Einführung in die Informatik und Programmierung Problemlösen in C++ (Informatik I)

Prof. Dr. Hans-Jürgen Buhl

1998

Fachbereich Mathematik (7)
Institut für Angewandte Informatik
Bergische Universität – Gesamthochschule Wuppertal

Interner Bericht der Integrierten Arbeitsgruppe
Mathematische Probleme aus dem Ingenieurbereich
IAGMPI – 9702
Oktober 2000

Praktische Informatik 04

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1a] Bjarne Stroustrup: C++ Programming Language, 3. Auflage, Addison-Wesley, 1997

[1b] Bjarne Stroustrup / Andrew Koenig: The Annotated C++ Language Standard, to appear

[1c] M. A. Ellis / B. Stroustrup: The Annotated C++ Reference Manual, Addison-Wesley, 1990

[2] P. Harmon / M. Watson: Understanding UML, 1997

[3] Ulrich Breymann: Die C++ Standard Template Library, Addison-Wesley, 1996

[4a] Scott Meyers: Effective C++, Addison-Wesley, 1992

[4b] Scott Meyers: More Effective C++, Addison-Wesley, 1996

[5] ISO/IEC 14882 International Standard: Programming Languages – C++, 27.7.1998

[6] Rechenberg / Pomberger: Informatik-Handbuch, Hanser-Verlag, 1997

[7] Werner u.a.: Taschenbuch der Informatik, Fachbuchverlag. Leipzig, 1995

[8] Duden Informatik, BI, 1993

Kapitel 0

Einleitung

Informatik:
In der Informatik werden Methoden und Verfahren der automatischen Informationsverarbeitung insbesondere mit Hilfe von Digitalrechnern behandelt. Die Informatik besteht aus

- Kerninformatik
  - Praktische Informatik (Betriebssysteme, Sprachen, Compiler, Software-Technologie, Tools, KI)
  - Technische Informatik/Computertechnik (Rechnerarchitektur, Rechnerbaugruppen, Peripheriegeräte, DVÜ, Rechnernetze)
  - Theoretische Informatik (Automatentheorie, Theorie formaler Sprachen, Codierungstheorie, Algorithmentheorie)
- Angewandter Informatik
  - Wirtschaftsinformatik
  - Rechtsinformatik
  - Medizinische Informatik
  - ...
  - Informatik und Gesellschaft

Information technology:
<business, jargon> (IT) Applied computer systems - both hardware and software, and often including networking and telecommunications, usually in the context of a business or other enterprise. Often the name of the part of an enterprise that deals with all things electronic.

The term “computer science“ is usually reserved for the more theoretical, academic aspects of computing, while the vaguer terms “information systems“ may include more of the human activities and non-computerised business processes like knowledge management.

Abbildung 1: Informatik und IT

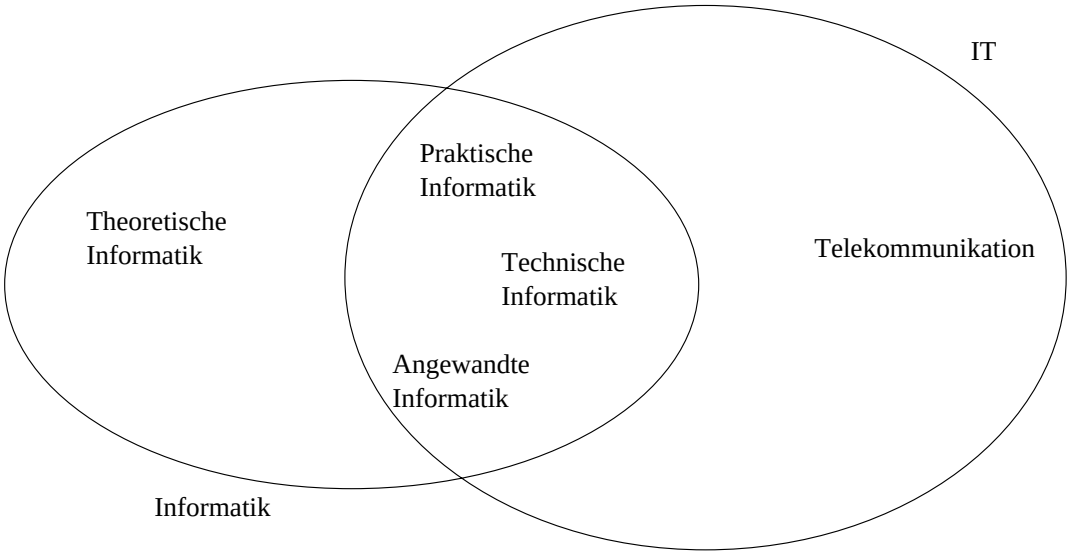


Tabelle 1: Zeittafel zur Entwicklung der Computertechnik

1833	Charles Babbage (1792-1871), Professor an der Universität Cambridge (Großbritannien), entwirft und baut einen programmgesteuerten mechanischen Rechenautomaten, die *Analytical engine*. Sie enthält ein 4-Spezies-Rechenwerk, 1000 Zahlenspeicher, Lochkartensteuerung und Ergebnisdruckwerk (nicht vollendet).
1941	Vorführung des ersten arbeitsfähigen programmgesteuerten Rechenautomaten ZUSE Z3 in Relaistechnik durch Konrad Zuse (geb. 1910)
1944	Inbetriebnahme des programmgesteuerten elektromechanischen Rechenautomaten Mark 1 von Howard H. Aiken (1900-1973).
1944/46	Formulierung der Prinzipien des *von-Neumann-Computers* (John von Neumann (1903-1975). Realisiert erstmals mit der EDVAC (1952/53)
1946	Inbetriebnahme des ENIAC, des ersten Computers mit Elektronenröhren durch John P. Eckert (geb. 1919) und John W. Mauchley (1907-1980). Beginn der Epoche der elektronischen Computer.
1951	Beginn der Serienproduktion elektronischer Computer mit der Anlage UNIVAC I der Firma Remington Rand. Beginn der 2. Computergeneration.
1955	Erster Computer mit Transistoren: TRADIC (Bell. Labor.)
um 1965	Computer-Familie IBM/360. Beginn der 3. Computergeneration. Der Begriff *Rechnerarchitektur* wird erstmals verwendet.
um 1965	Minirechner PDP-8 (Digital Equipment Corp.). Kleinere Rechner entstehen neben den Mainframes.
1969	Pilotprojekt Weitverkehrsrechnernetz ARPANET (USA) in Betrieb genommen.
1971	4-Bit-Mikroprozessoren i4004 der Firma INTEL Corp. Beginn der Mikroprozessorära. Rascher Übergang zu 8-Bit-Mikroprozessoren.
1975-1980	Personalcomputer auf Mikroprozessorbasis und die Software dafür werden zu Massenartikeln. Beginn der 4. Computergeneration.
um 1978	Erste Installationen lokaler Rechnernetze.
1978-1980	16-Bit-Mikroprozessoren kommen auf den Markt.
1979	Standardvorschlag der ISO »*Reference model of open system interconnections*« für Rechnernetze.
1981-1985	Personalcomputer mit 16-Bit-Mikroprozessoren werden marktbestimmend; insbesondere PC XT und AT von IBM und Kompatible dazu sowie das Betriebssystem MS-DOS der Firma Microsoft.
ab 1988	Personalcomputer mit 32-Bit-Mikroprozessoren kommen auf den Markt und lösen im Verlauf einiger Jahre die 16-Bit-Systeme ab.
ab 1996	Workstations mit 64-Bit-Mikroprozessoren

Quelle: D.Werner (Hrsg.): Taschenbuch der Informatik; Fachbuchverlag Leipzig, 1995

Tabelle 2: Zeittafel zur Vorgeschichte der Informatik

vor unserer Zeitrechnung	
um 4000	Älteste Zahlzeichen der Welt
um 1100	*Abakus* in Ostasien
um 450	Beschreibung des *Rechnens mit Steinen* durch Herodot (etwa 484-425)
um 325	Buch *Elemente* von Euklid (etwa 365 - 300). Enthält zahlreiche Algorithmen mit Axiomen, Regeln und Sätzen
unsere Zeitrechnung	
im 7.Jh.	Volles dezimales Stellenwertsystem in Indien
um 825	Rechenbuch von Muhammad Ibn Musa al-Hwarizmi (780 - 859). Auf den Titel der lat. Übersetzung aus dem 12. Jh. soll der Begriff Algorithmus zurückgehen.
1518 - 1550	Rechenbücher von Adam Ries (1492 - 1559). Formalisierung der dezimalen Rechenmethoden
1614	Logarithmentafeln von John Napier (1550 - 1617)
1623 - 1624	*Rechenuhr* von Wilhelm Schickhardt (1592 - 1635). Mechanismus für die vier Grundrechenarten
1641	Mechanische 2-Spezies-Rechenmaschine von Blaise Pascal (1623 - 1662). Etwa 40 verschiedene Ausführungen werden gebaut.
1671 - 1694	Entwicklung von mechanischen 4-Spezies-Rechenmaschinen durch Gottfried Wilhelm Leibniz (1646 - 1716)
1679	Leibniz entwickelt das duale Zahlensystem und Rechenregeln für die Dualarithmetik.
1808	Joseph-Marie Jacquard (1752 - 1834) erfindet die Lochkartensteuerung für Webstühle
um 1820	Aufnahme der ersten Fabrikproduktion von mechanischen 4-Spezies-Rechenmaschinen [*Arithmometer* von Christian Thomas (1785 - 1870)] in Paris
um 1855	Begründung der Boolschen Algebra durch George Boole (1815 - 1864)
ab 1882	Lochkarteneinrichtungen (zunächst für Volkszählungen) durch Hermann Hollerith (1860 - 1929). Damit beginnt die Datenverarbeitung mittels Lochkartenmaschinen.
1906	Erfindung der Elektronenröhre (Triode) durch Lee de Forrest (1873 - 1961) und Robert von Lieben (1887 - 1913)
1929	Erste elektrisch angetriebene mechanische 4-Spezies-Rechenmaschine mit Ergebnisdruck (Mauser-Cordt)
1947	Erfindung des Spitzentransistors durch John Bardeen, Walter H. Braittain und William Shockley
1961	Erster integrierter Schaltkreis (Fa. Fairchild)
1961	Anita: erster vollelektronischer 4-Spezies-Tischrechner (Fa. Sumlock). die Ablösung der mechanischen Rechner beginnt.

