```

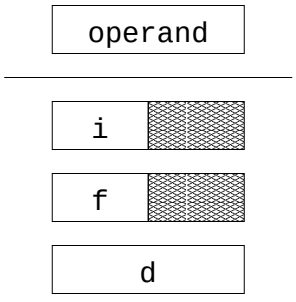
erlaubt die Deklaration von Variablen diesen Typs

```
[union] <union_bezeichner> <var_bezeichner>;
```

Der Zugriff auf Komponenten der Union erfolgt wie bei einer Struktur.

Ex530.cc

```
// Union
#include <iostream.h>
main()
{
 union operand           // new  union
  {
    int    i;
    float  f;
    double d;            // longest data
  };
 operand u;              // declare variable

 cout << endl << "Size (operand) : "
      << sizeof(u) << " Bytes" << endl;

 u.i = 123;              // Init as int
 cout << endl << u.i << "  " << u.f << "  " << u.d << endl;

 u.f = 123;              // Init as float
 cout << ...

 u.d = 123;              // Init as double
 cout << ...
}
```



Der Speicherplatzbedarf einer Union richtet sich nach der größten Komponente (hier `sizeof(double)` = 8). Die Union wird benutzt, um Speicherplatz zu sparen, sollte jedoch wegen der Fehlermöglichkeiten erfahrenen Programmierern vorbehalten bleiben (d.h., keine Verwendung im Praktikum).

5.4 Aufzählungstyp

Der Aufzählungstyp ist ein Grundtyp mit frei wählbarem Wertebereich, dies sei an Hand der Wochentage veranschaulicht.

```
// enum
#include <iostream.h>
main()
{
//                                new enum
 enum tag
  {
    montag, dienstag, mittwoch, donnerstag,
    freitag, samstag, sonntag
  };

 tag wochentag;              //  variable of enum
 wochentag = montag;         //  data init

 if ( wochentag == montag )
  {
   cout << "Schlechte Laune" << endl;
  }
}
```

Ex540.cc

C++ besitzt einen vordefinierten Typ `bool`, welcher die Werte `false` und `true` annehmen kann. In C läßt sich dies durch die Definition

```
enum bool {false,true}
```

in analoger Weise erreichen, wobei `false` durch 0 und `true` durch 1 repräsentiert werden was konform zu § 2.1.1 und § 3.3 ist.

5.5 Allgemeine Typdefinitionen

Die allgemeine Typdefinition

```
typedef <type_definition> <type_bezeichner>
```

ist die konsequente Weiterentwicklung zu frei definierbaren Typen.

Das nachfolgende Programmbeispiel illustriert die Definition der drei neuen Typen `Boolean`, `Text` und `Point3D`.

Ex550.cc

```
// general type definitions
main()
{
//                    new types
 typedef short int Boolean;
 typedef char      Text[100];
 typedef struct
  {
   double x,y,z;
  }                 Point3D;

//                    new variables
 Boolean a,b;
 Text    eintrag;
 Point3D pts[10], p = {1, 2, 3.45};
 ...
}
```

Interessanterweise ist eine Variable vom Typ `Text` nunmehr stets eine Zeichenkettenvariable der (max.) Länge 100. Man beachte auch die Initialisierung der Variablen `p`. Damit kann sogar eine Konstante vom Typ `Point3d` deklariert und initialisiert werden.

Kapitel 6

Zeiger (Pointer)

Bislang griffen wir stets direkt auf Variablen zu, d.h., es war nicht von Interesse, wo die Daten im Speicher abgelegt sind. Ein neuer Variablentyp, der Pointer (Zeiger), speichert Adressen unter Berücksichtigung des dort abgelegten Datentyps.

6.1 Vereinbarung von Zeigern

Sei der Zeiger auf ein Objekt vom Typ `int` mit `p` bezeichnet, so ist

```
int *p;
```

dessen Deklaration, oder allgemein wird durch

```
[speicherklasse] <typ> *<bezeichner>;
```

ein Zeiger auf den Datentyp `<typ>` definiert.
So können die folgenden Zeigervariablen definiert werden

```
//        Pointer declaration
{
 struct Student
  {
    ...
  };

 char            *cp;          // pointer on char
 int          x, *px;          // int-variable, pointer on int
 float           *fp[20];      // array of 20 pointers on float
 float           *(fap[10]);   // pointer on array of 10 float
 Student         *ps;          // pointer on structure Student
 char           **ppc;         // pointer on pointer of char
}
```

Ex610.cc

6.2 Zeigeroperatoren

Der unäre **Referenzoperator** (Adressoperator)

```
&<variable>
```

bestimmt die Adresse der Variablen im Operanden.

Der unäre **Dereferenzoperator** (Zugriffsoperator)

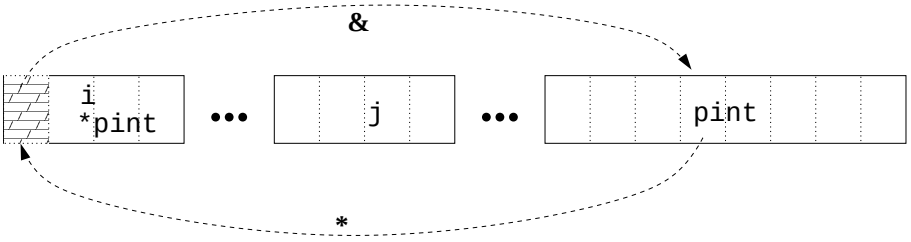
```
*<pointer>
```

erlaubt den (indirekten) Zugriff auf die Daten auf welche der Pointer zeigt. Die Daten können wie eine Variable manipuliert werden.

Ex620.cc

```
// Pointer operators
#include <iostream.h>
main()
{
  int i, j, *pint;

  i    = 10;                    // i  = 10
  pint = &i;          // pointer initialization
  j    = *pint;       // access on int

  *pint = 0;                    // i  = 0

  *pint += 2;                   // i += 2
}
```

In obigem Beispiel fungiert `*pint` als `int`-Variable und dementsprechend können auch alle dafür definierten Operationen mit ihr ausgeführt werden.



Achtung : In dem Programmfragment

```
{
  double *px;
  *px = 3.1;     // WRONG!
}
```

wird zwar Speicherplatz für den Zeiger reserviert (8 Byte), jedoch ist der Wert von `px` noch undefiniert und daher wird der Wert `3.1` in einen dafür nicht vorgesehenen Speicherbereich geschrieben
$\Longrightarrow$ mysteriöse Programmabstürze und -fehler.

Es gibt eine spezielle Zeigerkonstante `0` (`NULL` in C), welche auf die (hexadezimale) Speicheradresse `0x0` (= `nil`) verweist und bzgl. welcher eine Zeigervariable getestet werden kann.

6.3 Zeiger und Felder - Zeigerarithmetik

Felder nutzen das Modell des linearen Speichers, d.h., ein im Index nachfolgendes Element ist auch physisch im unmittelbar nachfolgenden Speicherbereich abgelegt. Dieser Fakt erlaubt die Interpretation von Zeigervariablen als Feldbezeichner und umgekehrt.

```
{
  const int N = 10;
        int f[N], *pint;  // array and pointer

  pint = &f[0];           // init pointer
}
```

Feldbezeichner werden prinzipiell als Zeiger behandelt, daher ist die Programmzeile

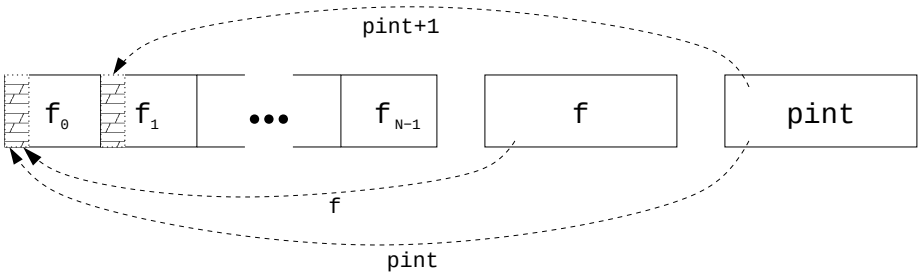
```
pint = &f[0];
```

identisch mit

```
pint = f;
```

Folgerichtig stellen daher die Ausdrücke `f[1]` , `*(f+1)` , `*(pint+1)` , `pint[1]` den identischen Zugriff auf das Feldelement f_1 dar.

Ex630.cc



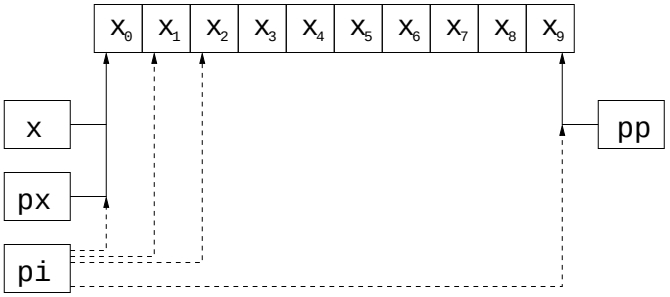
Die Adresse welche durch (`pint+1`) dargestellt wird ergibt sich zu (Adresse in `pint`) + `sizeof(int)` . Dabei bezeichnet `int` den Datentyp, auf welchen der Zeiger `pint` verweist. Der Zugriff auf andere Feldelemente f_i, $i = 0 \ldots N-1$ ist analog.

Die folgenden Operatoren sind auf Zeiger anwendbar:

- Vergleichsoperatoren: `==` , `!=` , `<` , `>` , `<=` , `>=`
- Addition `+` und Subtraktion `-`
- Inkrement `++` , Dekrement `--` und zusammengesetzte Operatoren `+=` , `-=`

Zur Demonstration betrachten wir ein **Beispiel**, in welchem ein Feld erst auf konventionelle Weise definiert und initialisiert wird um danach mittels Zeigeroperationen ausgegeben zu werden.

Ex630.cc




```
// Pointers and arrays
#include <iostream.h>
main()
{
  const int N=10;
  double    x[N], *px, *pp, *pi;
  int       i;

//                    initialize x
  px = x;
  for (i = 0; i < N; i++ )
   {
     *(px+i) = (i+1)*(i+1);    // x[i] = ...
   }
//              output x;
//                       pointer pi as loop variable
  pp = x+N-1;         // pointer at last element of x
  for ( pi = x; pi <= pp; pi++)
   {
     cout << "  " << *pi << endl;
   }
}
```

6.4 Dynamische Felder mittels Zeigervariablen

Bisher verwies ein Zeiger auf bereits bereitgestellten (allocated) Speicher für eine einfache Variable, Struktur, Feld. Für Zeiger kann aber auch ein zum Typ dazugehöriger Speicherbereich dynamisch allokiert werden. Hierzu benutzt man das neue Schlüsselwort `new` . Der damit allokierte Speicher kann mittels `delete` wieder freigegeben werden.

Ex641.cc