Kapitel 1

Einfache Problemlösungen mit dem Computer

1.1 Nachrichten (zunächst) zur Ausgabe auf der Konsole:

Häufig benötigt:

Nachrichten textueller Art, die z.B. auf die Konsole ausgegeben, per Datennetz ausgetauscht ... werden sollen.

Beim *objektbasierten Problemlösen* konzipiert man die für eine Problemlösung nötigen

Objekte

mit den jeweiligen

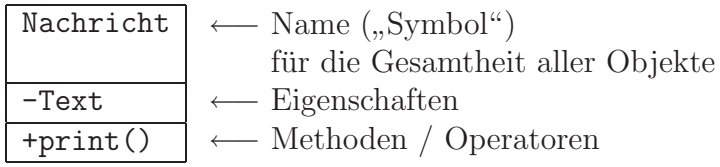
Eigenschaften

und

auszuführenden Operationen (Methoden)

In unserem Beispiel etwa

Abbildung 1.1: Klasse `Nachricht`



Durch `-` werden nur innerhalb der Klasse `Nachricht` benutzbare Eigenschaften bzw. Methoden gekennzeichnet, durch `+` solche, die auch außerhalb benutzt werden können.

Exemplare (Instanzen) der Klasse `Nachricht` sind dann etwa

Abbildung 1.2: Instanzen von `Nachricht`

Textuelle Inhalte können in `C++` etwa in Form von in Anführungsstrichen eingeschlossenen Texten spezifiziert werden.
Um Exemplare einer

Klasse (Mengen von Objekten, Typen)

zu erzeugen, benötigt man eine Operation, die man „Konstruktor-Operation“ nennt. Jede Klasse muß mindestens eine solche Operation besitzen, weshalb man ihr den Namen der Klasse gibt und diese in den Klassendiagrammen nicht besonders aufführt, außer wenn man spezielle Eigenschaften der Konstruktoren spezifizieren will. Benutzt wird der Konstruktor, um Exemplare der Klasse zu erzeugen:

```
...
Nachricht Begruessung("Herzlich willkommen");
Nachricht Abschied("Auf wiedersehen");
...
```

Bemerkung: Exemplar (Element einer Menge, Wert eines Typs)

Hier wird unter dem Namen `Begruessung` ein Exemplar der Klasse `Nachricht` erzeugt, das mit dem Wert `"Herzlich willkommen"` initialisiert wird.

Andere Beispiele:

Abbildung 1.3: Weitere Beispiele für Klassen

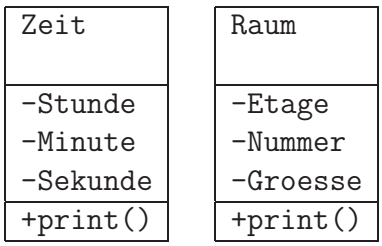


Abbildung 1.4: Klassendiagramm von `Nachricht`

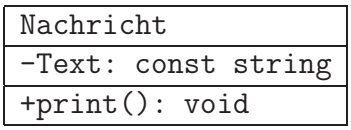
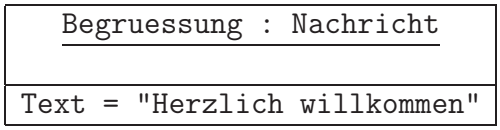
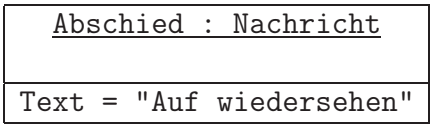


Abbildung 1.5: Objektdiagramme zur Klasse `Nachricht`





Die `Nachricht`-Methode `print()` ist eine Funktion ohne Ergebnistyp, was man mittels `print():void` kennzeichnet. Außerdem soll die Klasse `Nachricht` (genauer: das einzige Attribut `Text` der Klasse) nur einmalig (bei Erzeugung durch den Konstruktor) einen Wert zugewiesen bekommen, der von da ab konstant erhalten bleibt, was mittels

```
Text : const string;
```

gekennzeichnet wird. `string` ist eine in `C++` vordefinierte Klasse für (textuelle) Zeichenketten.

Die Signatur von Konstruktoren und anderen Methoden:

```
Nachricht()                 // parameterlos
Nachricht(const char*)      // mit einem Zeichenketten-
                            // parameter
void print()                // parameterlos ohne Ergebnis
```

Von Konstruktoren unterschiedliche Methoden werden in der folgenden Art zur Ausführung gebracht:

```
Begruessung.print();
```

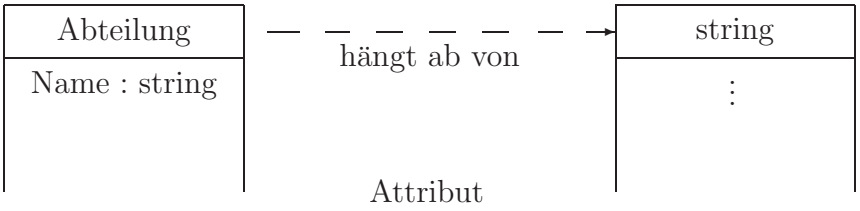
(Exemplarname, Methodenname, Parameterliste)

Signaturen in

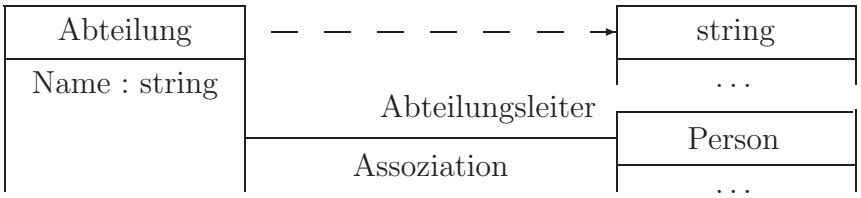
Klassendiagrammen	`C++`-Programmen
`Text: const string`	`const string Text;`
`getText():string`	`string getText();`
`print()`	`void print();`
`Nachricht(const char*)`	`Nachricht(const char*);`

Weitere Klassen und Beziehungen zwischen Objekten:

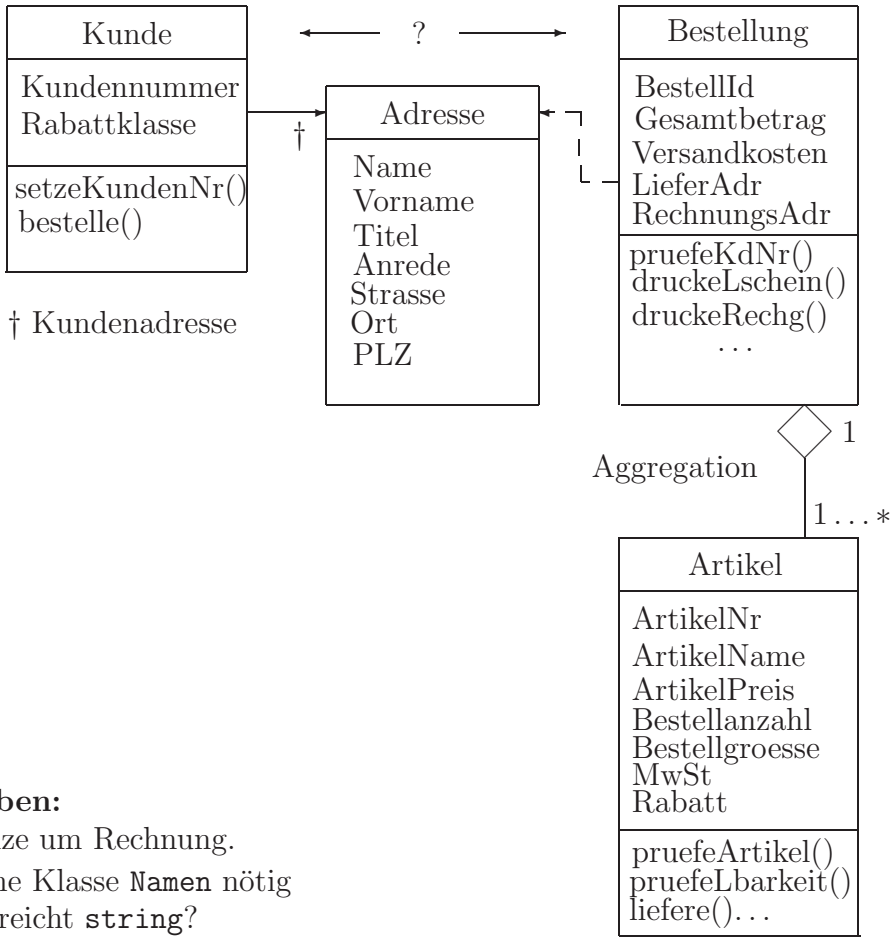
a)



b)



c)



† Kundenadresse

Aufgaben:

- Ergänze um Rechnung.
- Ist eine Klasse `Namen` nötig oder reicht `string`?

Literale vom Zeichenkettentyp `const char*` sind:

```
"Fachbereich Mathematik",
    "Programmierkurs"
          ⋮
```

Texte dürfen sich nicht über mehrere Zeilen erstrecken. Um einen zweizeiligen Text

```
Fachbereich 7
Bergische Universität
```

als C++-Objekt zu realisieren, ist folgendermaßen vorzugehen:

```
"Fachbereich 7\nBergische Universität"
```

Hierbei steht die Symbolfolge `"\n"` für einen Zeilenumbruch innerhalb einer Zeichenkette.
Texte (auch mehrzeilige) werden also als in Anführungsstriche eingeschlossene Zeichenfolgen realisiert.
Um zum Beispiel den festen (konstanten) Text

```
"Willkommen zum Programmierkurs\n"
```

(mit einem Zeilenende `"\n"` am Schluß) auf die Computerkonsole auszugeben, kann man das folgende einfache Programm schreiben:

```
#include <iostream>

using namespace std;

int main()
{
  cout << "Willkommen zum Programmierkurs\n";
  return 0;
}
```

Das „Programm“ gibt man mittels eines Editors (z.B. `xemacs`) ein, speichert es auf die Festplatte des Computers (etwa unter dem Dateinamen `first.cc`) und kann es sodann mittels des Kommandos

```
% make first <cr>
```

(`<cr>` steht für das Drücken der Wagenrücklauftaste ⏎) „übersetzen“ in einen vom Computer ausführbaren Auftrag, der dann nach erfolgreicher Übersetzung in der Datei `first` zur Verfügung steht und mittels

```
% first <cr>
```

beliebig häufig ausgeführt werden kann.

Aufgabe:

a) Einführung in Computerbedienung: Einloggen, ...

b) obiges Beispiel durchführen

c) Text variieren

Bemerkungen zum Programm `first.cc`:
Jedes Programm mit Ein- bzw. Ausgabe hat die vordefinierte Klasse `iostream` zu benutzen, was mittels

```
#include <iostream>

int main()
{
  std::cout << "Willkommen zum Programmierkurs\n";
  return 0;
}
```

geschieht. Alle vordefinierten Objekte sind mittels

```
std::OBJEKTNAME
```

verfügbar, etwa auch

```
#include <iostream>
#include <cmath>

int main()
{
  std::cout << std::sin(3.1415) << "\n";
}
```

Um nicht immer die volle Namensspezifikation angeben zu müssen, kann man die Zeile

```
using namespace std;
```

benutzen.
Die Methode `main` auf globaler Ebene ist die Methode, die bei Start eines ausführbaren übersetzten Programmes (Binaries) zur Ausführung kommt. `main()` gibt immer einen ganzzahligen Ergebniswert (`int`) an das Betriebssystem zurück, der per Konvention den Wert Null (`0`) hat, wenn die Programmausführung erfolgreich verläuft, und ansonsten einen im Einzelfall definierten Wert ungleich Null bekommt. Im „erfolgreichen“ Falle kann man die „Anweisung“

```
return 0;
```

auch der Einfachheit halber einsparen.

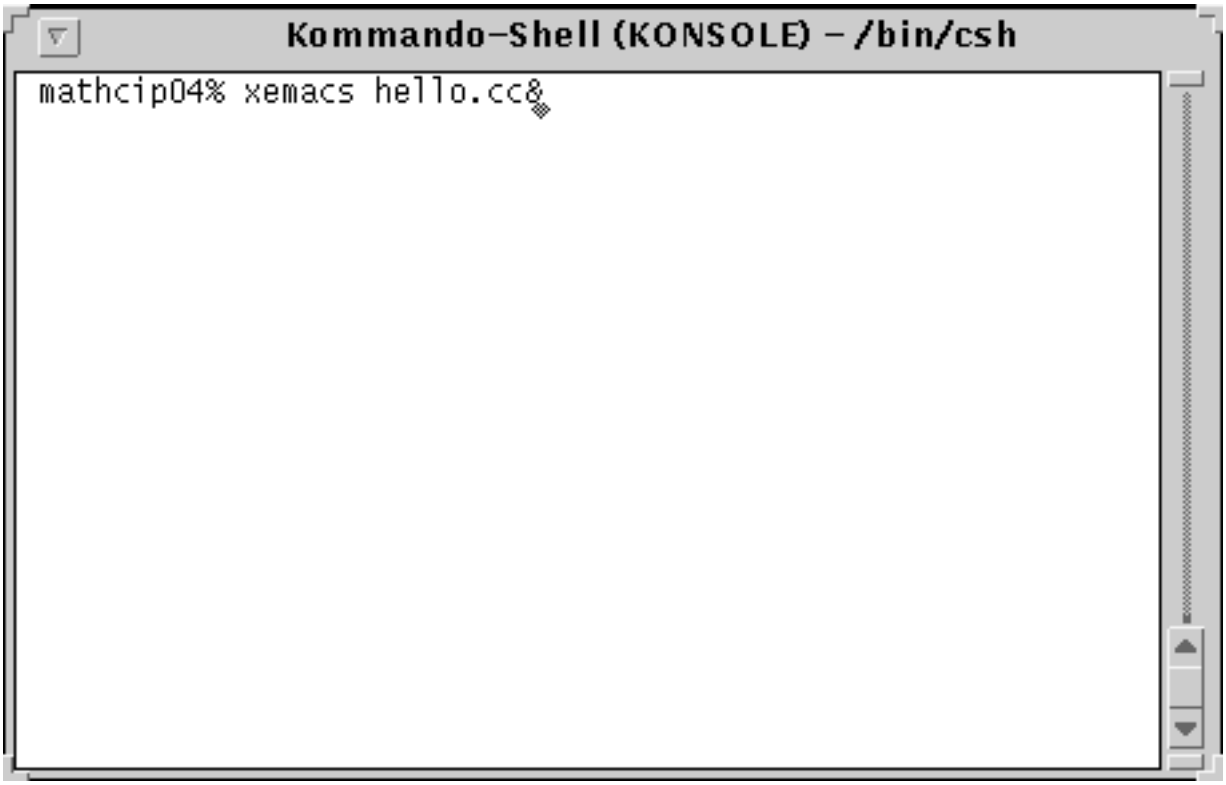
Tabelle 1.1: `C++` Library Headers

<algorithm>	<iomanip>	<list>	<queue>	<typeinfo>
<bitset>	<ios>	<locale>	<set>	<utility>
<complex>	<iosfwd>	<map>	<sstream>	<valarray>
<deque>	<iostream>	<memory>	<stack>	<vector>
<exception>	<istream>	<new>	<stdexcept>	
<fstream>	<iterator>	<numeric>	<streambuf>	
<functional>	<limits>	<ostream>	<string>	

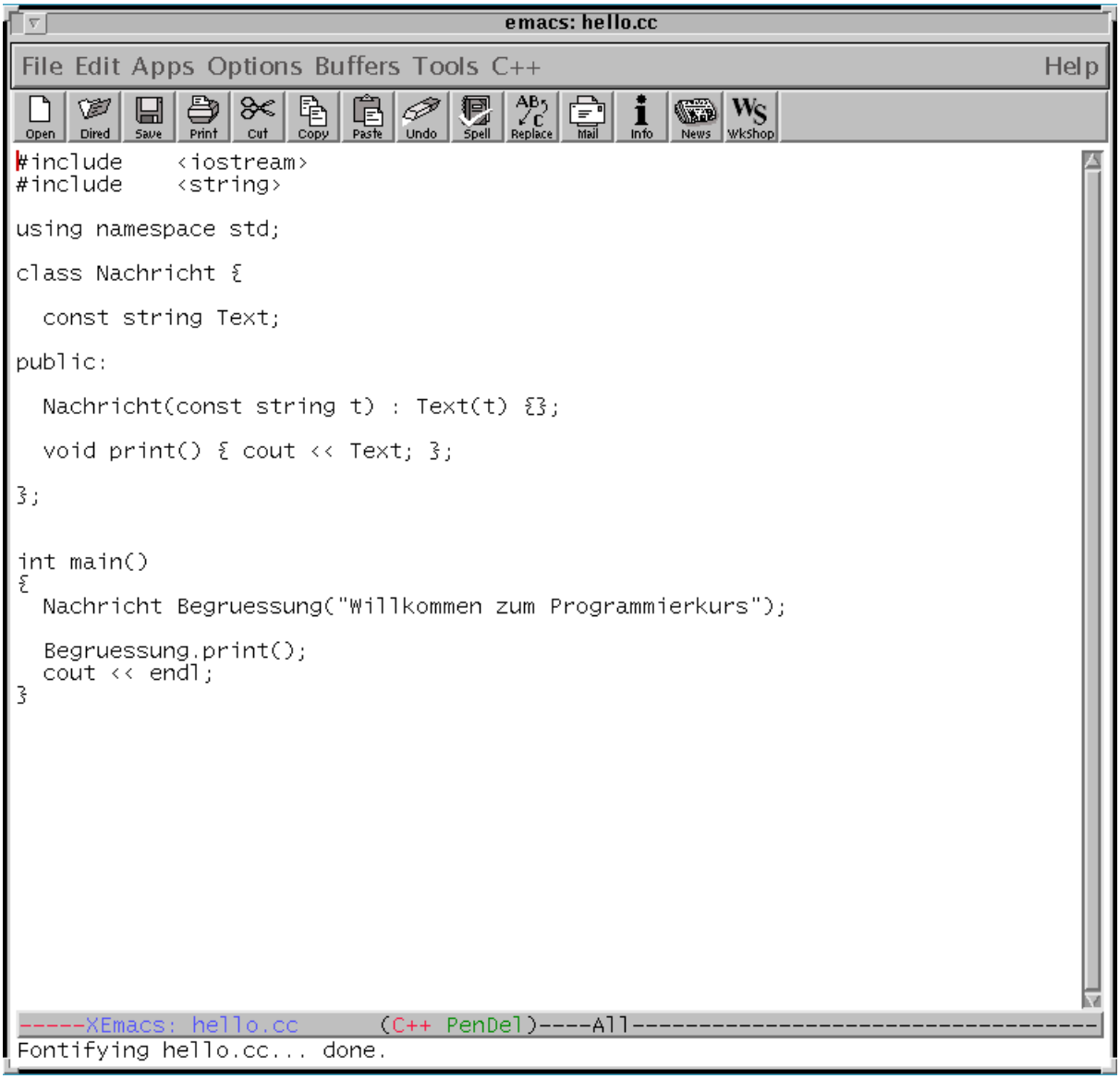
Tabelle 1.2: `C++` Headers for `C` Library Facilities