```
// Dynamic variable and
// Dynamic array 1D
#include <iostream.h>
main()
{
  int     n,i;
  double  *px, *pvar;

  cout << "  Eingabe n : ";  cin >> n;

  px = new double[n];           //  Allocate array

//  initialize array
  for (i = 0; i < n; i++ )
   {
     px[i] = (i+1)*(i+1);
   }

// output x;
  for ( i = 0; i < n; i++)
   {
     cout << "  " << px[i] << endl;
   }

  delete [] px;                    // Deallocate array

   pvar = new double;              // Allocate variable
  *pvar = 3.5 * n;
  delete pvar;                     // Deallocate variable
}
```

Die Anweisung `px = new double[n];` allokiert `n*sizeof(double)` Byte für den Zeiger `px`. Danach kann das dynamische Feld `px` wie ein statisches Feld behandelt werden. Allerdings nutzen dynamische Felder den vorhandenen Speicherplatz besser, da dieser mit dem `delete`-Befehl freigegeben und anderweitig wieder genutzt werden kann.

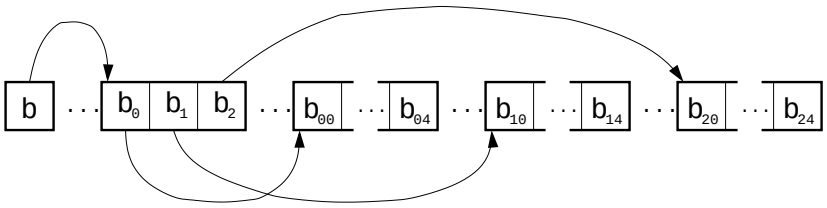
Achtung: Obige dynamische Feldvereinbarung ist nur für C++ gültig. In C müssen andere Befehle verwendet werden - hier die Unterschiede. Ex640.c

C++	C
	`#include <malloc.h>`
`px = new double[n];`	`px = (double*) malloc(n*sizeof(double));`
`delete [] px;`	free(px);

Ein **zweidimensionales dynamisches Feld** läßt sich einerseits durch ein eindimensionales dynamisches Feld darstellen (analog zu Variante 2 in § 5.1.2) als auch durch einen Zeiger auf ein Feld von Zeigern. Dies sieht für eine Matrix mit n Zeilen und m Spalten wie folgt aus. Ex642.cc

```
// Dynamic array 2D
#include <iostream.h>
main()
{
  int     n,m,i,j;
  double  **b;          // pointer at pointers at double

  cout << "  Eingabe Zeilen  n : ";  cin >> n;
  cout << "  Eingabe Spalten m : ";  cin >> m;

  b = new (double* [n]);      // Allocate row pointers
  for (i = 0; i < n; i++)
   {
     b[i] = new double[m];    // Allocate rows
   }

  for (i = 0; i < n; i++ )    // initialize b
   {
     for (j = 0; j < m; j++)
      {
        b[i][j] = (i+1)*(j+1);
      }
   }
...
  for (i = n-1; i >= 0; i--)
   {
     delete []  b[i];            //  Deallocate rows
   }
  delete [] b;                   //  Deallocate row pointers
}
```

Zuerst muß der Zeiger auf die Zeilenpointer allokiert werden, erst danach kann der Speicher für die einzelnen Zeilen angefordert werden. Beim Deallokieren des Speichers müssen ebenfalls alle Zeilen wieder freigegeben werden. Für den Fall $n = 3$ und $m = 4$ veranschaulicht das Bild die Ablage der Daten im Speicher.



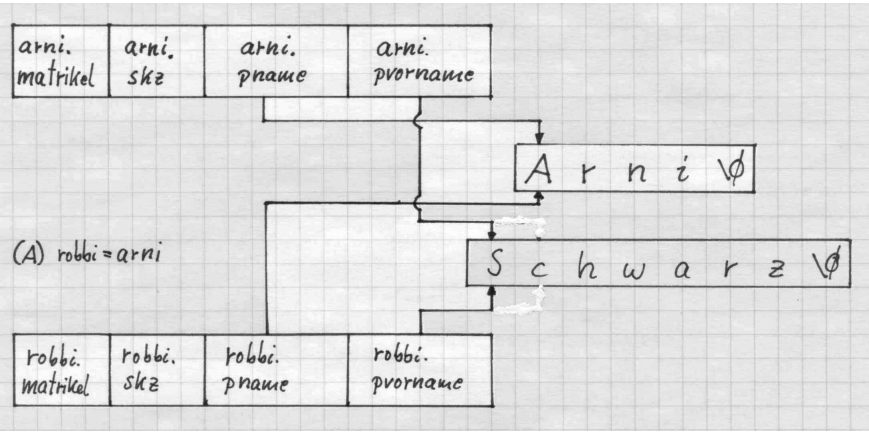
Achtung: Es gibt keine Garantie, daß die einzelnen Zeilen der Matrix hintereinander im Speicher angeordnet sind. Somit unterscheidet sich die Speicherung des dynamischen 2D-Feldes von der Speicherung des statischen 2D-Feldes obwohl die Syntax des Elementzugriffes `b[i][j]` identisch ist. Dafür ist diese Matrixspeicherung flexibler, da die Zeilen auch unterschiedliche Längen haben dürfen (dünnbesetzte Matrizen oder Matrizen mit Profil).

Zeiger können wiederum **in Strukturen** oder allgemeinen Typen auftreten. Hierbei ist allerdings größtmögliche Sorgfalt in der Verwendung der dynamischen Felder geboten, da für statische Variablen ansonsten unkritische Operationen plötzlich zu mirakulösem Programmverhalten führen können.

Ex643-warning.cc

```
// Demonstration of   w r o n g   code
// wrt. copying a structure with pointers
#include <iostream.h>
#include <strings.h>// strcpy, strlen

struct Student2
  {
    long long int matrikel;
              int skz;
              char *pname, *pvorname; // Pointers in structure
  };

main()
{
 Student2 arni, robbi;
 char     tmp[80];           // temp. input string

 cin  >> tmp;                // read vorname
// Allocate memory for arni.pvorname
 arni.pvorname = new char[strlen(tmp)+1];// Don't forget "+1
 strcpy(arni.pvorname,tmp);  // and copy input on it

// the same with the remaining data on arni
 ...
// rough and  w r o n g  copying
 robbi = arni;
// Point (A)
 ...
 delete [] arni.pvorname;  // deallocate
// Point (B)
 ...
//                    Let us allocate some tiny dynamical array
 char *tiny;
 tiny = new char [5];
 strcpy(tiny,"tiny");
// Point (C)
...
}
```

Wie sieht der Datenspeicher zu den Zeitpunkten (A), (B) und (C) aus?



- `robbi` besitzt keine eigenen dynamischen Felder.

- `delete [] arni.pvorname;`
  gibt somit auch den von `robbi.pvorname` adressierten Speicherbereich frei und somit zeigt `robbi.pvorname` auf einen nicht mehr reservierten Speicherbereich, welcher vom Programm nach Gutdünken verwendet werden darf.

- `tiny = new char [5];`
  nimmt sich den frei gewordenen Speicherplatz und überschreibt ihn später.

- Unter LINUX-gcc zeigt zum Zeitpunkt (C) `robbi.pvorname` auf die gleiche Adresse wie der Zeiger `tiny`, sodaß die Ausgabe der Daten von `robbi.pvorname` mittels
  `cout << robbi.pvorname << endl;`
  die Ausgabe `tiny` liefert.

- **Ausweg:**
  Es müssen für `robbi` eigene dynamische Felder allokiert werden und die Inhalte der dynamischen Felder von `arni` müssen auf diese kopiert werden. (siehe Zuweisungsoperatoren und Copykonstruktoren für Klassen, auch § 11).

Ex643-correct.cc

6.5 Zeiger auf Strukturen

Wir betrachten die Struktur `Student` (§ 5.2) und definieren einen Zeiger darauf.

```
// Pointer at structure
{
  struct Student
  {
   ...
  };
  Student peter, *pg;

//      init peter
  ...

  pg = &peter;           // pointer at peter

  cout << (*pg).vorname; // conventional access

  cout <<   pg->vorname; // better access
  ...
}
```

Die Zugriffe `(*pg).vorname` und `pg->vorname` sind völlig äquivalent. Allerdings verbessert letzterer deutlich die Lesbarkeit eines Programmes insbesondere, wenn der Zeiger ein dynamisches Feld des Typs `Student` darstellt. Dies zeigt sich insbesondere beim Zugriff auf Feldelemente von `vorname` (d.h., einzelne Zeichen). Der Zugriff auf das 0. Zeichen erfolgt mittles

Ex650.cc

`pg->vorname[0]`	oder	`*pg->vorname`	
oder	`(*pg).vorname[0]`	oder	`*(*pg).vorname`

und der Zugriff auf das 3. Zeichen mittels

`pg->vorname[3]`	oder	`*(pg->vorname+3)`	
oder	`(*pg).vorname[3]`	oder	`*((*pg).vorname+3)`

Beachten Sie, daß `pg->vorname` einen Zeiger auf den Typ `char` darstellt und der Dereferenzierungsoperator `*` vor der Addition `+` ausgeführt wird. Vermuten und testen Sie, was bei Verwendung von `*pg->vorname+3` herauskommt.

6.6 Referenz

Eine Referenz ist ein Alias (Pseudoname) für eine Variable und kann genauso wie diese benutzt werden. Referenzen stellen (im Gegensatz zu Zeigern) kein eigenes Objekt dar, d.h., für sie wird kein zusätzlicher Speicher angefordert.

```
// Reference
// i, ri, *pi are different names for one variable
#include <iostream.h>
main()
{
 int   i; // i
 int &ri = i; // declaration reference on i
 int *pi;

 pi = &i; // declaration pointer on i;

 i = 7;
 cout << i << ri << *pi;

 ri++;
 cout << i << ri << *pi;

 (*pi)++;
 cout << i << ri << *pi;
}
```

Referenzen werden häufig zur Parameterübergabe an Funktionen benutzt, siehe § 7.2. Eine weitere sinnvolle Anwendung besteht in der Referenz auf ein Feldelement, Strukturelement oder auf innere Daten einer komplizierten Datenstruktur wie nachstehend, vom Beispiel auf Seite 54 abgeleitet, gezeigt wird.