<cassert>	<ciso646>	<csetjmp>	<cstdio>	<cwchar>
<cctype>	<climits>	<csignal>	<cstdlib>	<cwctype>
<cerrno>	<clocale>	<cstdarg>	<cstring>	
<cfloat>	<cmath>	<cstddef>	<ctime>	

Starten der Entwicklungsumgebung:
Im Commandtool



das `xemacs`-Kommando aufrufen und den Text eintippen:



Bemerkung:

a) `endl` steht für `"\n"` (end of line)

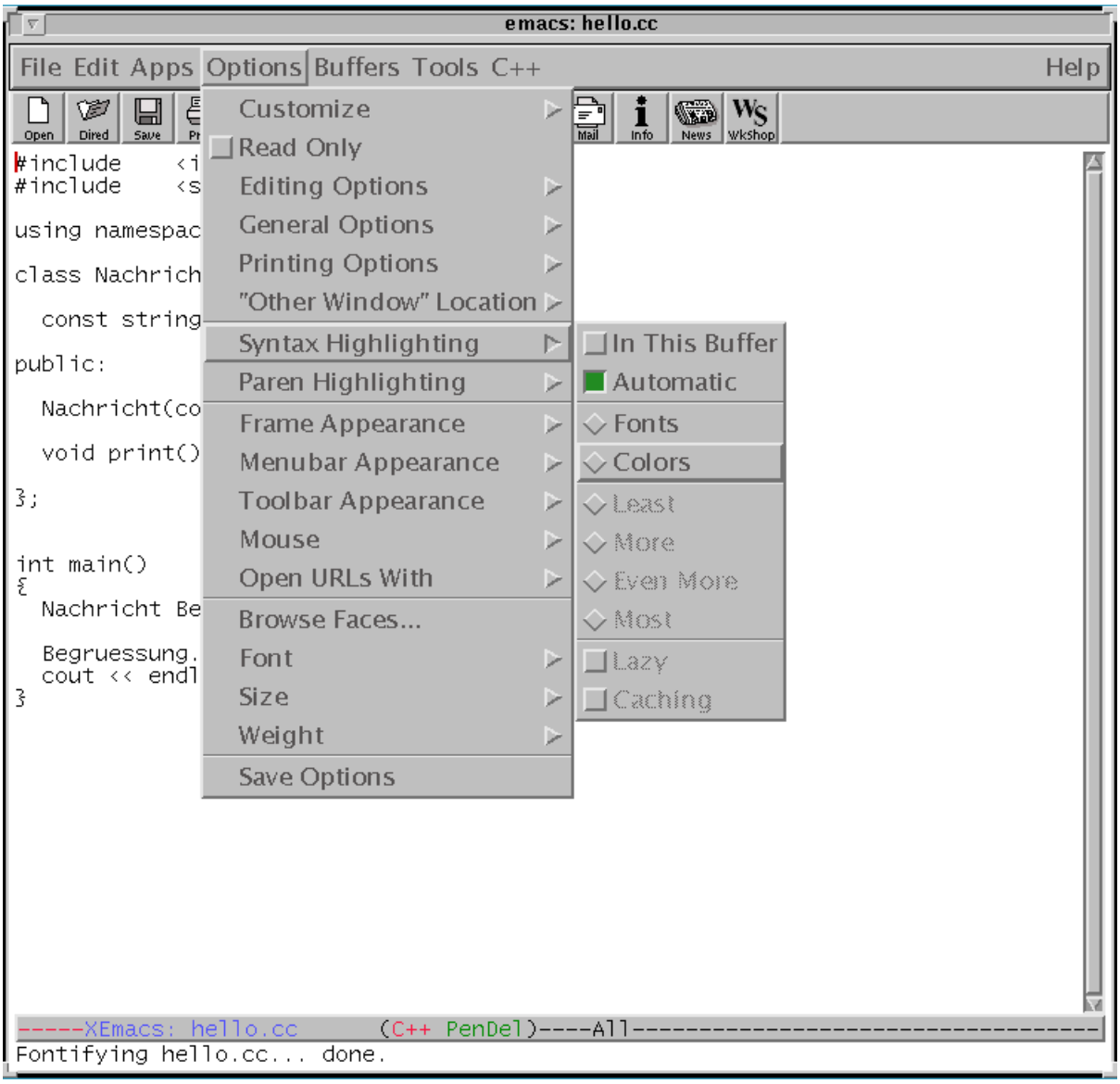
b) Innerhalb einer Klasse können die eigenen Attribute und Methoden ohne

`Klassenname::Objektname`

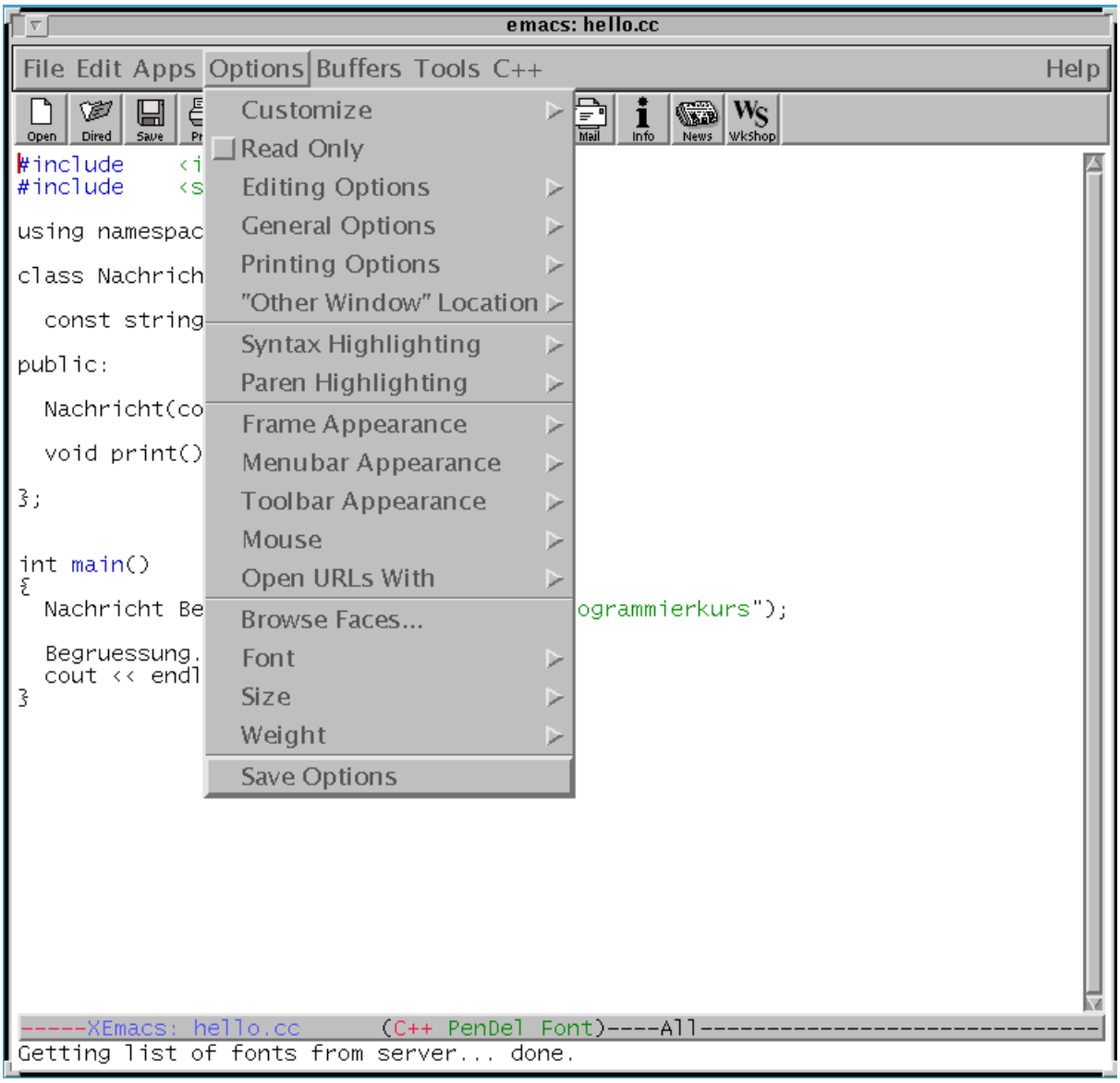
benutzt werden, während bei Trennung von Deklaration und Definition folgendermaßen vorgegangen werden muß:

```
class Nachricht {
   ...
   void print();
};

void Nachricht::print(){
  cout << Text;
};
```

Zuvor `Colors`

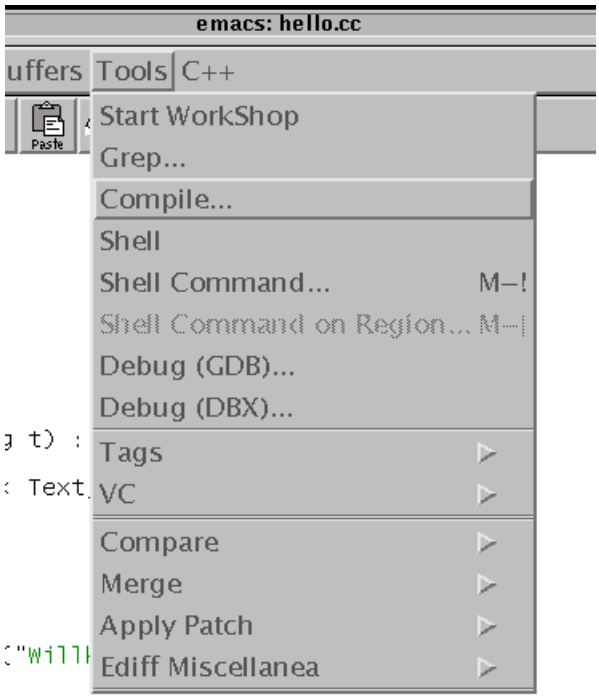


anklicken und permanent speichern:

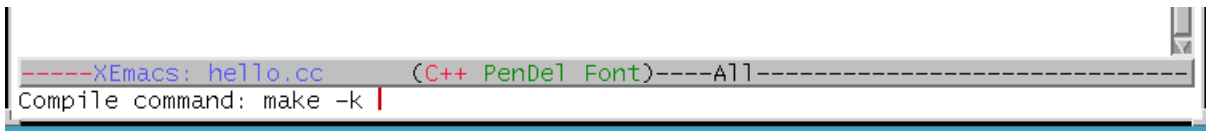


(Save Options)

Nun das Compile-Icon oder den Menüpunkt „Compile“ anwählen:



und die im Editor-Fenster unten erscheinenden Zeile

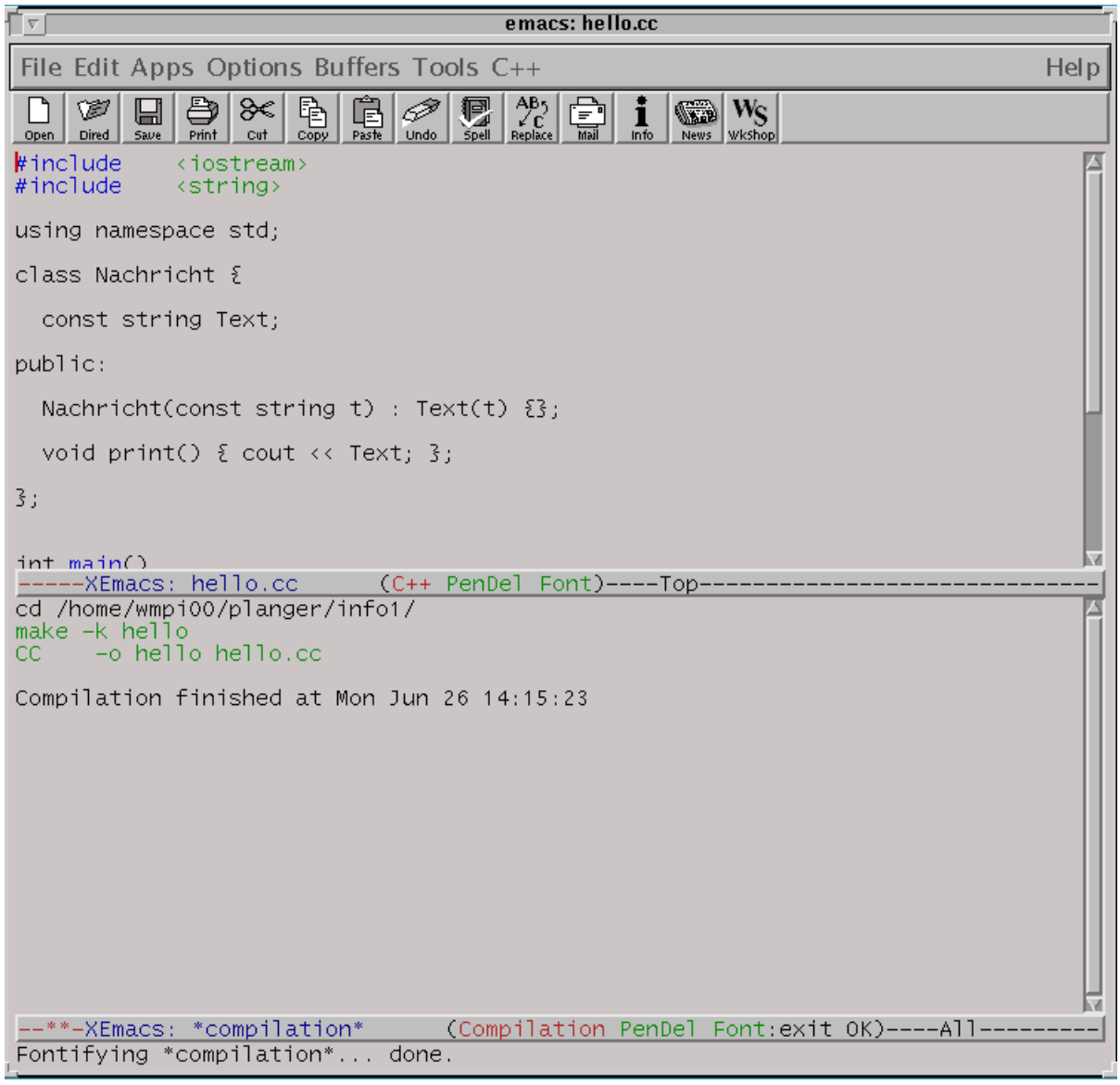


zu

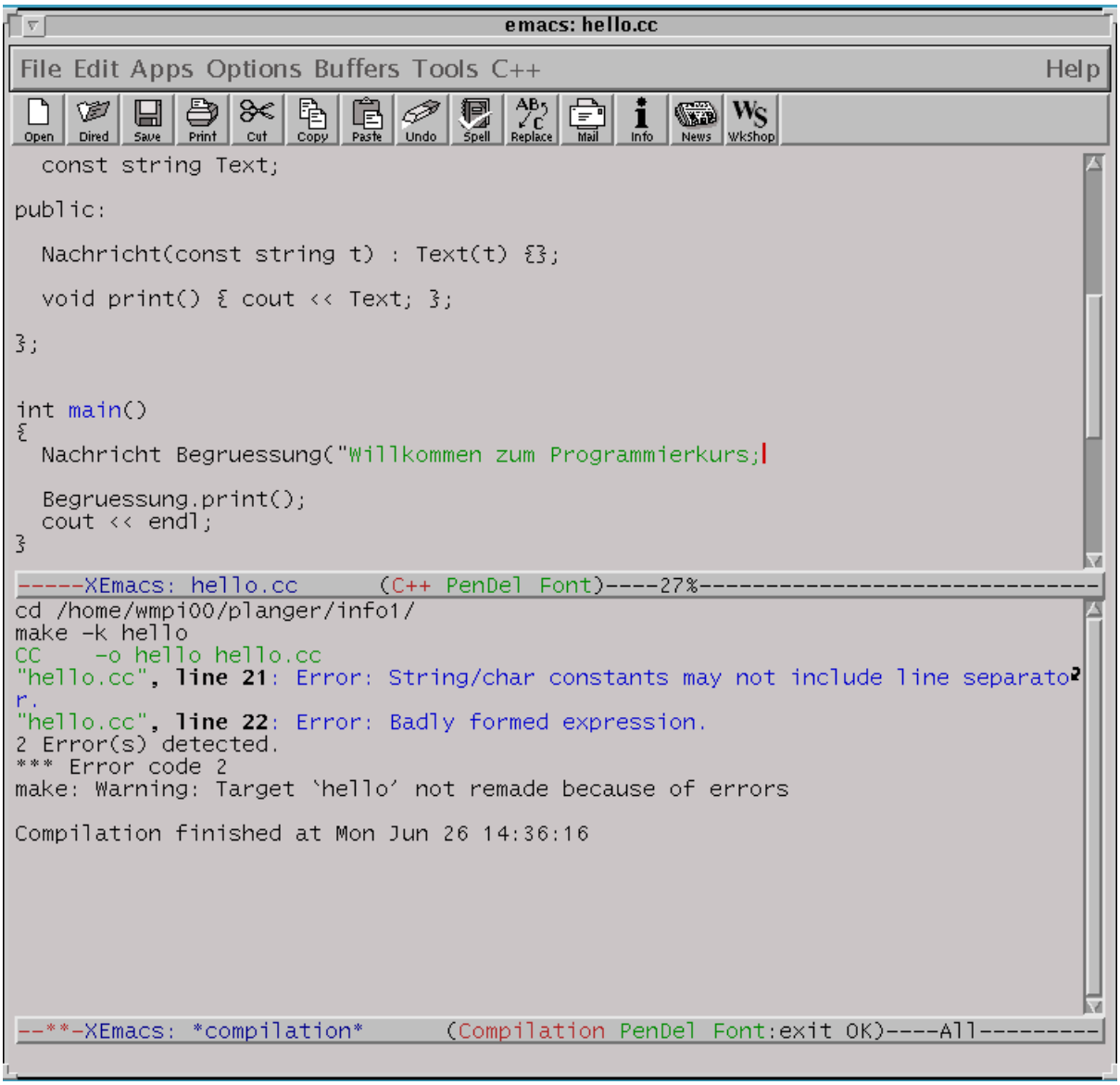
```
Compile command: make -k hello <cr>
```