```
// Reference and dynamical array of type student
#include <iostream.h>
main()
{
 struct Student
  {
    long long int matrikel;
               int skz;
               char name[30], vorname[20];
  };

 Student gruppe[4];        // pointer at Student
// Data input;
 ...

 i = 3;
 {
// reference on comp. of structure
   int           &rskz = gruppe[i].skz;
// reference on structure
   Student       &rg   = gruppe[i];
// reference on comp. of referenced structure
   long long int &rm   = rg.matrikel;

   cout << "Student nr. " << i << " : ";
   cout << rg.vorname << " " << rg.name << " , ";
   cout << rm << " , " << rskz << endl;
 }
}
```

Ex662.cc

Kapitel 7

Funktionen

7.1 Definition und Deklaration

Zweck einer Funktion:

- Des öfteren wird ein Programmteil in anderen Programmabschnitten wieder benötigt. Um das Programm übersichtlicher und handhabbarer zu gestalten wird dieser Programmteil einmalig als Funktion programmiert und im restlichen Programm mit seinem Funktionsnamen aufgerufen.

- Bereits fertiggestellte Funktionen können für andere Programme anderer Programmierer zur Verfügung gestellt werden, analog zur Benutzung von `pow(x,y)` und `strcmp(s1,s2)` in § 3.6.

In der allgemeinen Form der Funktions**definition** mit

```
<speicherklasse> <typ>  <funktions_name> (parameter_liste)
{
 <vereinbarungen>
 <anweisungen>
}
```

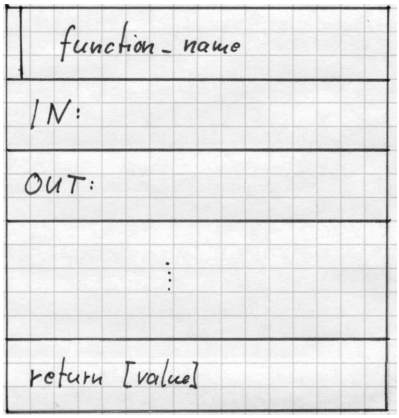
stellen Vereinbarungs- und Anweisungsteil den Funktionskörper dar und `<typ>` legt den Typ des Rückgabewertes fest. Die Kombination `<funktions_name>` und `(parameter_liste)` kennzeichnet eindeutig eine Funktion und wird daher als **Signatur** einer Funktion bezeichnet. Die Funktionsdefinition wird für jede Funktion genau einmal benötigt.

Im Unterschied dazu ist die Funktions**deklaration**

```
<speicherklasse> <typ>  <funktions_name> (parameter_liste) ;
```

in jedem Quellfile nötig welches die Funktion `<funktions_name>` aufruft.

Struktogramm:



Ex710.cc

Beispiel: Wir schreiben die Berechnung von $\mathrm{sgn}(x)$ von Seite 29 als Funktion.

```
// Demonstration of function declaration and definition

#include <iostream.h>

double sgn(double x);   // declare function sgn
...

main()
{
 double a,b;
 ...
 cin  >> a;

 b  = sgn(a);           // function call

 cout << "  sgn(" << a << ") = " << b << endl;
}

//-----------------------------------------

double sgn(double x)    // definition of function sgn
{
 double y;

 y = (x > 0 ? 1. : 0.) + (x < 0 ? -1. : 0.);

 return y;              // return value of function sgn
}
```

Bemerkungen: Die Funktion `sgn()` ist durch ihre Signatur eindeutig beschrieben. Dies hat für Deklaration und Definition von Funktionen die Konsequenzen:

(i) Einige weitere (oder noch mehr) identische Funktionsdeklarationen

```
double sgn(double x);
```

sind in obigem Beispiel erlaubt.

(ii) Zusätzliche Funktionsdeklarationen mit anderen Parameterlisten sind erlaubt, z.B.:

```
double sgn(double* x);
double sgn(int x);
```

da sich die Argumente von der Ausgangsdefinition unterscheiden. Allerdings haben wir diese neuen Funktionen noch nicht definiert.

(iii) Eine zusätzliche Deklaration (siehe § 7.2)

```
double sgn(double& x);
```

ist nicht erlaubt, da die Signatur wie unter (i) ist. Daher kann der Compiler nicht herausfinden, ob die Funktion unter (iii) oder die Funktion unter (i) in der Anweisung

```
y = sgn(x);
```

gemeint ist.

(iv) Verschiedene Funktionen mit gleichem Namen werden anhand ihrer unterschiedlichen Parameterlisten identifiziert, siehe Pkt. (iii).

(v) Der Rückgabewert einer Funktion kann nicht zu ihrer Identifikation herangezogen werden, die Deklarationen

```
double sgn(int x);
   int sgn(int x);
```

können nicht unterschieden werden (gleiche Signatur) und daher lehnt der Compiler diesen Quelltext ab.

7.2 Parameterübergabe

Beim Programmentwurf unterscheiden wir drei Arten von Parametern einer Funktion:

INPUT Parameterdaten werden in der Funktion benutzt aber nicht verändert, d.h., sie sind innerhalb der Funktion konstant.

INOUT Parameterdaten werden in der Funktion benutzt und verändert.

OUTPUT Parameterdaten werden in der Funktion initialisiert und gegebenenfalls verändert.

Wir werden programmtechnisch nicht zwischen INOUT- und OUTPUT-Parametern unterscheiden. Es gibt generell drei Möglichkeiten der programmtechnischen Übergabe von Parametern

Ex721.cc

1. Übergabe der Daten einer Variablen (engl.: by value).

2. Übergabe der Adresse einer Variablen (engl.: by address)

3. Übergabe der Referenz auf eine Variable (engl.: by reference), wobei hierbei versteckt eine Adresse übergeben wird.

Bemerkung:
Wenn eine Variable in der Funktion als Konstante benützt wird, dann sollte sie auch so behandelt werden, d.h., reine INPUT-Parameter sollten stets als `const` in der Parameterliste gekennzeichnet werden. Dies erhöht die Sicherheit vor einer unbeabsichtigten Datenmanipulation.

7.3 Rückgabewerte von Funktionen

Jede Funktion besitzt ein Funktionsergebnis vom Datentyp `<typ>` . Als Typen dürfen verwendet werden:

- einfache Datentypen (§ 2.1.1),
- Strukturen (§ 5.2), Klassen,
- Zeiger (§ 6.1),
- Referenzen (§ 6.6),

jedoch keine Felder und Funktionen - dafür aber Zeiger auf ein Feld bzw. eine Funktion und Referenzen auf Felder.

Der Rückgabewert (Funktionsergebnis) wird mit

```
return <ergebnis> ;
```

an das rufende Programm übergeben. Ein Spezialfall sind Funktionen der Art

```
void f(<parameter_liste>)
```

für welche kein Rückgabewert (void = leer) erwartet wird, sodaß mit

```
return ;
```

in das aufrufende Programm zurückgekehrt wird.

Wir betrachten die Möglichkeiten der Parameterübergabe am Beispiel der `sgn` Funktion mit der Variablen `double a` .

Übergabeart	Parameterliste	Aufruf	Effekt von `x++`	Effekt von `(*x)++`	Verwendung	Empfehlung
by value	`double x`	`sgn(a)`	intern	—-	**I**NPUT	[C]
	`const double x`		nicht erlaubt	—-	**I**NPUT	C [einfache Datentypen]
by address	`double* x`	`sgn(&a)`	intern	intern/extern	IN**O**UT	C
	`const double* x`		intern	nicht erlaubt	**I**NPUT	C [komplexe Datentypen]
	`double* const x`		nicht erlaubt	intern/extern	IN**O**UT	[C]
by reference	`double& x`	`sgn(a)`	intern/extern	—-	IN**O**UT	C++
	`const double& x`		nicht erlaubt	—-	**I**NPUT	C++

Tabelle 7.1: Möglichkeiten der Parameterübergabe

Die "by-reference"-Variante `double &const x` wird vom Compiler abgelehnt und die "by-address"-Variante `const double* const x` , d.h., Zeiger und Daten dürfen lokal nicht verändert werden, ist praktisch bedeutungslos.

Ex731.cc

```
// Demonstration of void
void spass(const int);

main()
{
 ...
 spass(13);
 ...
}

void  spass(const int i)
{
 cout << "Jetzt schlaegt's aber" << i << endl;
 return;
}
```

Beispiele für Funktionsergebnisse:

`float f1(...)`	float-Zahl
`[struct] Student f2(...)`	Struktur Student
`int* f3(...)`	Zeiger auf int-Zahl
`[struct] Student* f4(...)`	Zeiger auf Struktur Student
`int (*f5(...))[]`	Zeiger auf Feld von int-Zahlen
`int (*f6(...))()`	Zeiger auf Funktion, welche den Ergebnistyp int besitzt

Bemerkungen:
Eine Funktion darf mehrere Rückgabeanweisungen `return [<ergebnis>];` besitzen, z.B., in jedem Zweig einer Alternative eine. Dies ist jedoch kein sauber strukturiertes Programmieren mehr.
$\Longrightarrow$ Jede Funktion sollte genau eine `return`-Anweisung am Ende des Funktionskörpers besitzen (Standard für das Praktikum).

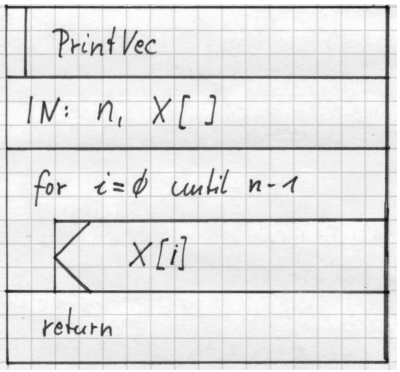
7.4 Felder als Parameter

Statische Felder können analog zu ihrer Deklaration als Funktionsparameter übergeben werden. Allerdings müssen alle Dimensionen, außer der höchsten Dimension, zum Compilierungszeitpunkt bekannt sein.

Ex740.cc

Wir betrachten als erstes **Beispiel** die Ausgabe eines (statischen oder dynamischen) 1D-Feldes, d.h., Vektors $\underline{x}$ der Länge n.

Struktogramm:




```
#include <iostream.h>
//---------------------------------------------------------------
//              Print elements of a vector (length n)
//
void PrintVec(const int n, const double x[])
{
 int i;

 cout << endl;
 for (i=0; i<n; i++)
  {
   cout << "  " << x[i];
  }
 cout << endl << endl;
 return;
}
//---------------------------------------------------------------
main()
{
 const int N=4;
       int n,i;
 double f[N] = {1,2,3,4};
 double *df;

 cin >> n;
 df = new double [n];  // Allocate dynamic array
//            Initialize df
 ...
 PrintVec(N,f);        // Print static array
 PrintVec(n,df);       // Print dynamic array
}
```

Als nächstes betrachten wir die Ausgabe eines statischen 2D-Feldes, d.h., einer Matrix mit `MCOL` Spalten und `NROW` Zeilen. Hier **muß** die Anzahl der Spalten als globale Konstante definiert werden, da ansonsten die nachfolgende Funktion nicht compiliert werden kann.