ergänzen.

Wenn beim Übersetzen keine Fehler auftraten, wird die erfolgreiche Übersetzung in der unteren Hälfte des `xemacs`-Fensters mitgeteilt.



Eventuell auftretende Fehler oder Warnungen werden ebenfalls in der unteren Hälfte des `xemacs`-Fensters angezeigt:



Die Fehlermeldungen werden beim Überstreichen durch die Maus farbig hinterlegt (anklickbar) und Sie können bei Anklicken durch die mittlere Maustaste im Editorfenster direkt an die Fehlerstelle geführt werden.

Bemerkung: Folgefehler

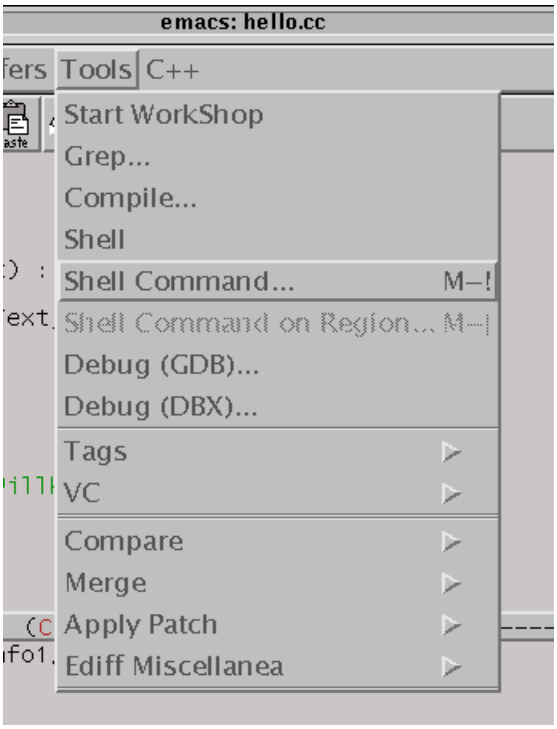
Sind keine Fehler mehr vorhanden, so kann das Programm gestartet werden:

a) In einem Commandtool mit

```
% ./hello <cr>
```

(Vgl. hierzu auch den Aufruf des `xemacs` (s.o.))

b) Im `xemacs`

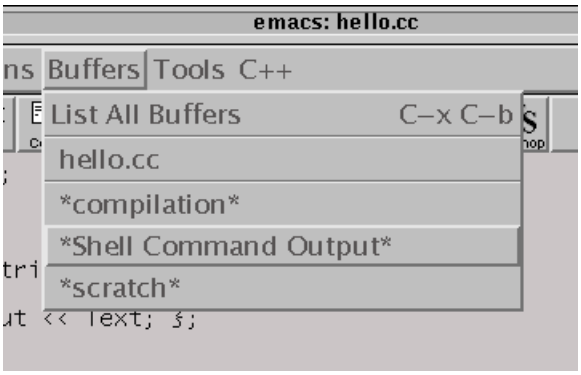


und ähnlich wie bereits oben gesehen die untere Zeile

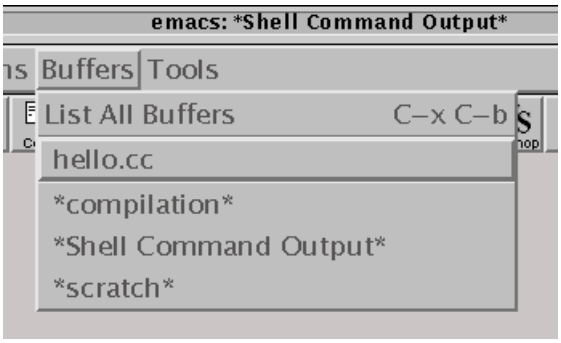
zu

```
Shell command: ./hello <cr>
```

ergänzen.
Die Ausgabe des Programmes kann man sich dann mittels



anschauen. Durch



kehrt man dann wieder zum Quelltext zurück.

Es wird häufig vorkommen, daß das Programm – obwohl es fehlerfrei übersetzt wurde – noch nicht einwandfrei bzw. wunschgemäß funktioniert. Um die noch vorhandenen logischen Fehler („Bugs“) zu beseitigen, wäre es wünschenswert, wenn man Zeile für Zeile nachvollziehen könnte, was das übersetzte Programm tut. Für diesen Zweck gibt es sogenannte *Debugger*.
Auf den Workstations steht uns eine integrierte `C++`-Entwicklungsumgebung, der `workshop`, zur Verfügung, den wir im folgenden exemplarisch benutzen wollen:

a) **Vorbereitung:**
Zu allererst muß man sicherstellen, daß für den Debugger-Betrieb die Environment-Variable `CPPFLAGS` richtig gesetzt ist. Dies erreicht man in einer `C`-Shell dadurch, daß man folgende Schritte ausführt:

- Im `commandtool`

  `% xemacs ~/.cshrc& <cr>`

  eingeben, um die Datei mit den Systemeinstellungen zu editieren.
- Gegebenenfalls noch die Zeile

  `setenv CPPFLAGS -g`

  ergänzen und die Datei abspeichern.

b) **Workshop starten:**
Es gibt 2 Möglichkeiten, `workshop` zu starten:

(a) Im gestarteten `xemacs` auf das `workshop`-Icon klicken.

(b) Im `commandtool`

`% workshop&  <cr>`

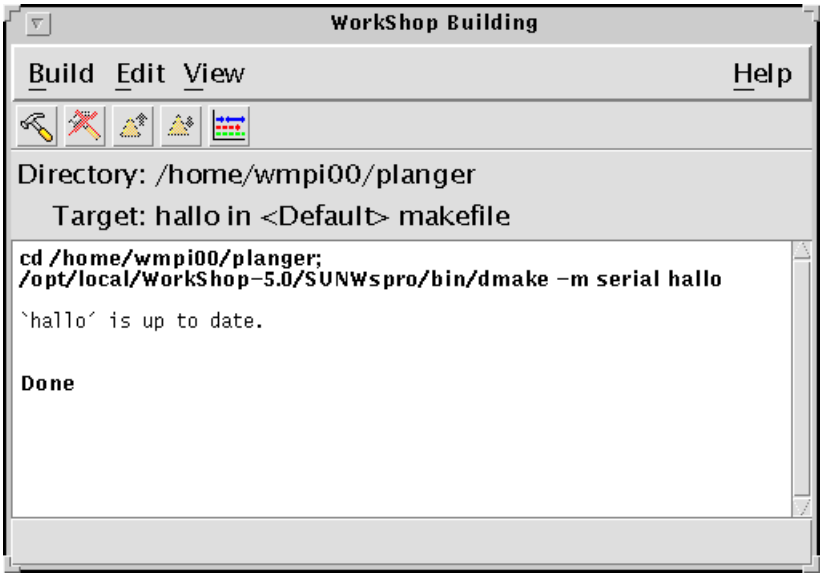
eingeben.