```
#include <iostream.h>

const int MCOL=3;               // global constant
//-------------------------------------------------------------
//      Print elements of a matrix
//         (nrow rows and fixed number MCOL of columns)
//
//              doesn't compile
//void PrintMat_fix(const int nrow, const double a[][])
//             doesn't help to fix that
//void PrintMat_fix(const int nrow, const int ncol,
//                                  const double a[][])
//
void PrintMat_fix(const int nrow, const double a[][MCOL])
{
 int i,j;

 cout << endl;
 for (i=0; i<nrow; i++)
  {
    cout << "row " << i << ":";
    for (j=0; j<MCOL; j++)
     {
       cout << "  " << a[i][j];
     }
    cout << endl;
  }
 cout << endl << endl;
 return;
}
//-------------------------------------------------------------
main()
{
 const int NROW=4;              // local constant
 double a[NROW][MCOL] = { ... };

 PrintMat_fix(NROW,a);        // print static matrix
}
```

Leider können wir die Funktion `PrintMat_fix` nur für statische 2D-Felder (Matrizen) anwenden, und dann auch nur für solche mit `NCOL=3` Spalten - schon eine Matrix `double aa[7][9]` kann mit dieser Funktion nicht mehr ausgegeben werden. Jedoch können wir das 2D-Feld als 1D-Feld der Länge `NROW*MCOL` auffassen und so die Funktion dahingehend verallgemeinern, daß beliebige statische 2D-Felder und als 2D-Felder interpretierbare dynamische 1D-Felder (wie in Version 2 auf Seite 52) übergeben werden können.

Ex740.cc

```
#include <iostream.h>
//-------------------------------------------------------------
// Print elements of a matrix
// (nrow rows and ncol columns)
//
void PrintMat(const int nrow, const int ncol, const double a[])
{
 int i,j;

 cout << endl;
 for (i=0; i<nrow; i++)
  {
    cout << "Row " << i << ":";
    for (j=0; j<ncol; j++)
     {
       cout << "  " << a[i*ncol+j] ;
     }
    cout << endl;
  }
 cout << endl << endl;
 return;
}
//-------------------------------------------------------------
main()
{
 const int NROW=7, MCOL=9;          // local constants
 double a[NROW][MCOL] = { ... };    // static matrix  a
 double *b;                         // dynamic matrix b
 int    nrow,ncol;

 cin >> nrow; cin >> ncol;          // read dimensions of b
 b = new double [NROW*MCOL];        // allocate b
// initialize matrix b
 ...
// output matrices
 PrintMat(NROW,MCOL,a[0]);          // Pointer on first row
 PrintMat(nrow,ncol,b);
}
```

Da die Funktion `PrintMat` ein 1D-Feld erwartet (also ein Zeiger), muß vom statischen 2D-Feld `a` ein Zeiger auf die erste Zeile der Matrix übergeben werden. Daher erscheint `a[0]` in der entsprechenden Rufzeile.

7.5 Deklarationen und Headerfiles, Bibliotheken

Normalerweise setzt sich der Quelltext eines Computerprogrammes aus (wesentlich) mehr als einem Quelltextfile zusammen. Damit Funktionen, Datenstrukturen (und globale Konstanten, Variablen) und Makros aus anderen Quelltextfiles

(*name.cc*) genutzt werden können, benutzt man Headerfiles (*name.hh*, *name.h*) welche die Deklarationen für das Quelltextfile *name.cc* beinhalten.

7.5.1 Beispiel: `printvec`

Wir wollen die in §7.4 programmierten Funktionen `PrintVec` und `PrintMat` in einem anderen Quelltext (d.h., Hauptprogramm) benutzen. Zunächst kopieren wir die Definitionen der beiden Funktionen (und alles andere, was zum Compilieren benötigt wird) in das neue File *printvec.cc*.

printvec.cc

```
//       printvec.cc
#include <iostream.h>
void PrintVec(const int n, const double x[])
{
 ...
}
void PrintMat(const int nrow, const int ncol, const double a[])
{
 ...
}
```

Das File *printvec.cc* wird nun compiliert (ohne es zu linken!)

```
LINUX> g++ -c printvec.cc
```

wodurch das Objektfile *printvec.o* erzeugt wird. Das Hauptprogramm in *Ex751-old.cc* benötigt nunmehr die Deklarationen der beiden Funktionen.

Ex751-old.cc

```
//       Ex751-old.cc
//       declarations of functions from printvec.cc
void PrintVec(const int n, const double x[]);
void PrintMat(const int nrow, const int ncol, const double a[]);
main()
{
 const int N=4,M=3;                    // local constant
                                       // static matrix a
 double a[N][M] = {4,-1,-0.5, -1,4,-1, -0.5,-1,4, 3,0,-1 },
        u[N] = {1,2,3,-2};

 PrintMat(N, M, a[0]);                 // print matrix
 PrintVec(N, u);                       // print vector
}
```

Das Compilieren des Hauptfiles

```
LINUX> g++ -c Ex751-old.cc
```

erzeugt das Objektfile *Ex751-old.o* welches mit dem anderen Objektfile zum fertigen Programm *a.out* gelinkt werden muß

```
LINUX> g++ Ex751-old.o printvec.o
```

Sämtliches compilieren und linken läßt sich auch in einer Kommandozeile ausdrücken

```
LINUX> g++ Ex751-old.cc printvec.cc
```

wobei manche Compiler im ersten Quelltextfile (hier *Ex751-old.cc*) das Hauptprogramm `main()` erwarten.

Die Deklarationen im Hauptprogramm für die Funktionen aus *printvec.cc* schreiben wir in das Headerfile *printvec.hh*

printvec.hh

```
//       printvec.hh
//       declarations of functions from printvec.cc

void PrintVec(const int n, const double x[]);
void PrintMat(const int nrow, const int ncol, const double a[]);
```

und wir ersetzen den Deklarationsteil im Hauptprogramm durch die von Präprozessoranweisung

`#include "printvec.hh"`

welche den Inhalt *printvec.hh* vor dem Compilieren von *Ex751.cc* automatisch einfügt.

Ex751.cc

```
//       Ex751.cc
#include "printvec.hh"
main()
{
 const int N=4,M=3;                    // local constant
                                       // static matrix a
 double a[N][M] = {4,-1,-0.5, -1,4,-1, -0.5,-1,4, 3,0,-1 },
        u[N] = {1,2,3,-2};

 PrintMat(N, M, a[0]);                 // print matrix
 PrintVec(N, u);                       // print vector
}
```

Die Anführungszeichen `"  "` um den Filenamen kennzeichnen, daß das Headerfile *printvec.hh* im gleichen Verzeichnis wie das Quelltextfile *Ex751.cc* zu finden ist.

Das Kommando

```
LINUX> g++ Ex751.cc printvec.cc
```

erzeugt wiederum das Programm *a.out*.

7.5.2 Beispiel: `student`

Wir können auch selbstdefinierte Datenstrukturen, z.B. die Datenstrukturen `Student`, `Student_Mult` aus §5.2 und `Student2` aus §6.4 und globale Konstanten in einem Headerfile *student.hh* speichern.

student.hh

```
//       student.hh
const int MAX_SKZ = 5;

struct Student
 { ... };
struct Student_Mult
 { ... };
struct Student2
 { ... };

void Copy_Student2(Student2& lhs, const Student2& rhs);
```

student.cc

Die neue Funktion `Copy_Student2` wird in *student.cc* definiert, wobei der Funktionskörper aus *Ex643-correct.cc* kopiert wurde.

```
//       student.cc
#include <strings.h>
#include "student.hh"

void Copy_Student2(Student2& lhs, const Student2& rhs)
{
 lhs = rhs;
//               Allocate memory and copy data
 lhs.pname = new char[strlen(rhs.pname)+1];
 strcpy(lhs.pname,rhs.pname);
 lhs.pvorname = new char[strlen(rhs.pvorname)+1];
 strcpy(lhs.pvorname,rhs.pvorname);

 return;
}
```

Da die Struktur `Student2` verwendet wird, muß auch das Headerfile *student.hh* in *student.cc* eingebunden werden. Die neue Funktion `Copy_Student2` kann nunmehr im Hauptprogramm *Ex752.cc* zum Kopieren einer Struktur benutzt werden. Das Hauptprogramm benötigt dafür natürlich wieder das Headerfile *student.hh*.

Ex752.cc

Das Kommando

```
LINUX> g++ Ex752.cc student.cc
```

erzeugt schlußendlich das Programm *a.out*.

7.5.3 Eine einfache Bibliothek am Beispiel `student`

Um sich das wiederholte compilieren zusätzlicher Quelltextfiles und die damit verbundenen u.U. langen Listen von Objektfiles beim Linken zu ersparen, verwendet man Bibliotheken. Gleichzeitig haben Bibliotheken den Vorteil, daß man seine compilierten Funktionen (zusammen mit den Headerfiles) anderen in kompakter Form zur Verfügung stellen kann, ohne daß man seine Programmiergeheimnisse (geistiges Eigentum) verraten muß. Dies sei an Hand des (sehr einfachen) Beispiels aus §7.5.2 demonstriert.

- Erzeugen des Objektfiles *student.o* (compilieren)
  ```
  LINUX> g++ -c student.cc
  ```

- Erzeugen/Aktualisieren der Bibliothek *libstud.a* (archivieren) aus/mit dem Objektfile *student.o*. Der Bibliotheksbezeichner stud ist frei wählbar.
  ```
  LINUX> ar r libstud.a student.o
  ```
  Die Archivierungsoptionen (hier, nur `r`) können mit dem verwendeten Compiler variieren.

- Compilieren des Hauptprogrammes und linken mit der Bibliothek aus dem aktuellen Verzeichnis
  ```
  LINUX> g++ Ex752.cc -L. -lstud
  ```

Ex752.cc

Die folgenden Schritte sind notwendig, um das Programm ohne Verwendung einer Bibliothek zu übersetzen und zu linken.

$$\left.\begin{array}{ll} \textit{student.cc} \xrightarrow{\texttt{g++ -c student.cc}} \textit{student.o} \\ \textit{Ex752.cc} \xrightarrow{\texttt{g++ -c Ex752.cc}} \textit{Ex752.o} \end{array}\right\} \xrightarrow{\texttt{g++ Ex752.o student.o}} \textit{a.out}$$

Abkürzend ist auch möglich:

$$\textit{Ex752.cc, student.cc} \xrightarrow{\texttt{g++ Ex752.cc student.cc}} \textit{a.out}$$

Bei Verwendung der Biobliothek *lib***stud**.*a* sieht der Ablauf folgendermaßen aus

$$\left.\begin{array}{l} \textit{student.cc} \xrightarrow[\texttt{student.cc}]{\texttt{g++ -c}} \textit{student.o} \xrightarrow[\texttt{student.o}]{\texttt{ar r lib}\mathbf{stud}\texttt{.a}} \textit{lib}\mathbf{stud}\textit{.a} \\ \textit{Ex752.cc} \xrightarrow[\texttt{Ex752.cc}]{\texttt{g++ -c}} \textit{Ex752.o} \end{array}\right\} \xrightarrow[\texttt{-L. -l}\mathbf{stud}]{\texttt{g++ Ex752.o}} \textit{a.out}$$

was bei bereits vorhandener Bibliothek wiederum abgekürzt werden kann:

$$\textit{Ex752.cc, lib}\mathbf{stud}\textit{.a} \xrightarrow{\texttt{g++ Ex752.cc -L. -l}\mathbf{stud}} \textit{a.out}$$

7.6 Das Hauptprogramm

Die bislang benutzte Syntax für das Hauptprogramm

```
main()
{
 ...
}
```

wird vom Compiler stets als

```
int main()
{
 ...
 return 0;
}
```

verstanden, da bei Funktionen ohne Typspezifikation der Typ `int` als Standard verwendet wird mit dem Standardrückgabewert `0`. Ein Rückgabewert `0` bedeutet, daß das Hauptprogramm fehlerfrei abgearbeitet wurde.

Die Programmabarbeitung kann jederzeit, auch in Funktionen, mit der Anweisung `exit(<int_value>);` abgebrochen werden. Der Wert `<int_value>` ist dann der Rückgabewert des Programmes und kann zur Fehlerdiagnose herangezogen werden.

```
#include <iostream.h>
#include <stdlib.h>          // needed to declare exit()

void spass(const int);       // Declaration of spass()

int main()
{
 int n;

 cin  >> n;
 if (n<0)   exit(-10);       // choose an error code

 spass(n);                   // Call spass()

 return 0;                   // default return value is 0
}

void spass(const int n)      // Definition of spass()
{ ... }
```

Das obige Programm bricht bei $n < 0$ die Programmausführung sofort ab und liefert den Fehlercode 10. Die `exit` Anweisung kann auch in `spass()` verwendet werden.

Wie bei anderen Funktionen kann auch das Hauptprogramm mit Parametern aufgerufen werden, allerdings ist in

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

die Parameterliste (genauer, die Typen der Parameter) vorgeschrieben, wobei

- `argv[0]` den Programmnamen und
- `argv[1]` ... `argv[argc-1]` die Argumente beim Programmaufruf als Zeichenketten übergeben.

- Es gilt stets `argc`≥ 1, da der Programmname immer übergeben wird.

```
//      for a real C++ solution, see Stroustrup, p.126 (6.1.7)
#include <iostream.h>
#include <stdlib.h>             // needed for atoi, exit

void spass(const int);          // Declaration of spass()

int main(int argc, char* argv[])
{
 int n;

 cout << "This is code " << argv[0] << endl;

 if (argc > 1) // at least one argument
  {
   n = atoi(argv[1]); // atoi : ACSII to Integer
  }
 else // standard input
  {
   cout << " Eingabe n : ";
   cin  >> n;
   cout << endl;
  }

 spass(n); // Call spass()

 return 0; // default return value is 0
}


void spass(const int n)
{
 if (n<0) exit(-10); // choose an error code

 cout << "Jetzt schlaegt's aber " << n << endl;
 return;
}
```

Ex760.cc

Die Funktion `atoi(char *)` (= ASCII to int) wandelt die übergebene Zeichenkette in eine Integerzahl um und wird in *stdlib.h* deklariert. Mittels der analogen Funktion `atod(char *)` läßt sich eine `float`-Zahl als Parameter übergeben. Nach dem Compilieren und Linken kann das Programm `a.out` mittels

```
LINUX> a.out
```

bzw.

```
LINUX> a.out 5
```

gestartet werden. Im ersteren Fall wird der Wert von `n` von der Tastatur eingelesen, im zweiten Fall wird der Wert `5` aus der Kommandozeile übernommen und `n` zugewiesen. Eine elegante, und echte C++-Lösung, bzgl. der Übergabe

von Kommandozeilenparametern kann in [Str00, pp.126] gefunden werden.

7.7 Rekursive Funktionen

Funktionen können in C/C++ rekursiv aufgerufen werden.

Beispiel: Die Potenz x^k mit $x \in \mathbb{R}$, $k \in \mathbb{N}$ kann auch als $x^k = \begin{cases} x \cdot x^{k-1} & k > 0 \\ 1 & k = 0 \end{cases}$ realisiert werden.

Ex770.cc

```
// definition of function power
double power(const double x, const int k)
{
 double y;

 if ( k == 0)
  {
   y = 1.0;                // Stops recursion
  }
 else
  {
   y = x * power(x,k-1);   // recursive call
  }

 return y;                 // return value of function power
}
```

7.8 Ein größeres Beispiel: Bisektion

Im Beispiel auf Seite 39 ging es darum, die Nullstelle von $f(x) := \sin(x) - x/2$ im Intervall (a,b), mit $a = 0$ und $b = 1$ zu bestimmen. Unter der Voraussetzung $f(a) > 0 > f(b)$ kann dieses Problem (für stetige Funktionen) mittels Bisektion gelöst werden. Der Bisektionsalgorithmus besteht für jedes Intervall $[a, b]$ im wesentlichen aus den Schritten

(i) $c := (a + b)/2$

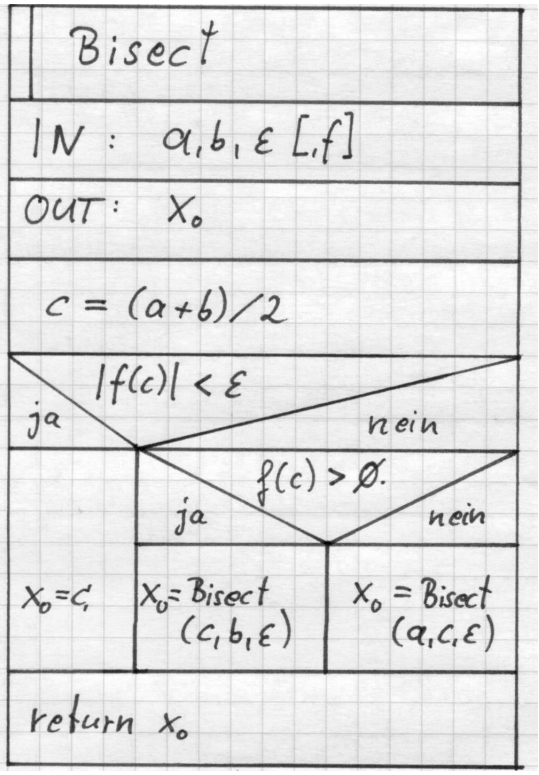
(ii) Ist $|f(c)|$ nah genug an 0 ?

(iii) In welcher Intervallhälfte muß ich weitersuchen ?

Dies ist eine klassische Rekursion, wobei Punkt (iii) die nächste Rekursion einleitet und Punkt (ii) den Abbruch der Rekursion garantieren soll. Formal können wir dies so ausdrücken:

$$x_0 := \text{Bisect}(a, b, \varepsilon) := \begin{cases} c := (a + b)/2 & \text{falls } |f(c)| < \varepsilon \\ \text{Bisect}(c, b, \varepsilon) & \text{sonst, falls } f(c) > 0 \\ \text{Bisect}(b, c, \varepsilon) & \text{sonst, falls } f(c) < 0 \end{cases}$$

Struktogramm:



Dies ergibt die Funktionsdefinition für `Bisect()` welche mit
`x0 = Bisect(a,b,1e-6);`
aufgerufen wird und zur Version 1 des Bisektionsprogrammes führt. Bisect1.cc

```
double Bisect1(const double a, const double b, const double eps)
{
 double x0, fc, c = (a+b)/2;

 fc = sin(c) - 0.5*c;
 if ( fabs(fc) < eps )
  {
   x0 = c;                        // end of recursion
  }
 else if (  fc > 0.0 )
  {
   x0 = Bisect1(c,b,eps);          // search in right intervall
  }
 else                              // i.e., fc < 0.0
  {
   x0 = Bisect1(a,c,eps);          // search in left intervall
  }

 return x0;                        // return the solution
}
```

Um das Programm etwas flexibler zu gestalten, werden wir die fix in `Bisect1()` einprogrammierte Funktion $f(x)$ durch die globale Funktion

```
double f(const double x)         // declaration and
  { return  sin(x) - 0.5*x ; }   //   definition of function f(x)
```

ersetzen. Gleichzeitig könnten wir den Funktionsparameter `eps` durch eine globale Konstante `EPS` ersetzen, sodaß sich Version 2 ergibt.

Bisect2.cc

Die Flexibilität der Bisektionsfunktion läßt sich weiter erhöhen indem wir die auszuwertende Funktion $f(x)$ als Variable in der Parameterliste übergeben. Eine Funktion als Parameter/Argument wird immer als Zeiger übergeben, d.h., eine Funktion als Argument muß wie die Deklaration für `f6` auf Seite 76 aufgebaut sein. Konkret heißt dies:

`double (*func)(double)` ist ein Zeiger auf eine Funktion `func` mit einer `double`-Variablen als Argument und `double` als Typ des Rückkehrwertes.

Dies erlaubt uns die Funktionsdeklaration und -definition von `Bisect3()`

```
               // declaration of Bisect3
double Bisect3(double (*func)(double), const double a,
                const double b, const double eps=1e-6);
...
main()
{...}
               // definition of Bisect3
double Bisect3(double (*func)(double), const double a,
                     const double b, const double eps)
{
 double x0, fc, c = (a+b)/2;

 fc = func(c);     // calculate value of parameter function
 if ( fabs(fc) < eps )
  {
   x0 = c;                       // end of recursion
  }
 else if (  fc > 0.0 )
  {
   x0 = Bisect3(func,c,b,eps);  // search in right intervall
  }
 else                            // i.e., fc < 0.0
  {
   x0 = Bisect3(func,a,c,eps);  // search in left intervall
  }

 return x0;                      // return the solution
}
```

Das vierte Argument (`eps`) in der Parameterliste von `Bisect3()` ist ein **optionales Argument**, welches beim Funktionsaufruf nicht übergeben werden muß. In diesem Fall wird diesem optionalen Argument sein, in der Funktionsdeklaration festgelegter, Standardwert automatisch zugewiesen. In unserem Falle würde also der Aufruf im Hauptprogramm

`x0 = Bisect3(f,a,b,1e-12)`

die Rekursion bei $|f(c)| < \varepsilon := 10^{-12}$ abbrechen, während

`x0 = Bisect3(f,a,b)`

schon bei $|f(c)| < \varepsilon := 10^{-6}$ stoppt. Bisect3.cc

Wir könnten jetzt eine weitere Funktion

```
                          // declaration and
double g(const double x)
                          //   definition of function g(x)
  { return -(x-1.234567)*(x+0.987654) ; }
```

deklarieren und definieren, und den Bisektionsalgorithmus in Version 3. mit ihr aufrufen: Bisect3.cc

`x0 = Bisect3(g,a,b,1e-12)`

Bemerkung: Da wir unsere als Argument in `Bisect3` übergebene Funktion `func` ein reiner INPUT-Parameter ist, sollten wir sie noch mit `const` kennzeichnen. Allerdings ist die richtige Kennzeichnung des ersten Arguments in `Bisect3`