Es erscheint folgendes Fenster:

Hat man `workshop` nach der zweiten Methode gestartet (also im

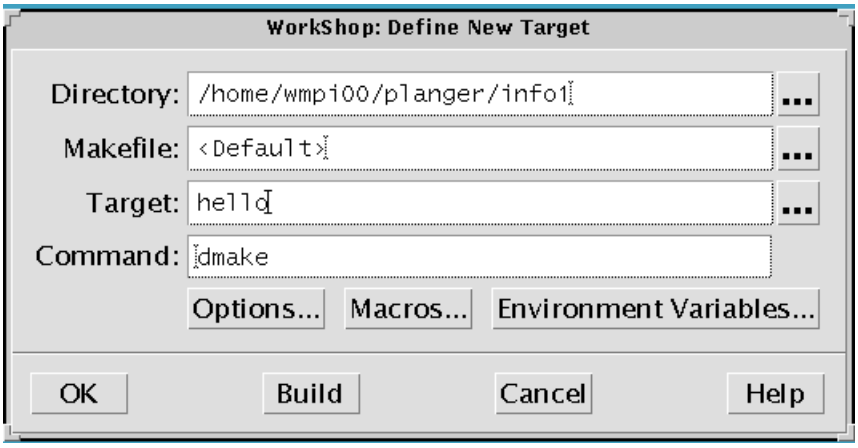
`commandtool`), so kann man durch einen Klick auf das linke Icon in der `workshop`-Menüleiste eine Datei auswählen, die dann in einen neuen, von `workshop` gestarteten `xemacs` geladen wird.

c) **Projekt übersetzen bzw. compilieren:**
Hierzu klickt man das zweite Icon von links an, woraufhin folgendes Fenster geöffnet wird:



Also wählt man im Menü `Build` den Punkt `New Target` an.

In dem nun erscheinenden Fenster trägt man in dem String-Feld `Target` den Namen der zu compilierenden Datei (ohne Endung `.cc`) ein, hier also z.B. `hello`:



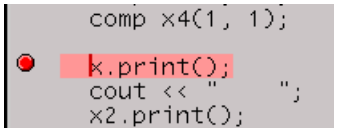
Nun kann man das Projekt compilieren. Dies geschieht durch einen Klick auf das `compile`-Icon entweder im `xemacs`-Fenster, im `workshop`-Fenster oder im zuletzt aufgerufenen `WorkShop Building`-Fenster. Natürlich kann man das Projekt auch auf die bereits bekannte Weise compilieren (s.o).

d) **Projekt debuggen:**
Trat beim Compilieren kein Fehler auf, kann man nun den Debugger starten. Dies erfolgt durch einen Klick auf das `Debug`-Icon (drittes Icon von links im `workshop`-Fenster). Es erscheint nun das Debugger-Fenster auf dem Bildschirm. Der Debugger bietet folgende Möglichkeiten, den Ablauf eines Programmes nachzuvollziehen:

- Setzen von sogenannten *Breakpoints*
  Hierzu geht man wieder in das `xemacs`-Fenster und wählt im `main`-Block eine Zeile aus, indem man mit dem Cursor eine Zeile anklickt. Nun kann man den Breakpoint setzen, indem man das zweite Icon von links in der Werkzeugleiste des `xemacs` anklickt.

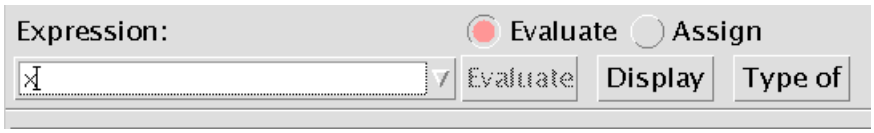
  Die gewählte Zeile wird nun im Editorfenster graphisch hervorgehoben:



Einen gesetzten Breakpoint kann man löschen, indem man wieder mit dem Cursor die Zeile des Breakpoints anwählt und in der Werkzeugleiste das Icon für das Löschen eines Breakpoints anwählt (viertes Icon von links).

- Ablauf des Programmes im Debugger
  Wenn man nun im Debugger-Fenster auf das `Start`-Icon klickt, wird das Programm bis zum Erreichen eines evtl. gesetzen Breakpoints „normal“ ausgeführt. An der Stelle des Breakpoints hält der Debugger die Abarbeitung des Programmes an. Auf diese Weise kann man „in aller Ruhe“ analysieren, was das Programm bis zu diesem Zeitpunkt bewirkt hat. Den bisherigen Output des Programmes kann man im Fenster mit dem Titel `WorkShop Program Input/Output` betrachten. Die aktuelle Zeile, an der sich das Programm gerade befindet, wird im `xemacs`-Fenster übrigens durch einen grünen Pfeil nach rechts gekennzeichnet. Will man die Abarbeitung des Programmes fortsetzen, so klickt man in der Werkzeugleiste des Debuggers auf das Icon mit dem grünen Pfeil nach unten. Das Programm läuft nun weiter bis zum nächsten gesetzten Breakpoint, u.s.w.
  Auf diese Weise (d.h. durch das Setzen von mehreren Breakpoints) kann man das Programm in mehrere übersichtliche Abschnitte unterteilen, deren Ablauf man leichter untersuchen kann.
  An besonders kritschen Stellen des Programmes kann man auch in den Einzelschrittbetrieb übergehen. Hierzu klickt man nach Erreichen eines Breakpoints (oder am Beginn des Programmes) in der Werkzeugleiste des Debuggers einfach auf das Icon mit dem grünen Pfeil nach rechts. Es wird nur eine einzige Zeile ausgeführt und dann sofort wieder angehalten. Durch Klick auf das Icon mit dem grünen Pfeil nach unten wird das Programm wieder „normal“ bis zum Erreichen des nächsten Breakpoints abgearbeitet.
  Hier noch ein Bild der Werkzeugleiste des Debuggers:

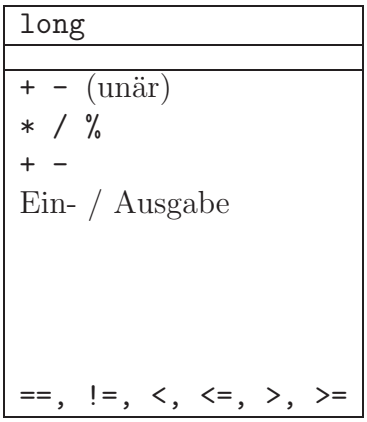
- Beobachten von Variablen und Ausdrücken:
  Häufig gewinnt man durch bloßes Setzen von Breakpoints noch keinen Aufschluß über eventuelle logische Fehler. Es wäre z.B. wünschenswert, wenn man das Programm nicht nur an Hand des Outputs beobachten könnte, sondern wenn man auch die Änderung einzelner Variablen nachvollziehen könnte. Auch hierfür stellt unser Debugger ein geeignetes Instrument zur Verfügung. Bearbeitet man beispielsweise ein Programm, in dessen `main`-Block die `integer`-Variable `x` vorkommt, so trägt man dies einfach in das `Expression`-Feld des Debuggers ein:



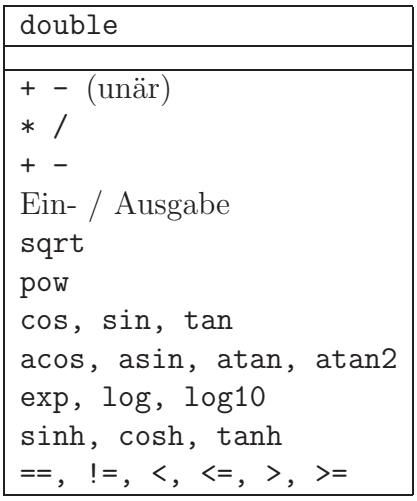
  Durch Klick auf den `Display`-Button öffnet sich ein Fenster mit dem Titel `WorkShop Data Display`. Wenn das Programm gerade nicht läuft, steht im Fenster `x = <not active>`.
  Startet man nun das Programm im Debugger (s.o.), kann man im `Display`-Fenster die Änderung von `x` im Programmverlauf nachvollziehen.

Skalare Objekte der Programmiersprache `C++`:

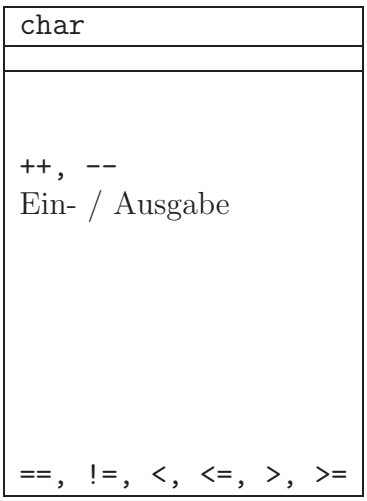
$\mathbb{B}$: `true, false`



„$\mathbb{Z}$“ : $-5, 0, 14, \ldots$



„$\mathbb{R}$“ : $-1.5E + 12, \ldots$



`char : 'a', '#',`
`...`

Ein Objekt `Rational Numbers` $\mathbb{Q}$ für exakt rechnende gebrochene Zahlen fehlt, muß also bei Bedarf selbst als Klasse implementiert werden.

Benötigt man in Programmen Objekte, die unterschiedliche Texte aufnehmen können, so stehen einem „Variablen“ des Objekttyps

`string`

für beliebig lange Zeichenketten zur Verfügung.

string
-Textinhalt
string() string(const char*) string(const string&) friend ostream& operator << (ostream&, const string&) string& operator= (const char*) string& operator= (const string&)

Ein Beispiel:

```
#include   <iostream>
#include   <string>

using namespace std;

int main()
{
  const string Begruessung("Willkommen zum Programmierkurs");

  cout << Begruessung << endl;
  return 0;
}
```

- Beschreibung: Erzeugung eines neuen Exemplars (einer Instanz) der „Klasse“ `string`, das ab sofort unter dem Namen `Begruessung` zur Verfügung steht und mit dem Text

  `"Willkommen zum Programmierkurs"`

  initialisiert ist.