```
double Bisect3(double (* const func)(double), const double a,
               const double b, const double eps=1e-6);
```

am Anfang etwas verwirrend.

Unser Programm arbeitet zufriedenstellend für $f(x) = \sin(x) - x/2$ und liefert für die Eingabeparameter $a = 1$ und $b = 2$ die richtige Lösung $x_0 = 1.89549$, desgleichen für $a = 0$ und $b = 2$ allerdings wird hier bereits die (triviale) Lösung $x_0 = 0$ nicht gefunden, da $a = 0$ eingegeben wurde. Bei den Eingaben $a = 0$, $b = 1$ bzw. $a = -1$, $b = 0.1$ ($x_0 := 0 \in [a, b]$) bricht das Programm nach einiger Zeit mit *Segmentation fault* ab, da die Rekursion nicht abbricht und irgendwann der für Funktionsaufrufe reservierte Speicher (*Stack*) nicht mehr ausreicht.

Können wir unser Programm so absichern, daß z.B. die vorhandene Nullstelle $x_0 = 0$ sowohl in $[0, 1]$ als in $[-1, 0.1]$ gefunden wird? Welche Fälle können bzgl. der Funktionswerte $f(a)$ und $f(b)$ auftreten (vorläufige Annahme: $a < b$)?

(i) $f(a) > 0 > f(b)$ (d.h., $f(a) > 0$ und $f(b) < 0$), z.B., $a = 1$, $b = 2$
$\Longrightarrow$ Standardfall in `Bisect3()`.

(ii) $f(a) > 0$ und $f(b) > 0$, z.B., $a = 0.5$, $b = 1.5$ bzw.
$f(a) < 0$ und $f(b) < 0$, z.B., $a = -1$, $b = 0.5$
evtl. keine Nullstelle $\Longrightarrow$ Abbruch.
(Es können Nullstellen im Intervall vorhanden sein, welche wir aber mit der Bisektionsmethode nicht finden können!)

(iii) $f(a) = 0$ oder $f(b) = 0$, besser $|f(a)| < \varepsilon$ etc.
$\Longrightarrow$ a oder b sind die Nullstelle,
oder $\Longrightarrow$ sowohl a als auch b sind eine Nullstelle.

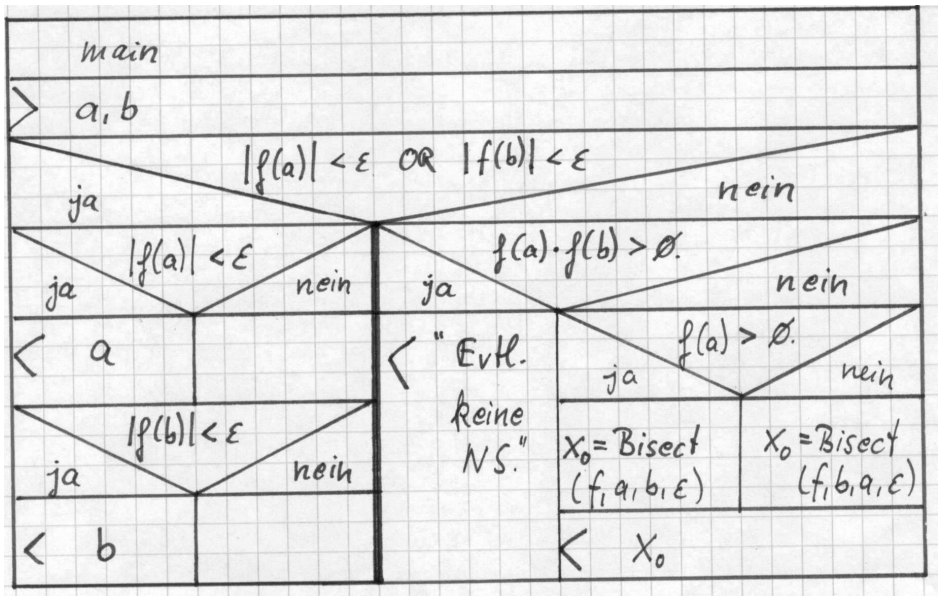
(iv) $f(a) < 0 < f(b)$, z.B. $a = -1$, $b = 0.1$
Vertausche a und b $\Longrightarrow$ Fall (i).

(v) $a = b$ $\Longrightarrow$ in (ii) und (iii) enthalten.
$b < a$ $\Longrightarrow$ führt auf (i) oder (iv).

Diese Fallunterscheidung führt uns zum folgenden Struktogramm und zur *Version 4*.

Bisect4.cc

Struktogramm:



Als krönenden Abschluß definieren wir uns im Programm weitere Funktionen $h(x) = 3 - e^x$, $t(x) = 1 - x^2$, fragen den Nutzer welche math. Funktion für die Nullstellensuche benutzt werden soll und berechnen die Nullstelle(n) im gegebenen Intervall. Diese Auswahl kann leicht mit einer `switch`-Anweisung realisiert werden und führt zu Version 5 des Programmes. Bisect5.cc

Bemerkung: Die drei Funktionen `Bisect`[1-3]() unterscheiden sich in ihren Parameterlisten. Deshalb können alle drei Funktionen unter dem Namen `Bisect()` verwendet werden, da sich ihre Signaturen unterscheiden und somit der Compiler genau weiß, welche Funktion `Bisect()` verwendet werden soll. Bisect6.cc

Kapitel 8

Der Datentyp Klasse (class)

Der Datentyp `Student2` aus §6.4 enthält dynamische Datenstrukturen wodurch Initialisierung und Kopieren der entsprechenden Variablen jedesmal speziell implementiert werden müssen (siehe auch §7.5.2). Ein großer Vorteil, neben vielen anderen, des Konzeptes der Klassen in C++ besteht darin, daß auch Datenstrukturen mit dynamischen Komponenten im Hauptprogramm wie einfache Datentypen gehandhabt werden können. Dies erfordert natürlich einige Vorarbeiten.

Eine Klasse (class) ist ein Datentyp samt zugehörigen Methoden (≈ Funktionen) und wird ähnlich einer Struktur angelegt und kann analog verwendet werden. Die Methoden einer Klasse haben stets Zugriff zu den Daten dieser Klasse.

Ausgehend von der Struktur `Student2` leiten wir eine Klasse `Studenten` ab. Hierbei speichern wir alle Deklarationen der Klasse im Headerfile *studenten.hh* und alle Definitionen in *studenten.cc*. Eine bzgl. der Pointerinitialisierung modifizierte Variante ist in den Files *studs.hh* und *studs.cc* zu finden.

studenten.hh

studenten.cc

studs.hh

studs.cc

8.1 Daten und Methoden der Klassendeklaration

Hier die Deklaration der Klasse `Studenten` mit den unbedingt notwendigen Methoden.

```
//       studenten.hh
class Studenten
{
// Data in Studenten
public:
  long int matrikel;
       int skz;
      char *pname, *pvorname;

//                       Methods in Studenten
public:

//                               Default constructor (no argument)
 Studenten();

//                                Constructor with 4 arguments
 Studenten(const char vorname[], const char name[],
       const long int  mat_nr = 0,   const int  skz_nr = 0);

//                               Copy constructor
 Studenten(const Studenten& orig);

//                               Destructor
 ~Studenten();

//                               Assignment operator
 Studenten& operator=(const Studenten & orig);

//                               Further methods
 ...
};
```

Die aufgeführten Methoden betrachten wir in der angegebenen Reihenfolge.

8.2 Die Konstruktoren

Konstruktoren dienen der Erstinitialisierung von Daten einer Klasse. Im allgemeinen sollte eine Klasse mindestens die folgenden Konstruktoren besitzen.

- Definition des **Standardkonstruktors** (ohne Argumente)

```
//        Standardkonstruktor
Studenten :: Studenten()
{
 matrikel = skz = 0;
 pname = pvorname = 0;
}
```

In der Konstruktion `Studenten ::` kennzeichnet der Scope-operator `::` die Zugehörigkeit der Methode `Studenten()` zur Klasse `Studenten` und ist Bestandteil der Signatur dieser Methode (Funktion).
Der Standardkonstruktor wird im Hauptprogramm mit
`Studenten robbi;`
automatisch aufgerufen, sodaß alle Daten von `robbi` mit 0 bzw. dem Nullpointer initialisiert werden.

- Definition eines **Parameterkonstruktors**

```
//        Parameterkonstruktor
Studenten :: Studenten
       (const char vorname[], const char name[],
        const long int  mat_nr,    const int  skz_nr)
{
 matrikel = mat_nr;
 skz      = skz_nr;

 pname = new char[strlen(name)+1];
 strcpy(pname,name);
 pvorname = new char[strlen(vorname)+1];
 strcpy(pvorname,vorname);
}
```

Mit dem Parameterkonstruktor kann eine Variable des Typs `Studenten` deklariert, definiert und gleichzeitig initialisiert werden.
`Studenten arni("Arni","Schwarz",812,7981579);`
Genauso wäre möglich:

```
    char tmp1[20], tmp2[20];
long int tm;
     int ts;

cin >> tmp1 >> tmp2 >> tm >> ts;

Studenten arni(tmp1,tmp2,tm,ts);
```

Parameterkonstruktoren können optionale Parameter enthalten, deren Standardwerte bereits in der Deklaration (Seite 94) festgelegt werden. Somit wäre auch eine Variablendefinition des Typs `Studenten` über
`Studenten arni("Arni","Schwarz");`
gültig.

- Definition des **Copykonstruktors**

```
//        Copykonstruktor
Studenten :: Studenten(const Studenten& orig)
{
 matrikel = orig.matrikel;
 skz      = orig.skz;

 if (orig.pname !=0 )
  {
    pname = new char[strlen(orig.pname)+1];
    strcpy(pname,orig.pname);
  }
 else
  {
    pname = 0;
  }
 if (orig.pvorname != 0)
  {
   pvorname = new char[strlen(orig.pvorname)+1];
   strcpy(pvorname,orig.pvorname);
  }
 else
  {
   pvorname = 0;
  }
}
```

Der Copykonstruktor erlaubt eine Definition anhand einer anderen Variablen derselben Klasse, wie in

```
Studenten mike(arni);
```

8.3 Der Destruktor

Jede Klasse besitzt genau einen Destruktor, welcher beim Verlassen des Gültigkeitsbereichs einer Variablen (Blockende, Funktionsende) automatisch aufgerufen wird. Die Hauptaufgabe des Destruktors besteht meist im Freigeben von dynamischem Speicherplatz der Klasse.

```
//                        Destruktor
Studenten :: ~Studenten()
{
 if (pvorname != 0) delete [] pvorname;
 if (   pname != 0) delete [] pname;
}
```

8.4 Der Zuweisungsoperator

Konstruktoren greifen stets auf nicht initialisierte Daten zu. Um der bereits (mit Standardwerten) initialisierten Variablen `robbi` die Daten von `arni` zuzuweisen, d.h., `robbi = arni;` , muß noch ein Zuweisungsoperator definiert werden,

welcher sich im wesentlichen aus den Funktionskörpern von Destruktor und Copykonstruktor zusammensetzt. Jedoch ist hier vorher zu überpüfen, ob die rechte Seite der Zuweisung (übergebene Variable) nicht identisch zur linken Seite der Zuweisung (`this`) ist.

```
//                        Zuweisungsoperator
Studenten& Studenten :: operator=(const Studenten & orig)
{
 if ( &orig != this )                          // test
  {
    if (pvorname != 0)  delete [] pvorname;
    if (pname != 0)     delete [] pname;

    matrikel = orig.matrikel;
    skz      = orig.skz;

    if (orig.pname !=0 )
     {
       pname = new char[strlen(orig.pname)+1];
       strcpy(pname,orig.pname);
     }
    else
     {
       pname = 0;
     }
    if (orig.pvorname != 0)
     {
      pvorname = new char[strlen(orig.pvorname)+1];
      strcpy(pvorname,orig.pvorname);
     }
    else
     {
      pvorname = 0;
     }
  }

 return *this;
}
```

Weiterführend sei hier auf den Stichpunkt *Überladen von Operatoren* in der Literatur [SK98, §16], [Str00, $ 11] verwiesen.

8.5 Der Printoperator

Ein nicht unbedingt notwendiger, aber recht nützlicher Operator ist der Printoperator für eine Klasse, welcher die Datenausgabe mittels

```
cout << robbi <<endl;
```

ermöglichen soll.

Die Deklaration in *studenten.hh*

```
#include <iostream.h>
//                        output operator
friend ostream & operator<<(ostream  & s,
                            const Studenten & orig);
```

erlaubt, dank des Bezeichners `friend`, die Nutzung der Klasse `Studenten` zur Definition einer neuen Methode der Klasse `ostream` (in *iostream.h* deklariert). Die Definition ist dann:

```
ostream & operator<<(ostream  & s, const Studenten & orig)
{
 return s << orig.pvorname << " "   << orig.pname << " , "
          << orig.matrikel << " , " << orig.skz;
}
```

Nunmehr können wir das Beispiel *Ex643-correct.cc* (bzw. *Ex752.cc*) aus §6.4 wesentlich einfacher schreiben und erweitern.

Ex851.cc

```
#include <iostream.h>
#include "studenten.hh"

int main()
{
 Studenten robbi;               // Default constructor

 {              // start block
                                // Constructor with args
   Studenten arni("Arni", "Schwarz", 812, 7938592);

   robbi = arni;                // Assignment operator

  }             // end block    // Destructor for arni

 Studenten mike(robbi);         // Copy constructor

 cout << "mike : " << mike << endl;

 cout << endl;
//      Data in Studenten are public therefore:
 cout << "Access to public data : ";
 cout << "mike.pvorname = " << mike.pvorname << endl;

 return 0;
}                               // Destructor for robbi, mike
```

Die Kommandozeile

```
LINUX> g++ Ex851.cc studenten.cc
```

erzeugt das ausführbare Programm.

8.6 Datenkapselung

Die Daten in `Studenten` wurden als `public` klassifiziert, d.h., jeder kann auf diese Daten zugreifen, wie mittels `mike.pvorname` . Um diese Daten vor unerwünschtem Zugriff zu schützen und um das Datenlayout evtl. nachträglich ändern zu können, kapselt man die Daten so in die Klasse ein, daß sie nur noch über Zugriffsmethoden erreichbar sind. Die diesbezügliche Klassifikation wird durch das Schlüsselwort `private` angegeben. Damit ändert sich der Deklarationsteil der Klasse `Studenten` in

```
//      studenten2.hh
#include <iostream.h>

class Studenten
{
// Data in Studenten are private now!!
private:
  long int matrikel;
       int skz;
      char *pname, *pvorname;

//                       Methods in Studenten
public:
//                       Constructors, Destructor, Access operator
 ...
// Output operator
 friend ostream & operator<<(ostream  & s, const Studenten & orig);

//              Methods to access the private data
//                       Methods for data manipulation in Studenten
 void SetVorname(const char vorname[]);
 void SetName(const char name[]);
 void SetMatrikel(const long int  mat_nr);
 void SetSKZ(const int  skz_nr);

//                 Methods that don't manipulate data in Studenten
 const long int& GetMatrikel() const;
 const int& GetSKZ() const;
 const char* GetVorname() const;
 const char* GetName() const;
};
```

studenten2.hh

In obigen Methoden sind zwei `const` Deklarationen zu finden. Ein `const` am Ende der Deklarationszeile zeigt an, daß die Daten in `Studenten` von der entsprechenden Methode, z.B., `GetSKZ` nicht verändert werden. Das `const` am Zeilenanfang gehört zum Ergebnistyp und zeigt an, daß die Daten auf welche mit der Referenz `int&` verwiesen wird nicht verändert werden dürfen. Damit wird gesichert, daß die Daten auch nicht unabsichtlich über Zeiger oder Referenzen manipuliert werden können.

Die Zugriffsmethoden werden wie folgt definiert:

studenten2.cc

```
//                    studenten2.cc
#include "studenten2.hh"
...

void Studenten :: SetVorname(const char vorname[])
{
 if (pvorname != 0) delete [] pvorname;
 pvorname = new char[strlen(vorname)+1];
 strcpy(pvorname,vorname);
 return;
}

void Studenten :: SetSKZ(const int  skz_nr)
{
 skz = skz_nr;
 return;
}
...
const char* Studenten :: GetVorname() const
{
 return pvorname;
}

const int& Studenten :: GetSKZ() const
{
 return skz;
}
```

Diese neuen Zugriffsmethoden können wie folgt benutzt werden:

```
//      Ex861.cc
#include <iostream.h>
#include <strings.h>
#include "studenten2.hh"

int main()
{
 Studenten mike("Arni", "Schwarz", 812, 7938592);

 cout << "mike : " << mike << endl;
 cout << endl;
//      Data in Studenten are private therefore --> inaccessible:
// cout << "Access to public data : ";
// cout << "mike.pvorname = " << mike.pvorname << endl;

//      Data in Studenten are private therefore :
 cout << "Access to private data via methods: " << endl;
 cout << "mike.pvorname = " << mike.GetVorname() << endl;

// mike.GetVorname()[3] = 'k'; // not allowed because of 'const'
// char *pp = mike.GetVorname();// not allowed because of 'const'
 char tmp[40];
 strcpy(tmp,mike.GetVorname()); // allowed

 return 0;
}
```

Ex861.cc

Einige Zugriffsfunktionen, z.B., `SetSKZ` und `GetSKZ` sind so kurz, daß sich ein Funktionsaufruf wegen des Aufwandes der Parameterübergabe eigentlich nicht lohnt. In diesem Falle werden Deklaration und Definition einer Methode miteinander im Headerfile verknüpft, die Methode/Funktion wird **inline** definiert. Diese inline-Zeilen ersetzen jedesmal den Funktionsaufruf.

```
// studenten3.hh

#include <iostream.h>

class Studenten
{
 ...
 void SetSKZ(const int  skz_nr)
  { skz = skz_nr; };           // inline function
 ...
 const int& GetSKZ() const
  { return skz; };             // inline function
};
```

studenten3.hh

Kapitel 9

File Input und Output

Die zur Ein-/Ausgabe verwendeten Objekte `cin` und `cout` sind (in *iostream.h*) vordefinierte Variablen vom Klassentyp `stream`. Um von Files zu lesen bzw. auf Files zu schreiben werden nun neue Streamvariablen angelegt und zwar vom Typ `ifstream` für die Eingabe und vom Typ `ofstream` für die Ausgabe. Der Filename wird beim Anlegen der Variablen übergeben (C++ Konstruktor).

9.1 Kopieren von Files

Ex911.cc

Das folgende Programm kopiert ein Inputfile auf ein Outputfile, allerdings ohne Leerzeichen, Tabulatoren, Zeilenumbrüche.

```
//                      Ex911.cc
#include <iostream.h>
#include <fstream.h>

int main()
{
 char infilename[100], outfilename[100];
 char str[100];

 cout << " Input file: "; cin >> infilename;
 cout << "Output file: "; cin >> outfilename;

 ifstream  infile(infilename);
 ofstream outfile(outfilename);

 while (infile.good())
 {
   infile >> str;
  outfile << str;
 }

 return 0;
}
```

Ex912.cc

Will man dagegen das File identisch kopieren, so muß auch zeichenweise ein- und ausgelesen werden. Hierzu werden die Methoden `get` und `put` aus den entsprechenden Streamklassen verwendet.

```
//                          Ex912.cc
#include <iostream.h>
#include <fstream.h>

int main()
{
 char infilename[100], outfilename[100];
 char ch;

 cout << " Input file: "; cin >> infilename;
 cout << "Output file: "; cin >> outfilename;

 ifstream  infile(infilename);
 ofstream outfile(outfilename);

 while (infile.good())
 {
   infile.get(ch);
  outfile.put(ch);
 }
 return 0;
}
```

9.2 Dateneingabe und -ausgabe via File

Die Dateneingabe und -ausgabe via File und Terminal kann gemischt benutzt werden.

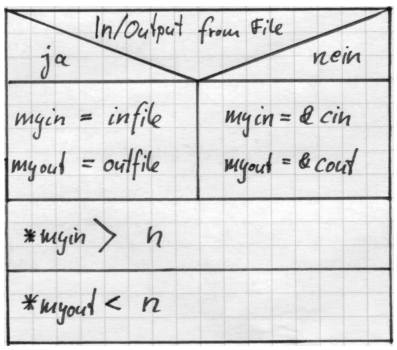
FileIO_a.cc

```
//              FileIO_a.cc
#include <iostream.h>
#include <fstream.h>       // needed for ifstream and ofstream

int main()
{
 int n_t, n_f;
// input file
 ifstream infile("in.txt");

 cout << "Input from terminal: ";
 cin  >> n_t;

 cout << "Input from file " << endl;
 infile >> n_f;
//                       check it
 cout << endl;
 cout << "Input  from terminal was " << n_t << endl;
 cout << "Output from     file was " << n_f << endl;
 cout << endl;
//                       output file
 ofstream outfile("out.txt");

 cout << "This is an output to the terminal" << endl;

 outfile << "This is an output to the file" << endl;

 return 0;
}
```

9.3 Umschalten der Ein-/Ausgabe

Manchmal ist ein problemabhängiges Umschalten zwischen File-IO und Terminal-IO wünschenswert oder nötig. Leider muß in diesem Falle mit Zeigern auf die Typen `istream` und `ostream` gearbeitet werden.

FileIO_b.cc



```
//          FileIO_b.cc
#include <iostream.h>
#include <fstream.h>

int main()
{
 int  n, tf;
 bool bf;
//                       variables for IO streams
 istream  *myin;
 ostream  *myout;
//                       input file
 istream* infile  = new ifstream("in.txt");
//                       output file
 ostream* outfile = new ofstream("out.txt");

//      Still standard IO
//      Decide whether terminal-IO or file-IO should be used
 cout << "Input from terminal/file - Press 0/1 : ";
 cin  >> tf;
 bf = (tf==1);

 if (bf)
   {                   // Remaining IO via file
     myin  = infile;
     myout = outfile;
   }
 else
   {                   // Remaining IO via terminal
     myin  = &cin;
     myout = &cout;
   }

 (*myout) << "Input: ";
 (*myin)  >> n;
//                        check
 (*myout) << endl;
 (*myout) << "Input was " << n << endl;
 (*myout) << endl;

 (*myout) << "This is an additional output" << endl;

 delete  outfile;       // don't forget it
 delete  infile;

 return 0;
}
```

Eine sehr komfortable Möglichkeit des Umschaltens der Ein-/Ausgabe mittels Kommandozeilenparameter ist in den Beispielen zu finden.

FileIO_c.cc

FileIO_d.cc

Kapitel 10

Ausgabeformatierung

Die Ausgabe über Streams (<<) kann verschiedenst formatiert werden. Eine kleine Auswahl von Formatierungen sei hier angegeben, mehr dazu in der Literatur.

Wir benutzen die Variablen

```
double da = 1.0/3.0,
       db = 21./2,
       dc = 1234.56789;
```

Format.cc

- Standardausgabe:
  ```
  cout << da << endl << db << endl << dc << endl << endl;
  ```
- Mehr gültige Ziffern (hier 12) in der Ausgabe:
  ```
  cout.precision(12);
  cout << ...
  ```
- Fixe Anzahl (hier 6) von Nachkommastellen:
  ```
  cout.precision(6);
  cout.setf(ios::fixed, ios::floatfield);
  cout << ...
  ```
- Ausgabe mit Exponent:
  ```
  cout.setf(ios::scientific, ios::floatfield);
  cout << ...
  ```
- Rücksetzen auf Standardausgabe:
  ```
  cout.setf(0, ios::floatfield);
  cout << ...
  ```
- Ausrichtung (rechtsbündig) und Platzhalter (16 Zeichen):
  ```
  cout.setf(ios::right, ios::adjustfield);
  cout.width(16);
  cout << da << endl;
  cout.width(16);
  cout << db << endl;
  cout.width(16);
  cout << dc << endl;
  ```

Eine allgemeine Lösung mittels Standardmanipulatoren ist in [Str00, § 1.4.6.2, pp.679] zu finden.

- Hexadezimalausgabe von Integerzahlen:
```
cout.setf(ios::hex, ios::basefield);
cout << "127 = " << 127 << endl;
```

Kapitel 11

Tips und Tricks

11.1 Präprozessorbefehle

Wir kennen bereits die Präprozessoranweisung

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

welche vor dem eigentlichen Compilieren den Inhalt des Files *math.h* an der entsprechenden Stelle im Quellfile einfügt. Analog können bestimmte Teile des Quelltextes beim Compilieren eingebunden oder ignoriert werden, je nach Abhängigkeit des Tests (analog einer Alternative wie in §4.3) welcher mit einer Präprozessorvariablen durchgeführt wird.

preproc.cc

Variablen des Präprozessors werden mittels

```
#define MY_DEBUG
```

definiert und wir können auch testen, ob sie definiert sind:

```
#ifdef MY_DEBUG
  cout << "Im Debug-Modus" << endl;
#endif
```

Analog kann mit

```
#ifndef MY_DEBUG
#define MY_DEBUG
#endif
```

zunächst getestet werden, ob die Variable `MY_DEBUG` bereits definiert wurde. Falls nicht, dann wird sie eben jetzt definiert. Diese Technik wird häufig benutzt um zu verhindern, daß die Deklarationen eines Headerfiles mehrfach in denselben Quelltext eingebunden werden.

studenten4.hh

```
//      studenten4.hh
#ifndef FILE_STUDENTEN
#define FILE_STUDENTEN
//      Deklarationen des Headefiles
...
#endif
```

Einer Präprozessorvariablen kann auch ein Wert zugewiesen werden

```
#define SEE_PI 5
```

welcher anschließend in Präprozessortests (oder im Programm als Konstante) benutzt werden kann:

```
#if (SEE_PI==5)
  cout << " PI = " << M_PI << endl;
#else
// leer oder Anweisungen
#endif
```

Eine häufige Anwendung besteht in der Zuweisung eines Wertes zu einer Präprozessorvariablen, falls diese noch nicht definiert wurde

```
#ifndef M_PI
#define M_PI 3.14159
#endif
```

Desweiteren können Makros mit Parametern definiert

```
#define MAX(x,y) (x>y ? x : y)
```

und im Quelltext verwendet werden.

```
  cout << MAX(1.456 , a) << endl;
```

Mehr über Präprozessorbefehle ist u.a. in [God98] und [Str00, $A.11] zu finden.

11.2 Zeitmessung im Programm

Zum Umfang von C++ gehören einige Funktionen, welche es erlauben die Laufzeit bestimmter Programmabschnitte (oder des gesamten Codes) zu ermitteln. Die entsprechenden Deklarationen werden im Headerfile *time.h* bereitgestellt.

Ex1121.cc

```
//      Ex1121.cc
...
#include <time.h>               // contains clock()

int main()
{
 ...
 double time1=0.0, tstart;      // time measurment variables

//      read data
 ...
 tstart = clock();              // start

//      some code
 ...
 time1 += clock() - tstart;     // end
 ...
 time1 = time1/CLOCKS_PER_SEC;  // rescale to seconds

 cout << "  time = " << time1 << " sec." << endl;

 return 0;
}
```

Es können beliebig viele Zeitmessungen im Programm erfolgen (irgendwann verlangsamen diese aber ihrerseits das Programm!). Jede dieser Zeitmessungen benötigt einen Start und ein Ende, allerdings können die Zeiten verschiedener Messungen akkumuliert werden (durch einfaches addieren).

Im File *Ex1121.cc* wird der Funktionswert eines Polynoms vom Grade 20 an der Stelle x d.h., $s = \sum_{k=0}^{20} a_k \cdot x^k$, berechnet. Die 21 Koeffizienten a_k und der Wert x werden im File *input.1121* bereitgestellt. Der Funktionswert wird auf zwei, mathematisch identische, Weisen im Programm berechnet. Variante 1 benutzt die Funktion `pow`, während Variante 2 den Wert von x^k durch fortwährende Multiplikation berechnet. Ex1121.cc

Das unterschiedliche Laufzeitverhalten (Ursache !?) kann nunmehr durch Zeitmessung belegt werden und durch fortschreitende Aktivierung von Compileroptionen zur Programmoptimierung verbessert werden, z.B.

```
LINUX>  g++ Ex1121.cc
LINUX>  g++ -O Ex1121.cc
LINUX>  g++ -O3 Ex1121.cc
LINUX>  g++ -O3 -ffast-math Ex1121.cc
```

Der Programmstart erfolgt jeweils mittels

```
LINUX>  a.out < input.1121
```

11.3 Profiling

Natürlich könnte man in einem Programm die Zeitmessung in jede Funktion schreiben um das Laufzeitverhalten der Funktionen und Methoden zu ermitteln. Dies ist aber nicht nötig, da viele Entwicklungsumgebungen bereits Werkzeuge zur Leistungsanalyse (performance analysis), dem **Profiling** bereitstellen. Darin wird mindestens die in den Funktionen verbrachte Zeit und die Anzahl der Funktionsaufrufe (oft graphisch) ausgegeben. Manchmal läßt sich dies bis auf einzelne Quelltextzeilen auflösen. Neben den professionellen (und kostenpflichtigen) Profiler- und Debuggingwerkzeugen sind unter LINUX/UNIX auch einfache (und kostenlose) Kommandos dafür verfügbar.

```
LINUX>  g++ -pg Jacobi.cc matvec.cc
LINUX>  a.out
LINUX>  gprof -b a.out > out
LINUX>  less out
```

Der Compilerschalter `-pg` bringt einige Zusatzfunktionen im Programm unter sodaß nach dem Programmlauf das Laufzeitverhalten durch `gprof` analysiert werden kann. Der letzte Befehl (kann auch ein Editor sein) zeigt die umgeleitete Ausgabe dieser Analyse auf dem Bildschirm an.

11.4 Debugging

Oft ist es notwendig den Programmablauf schrittweise zu verfolgen und sich gegebenenfalls Variablenwerte etc. zu Kontrollzwecken ausgeben zu lassen. Neben

der stets funktionierenden, jedoch nervtötenden, Methode

```
...
cout << "AA " << variable << endl;
...
cout << "BB " << variable << endl;
...
```

sind oft professionelle Debuggingwerkzeuge verfügbar. Hier sei wiederum ein (kostenfreies) Programm unter LINUX vorgestellt.

```
LINUX>  g++ -g Ex1121.cc
LINUX>  ddd a.out &
```

Die Handhabung der verschiedenen Debugger unterscheidet sich sehr stark. Beim ddd-Debugger kann mit `set args < input.1121` das Eingabefile angegeben werden und mit `run` wird der Testlauf gestartet, welcher an vorher gesetzten Break-Punkten angehalten wird. Dort kann dann in aller Ruhe das Programm anhand des Quellcodes schrittweise verfolgt werden.

Literaturverzeichnis

[Cap01] Derek Capper. *Introducing C++ for Scientists, Engineers and Mathematicians*. Springer, 2001.

[CF88] Matthias Clauß and Günther Fischer. *Programmieren mit C*. Verlag Technik, 1988.

[Cor93] Microsoft Corp. *Richtig einsteigen in C++*. Microsoft Press, 1993.

[Dav00] Stephen R. Davis. *C++ für Dummies*. Internat. Thomson Publ., Bonn/Albany/Attenkirchen, 2. edition, 2000.

[Erl99] Helmut Erlenkötter. *C Programmieren von Anfang an*. Rowohlt, 1999.

[God98] Eduard Gode. *ANSI C++: kurz & gut*. O'Reilly, 1998.

[Her00] Dietmar Herrmann. *C++ für Naturwissenschaftler*. Addison-Wesley, Bonn, 4. edition, 2000.

[Her01] Dietmar Herrmann. *Effektiv Programmieren in C und C++*. Vieweg, 5. edition, 2001.

[HT03] Andrew Hunt and David Thomas. *Der Pragmatische Programmierer*. Hanser Fachbuch, 2003.

[Jos94] Nicolai Josuttis. *Objektorientiertes Programmieren in C++: von der Klasse zur Klassenbibliothek*. Addison-Wesley, Bonn/Paris/Reading, 3. edition, 1994.

[KPP02] Ulla Kirch-Prinz and Peter Prinz. *Alles zur objektorientierten Programmierung*. Galileo Press, 2002.

[KPP03] Ulla Kirch-Prinz and Peter Prinz. *C++ für C-Programmierer*. Galileo Press, Okt. 2003.

[Mey97] Scott Meyers. *Mehr effektiv C++ programmieren*. Addison-Wesley, 1997.

[Mey98] Scott Meyers. *Effektiv C++ programmieren*. Addison-Wesley, 3., aktualisierte edition, 1998.

[OT93] Andrew Oram and Steve Talbott. *Managing Projects with make*. O'Reilly, 1993.

[SB95] Gregory Satir and Doug Brown. *C++: The Core Language*. O'Reilly, 1995.

[SK98] Martin Schader and Stefan Kuhlins. *Programmieren in C++*. RowohltVieweg, 5. neubearbeitete edition, 1998.

[Str00] Bjarne Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley, 4. aktualisierte edition, 2000.

[Str03] Thomas Strasser. *Programmieren mit Stil. Eine systematische Einführung*. dpunkt, 2003.

Index