- Mehrere Texte können nacheinander ausgegeben werden mittels

  `cout << "Text1" << "Text2";`

- Der Operator `<<` veranlaßt als „Nebeneffekt“ die Ausgabe und hat als Ergebniswert den Wert `cout`. Danach wird also ausgeführt:

  `cout << "Text2";`

Vordefinierte Namen, die nicht als eigene Bezeichner benutzt werden dürfen:

Tabelle 1.3: keywords

asm	do	inline	short	typeid
auto	double	int	signed	typename
bool	dynamic_cast	long	sizeof	union
break	else	mutable	static	unsigned
case	enum	namespace	static_cast	using
catch	explicit	new	struct	virtual
char	extern	operator	switch	void
class	false	private	template	volatile
const	float	protected	this	wchar_t
const_cast	for	public	throw	while
continue	friend	register	true	
default	goto	reinterpret_cast	try	
delete	if	return	typedef	

Tabelle 1.4: keywords (alternative Repräsentationen)

bitand	and	bitor	or	xor	compl
and_eq	or_eq	xor_eq	not	not_eq	

Ein erfolgreich beendetes Programm:

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

Der Ergebniswert eines Computerauftrages (Programms) ist ein ganzzahliger Wert:

- 0 für erfolgreiche Auftragsausführung
- $\neq 0$ (je nach Vereinbarung) für nicht erfolgreiche Ausführung

Ohne die explizite Anweisung `return` wird default-mäßig der Wert `0` als Ergebnis von `main` benutzt.

Beispiele für die Benutzung von `string`:

```
      ⋮

string Begruessung;
cout << Begruessung;

      ⋮

Begruessung = "1.Text";

      ⋮

Begruessung = "";

      ⋮

string Begr2("1.Text");

      ⋮

Begr2 = "2.Text";

      ⋮

string Begr3 = "1.Text";

      ⋮

string Begr4(Begr2);

      ⋮
```

```
const string Begruessung("Willkommen ... ");
```

führt den Bezeichner für ein nicht änderbares (konstantes) Exemplar von `string` mit Inhalt "Willkommen ..." ein:

> Bei häufiger Wiederholung der Zeichenkette braucht nur noch der (kürzere) Bezeichner benutzt werden.

Wertzuweisungen u.ä. führen dann schon beim Übersetzen zu Fehlermeldungen.

In der „Signatur" des Konstruktors

```
string (const string&)                    (*)
```

deutet `const` ebenfalls an, daß hier das Parameterobjekt durch Ausführung der Methode nicht geändert wird (Änderungen in einer Implementierung von (*) werden sofort vom Compiler mit einer Fehlermeldung zurückgewiesen !).

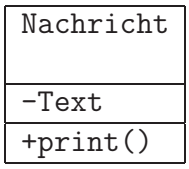
1.1.1 Qualität von Fehlermeldungen

Aufgabe:
Verschaffen Sie sich einen Überblick über die Fehlermeldungen des `C++`-Compilers:

- vergessenes `"`
- zusätzliches `"`
- zusätzlicher :
- `man` statt `main`
- `int main();`
  {
- `main()`
  {
- usw.

1.1.2 Eigene Klassen mit (restriktivem) Methodensatz

Wie Sie sehen erfüllt das Programm, das die Nachricht „Willkommen zum Programmierkurs“ mittels eines Exemplars der Klasse `string` realisiert, **nicht** die geforderte Spezifikation



da alle Methoden der Klasse `string` und nicht **nur** eine Methode, `print()`, auf das Exemplar `Begruessung` angewendet werden können.

Eine eigene Klasse kann folgendermaßen realisiert werden:

class Klassen_Name {

// Attribute (private)
// Methoden

public:

// Attribute
// Methoden

};

Um Nachrichten authentisch zu halten, soll ihnen ein Wert lediglich bei der Erzeugung durch einen Konstruktor zugewiesen werden können. Alle Methoden des Objekttyps `string` sollen nicht anwendbar sein, also:

```
class Nachricht {

  const string Text;

public:

  Nachricht(const string& t) : Text(t) { };

  void print() const { cout << Text; };

};
```

Anwendungsmöglichkeiten:

```
int main()
{
  Nachricht Begruessung("Willkommen zum Programmierkurs");
  Begruessung.print();
  cout << endl;
  return 0;
}
```

Andere mögliche Anwendungen:

```
Nachricht Begr2(Begruessung); // neue Nachricht, deren Text
                              // als Kopie des Textes von
                              // Begruessung gewonnen wird.

Nachricht Begr3("");

Begr3 = Begr2;                // nur falls nicht const
```

Unmögliche Anwendungen:

```
Begruessung.Text = "Ade"; // von ausserhalb der Klasse nicht zugreifbar!

cout << Begruessung;

    ...
```

Ebenfalls zum Fehler führt:

```
const Nachricht Begr4("ldots");

Begr4 = Begr2; // Fehler
```

```
////////////////////////////////////
// Datei:   hello.cc
// Version: 1.0
// Zweck:   Ausgabe einer Nachricht
// Autor:   Hans-Juergen-Buhl
// Datum:   17.09.1998
////////////////////////////////////

#include    <iostream>
#include    <string>

using namespace std;

class Nachricht {

  const string Text;

public:

  Nachricht(const string& t) : Text(t) { };

  void print() const { cout << Text; };

};

int main()
{
  Nachricht Begruessung("Willkommen zum Programmierkurs");

  Begruessung.print();
  cout << endl;

  Nachricht Begr2(Begruessung);
  Nachricht Begr3("");
  // Begr3 = Begr2;

  // Begruessung.Text = "Ade";
  // cout << Begruessung << endl;
  // cout << Begruessung.Text << endl;

  const Nachricht Begr4("Hello");
  // Begr4 = Begr2;

  return 0;
}
```

- **immutable attribute:**

  ```
  const string Text;
  ```

  ist ein unveränderbares / konstantes Attribut. Der Wert eines konstanten Attributes wird bei Instantierung eines Exemplars gesetzt und bleibt während der ganzen Lebenszeit des Objektes gleich!

- **read-only attribute:**
  Ein „read-only“-Attribut kann nicht direkt (weder durch den Konstruktor noch durch eine andere Methode) im Wert geändert werden, wohl aber indirekt durch Änderung eines anderen Attributwertes (das Attribut ist dann ein sogennantes *abgeleitetes Attribut*):

```
////////////////////////////
// Datei:   comp2.cc
// Version: 1.0
// Zweck:   default constructor
// Autor:   Hans-Juergen Buhl
// Datum:   17.09.1998
////////////////////////////

#include    <iostream>
#include    <cmath>

using namespace std;

class comp {

  double re;      // re == cos(Winkel) * Laenge
  double im;      // im == sin(Winkel) * Laenge
  double Winkel;  // -Pi <= Winkel < +Pi, {read only}
  double Laenge;  // >= 0.0              {read only}

public:

  comp() : re(0.0), im(0.0), Winkel(0.0), Laenge(0.0) {};

  comp( const double r, const double i ) : re(r), im(i),
                                           Winkel( atan2(im, re) ),
                                           Laenge( hypot(re, im) ) {}

  void print() const { cout << re << " + i * " << im; };

  void printPolar() const { cout << "Winkel= " << Winkel
                                 << " Laenge= " << Laenge; };
```

```
};

int main()
{
  comp x;
  comp x2(−2.0, −0.000000001);

  double d(3.5);
  comp x3(d, 4);

  x.print();
  cout << "  ";
  x2.print();
  cout << "  ";
  x3.print();
  cout << endl;

  x.printPolar();
  cout << endl;
  x2.printPolar();
  cout << endl;
  x3.printPolar();
  cout << endl;

  return 0;
}
```

Sollten Ihnen Programme mit Benutzung von

`<cstring>` und `new char[...]`

statt von

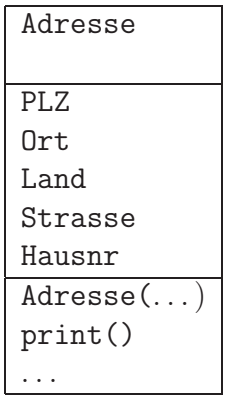
`<string>` und `string("...")`

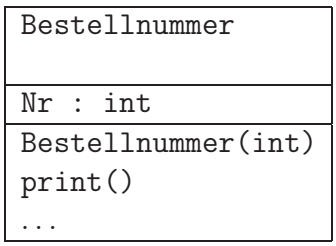
vorgelegt werden, so sind diese nicht im Sinne der Wiederverwendbarkeit von Klassen des C++-Standards realisiert worden!

Bemerkung: Solche „low-level“ Varianten sollten zugunsten der Klasse `string` vermieden werden (vergleiche Kapitel 2.2 Zeiger).

Aufgaben:

a) Konzipieren und testen Sie Klassen





b) Testen Sie:

```
class comp {

  double re,im;

public:

  comp(double a=0.0,
       double b=0.0) : re(a), im(b) {};

  void print() { cout << re << "+ i* " << im; };

}
```

1.1.3 Numerische Anwendungen

Eine einfache numerische Anwendung zur Berechnung des ggT:

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
  long x, y;
  cout << "Bitte zwei ganze Zahlen eingeben: ";
  cin >> x >> y;

  long a(labs(x));
  long b(labs(y));

  if (b > a)
    {
      long tmp(a);
      a = b;
      b = tmp;
    };

  long temp;
  while (b != 0)
    {
      temp = b;
      b = a % b;
      a = temp;
    };

  cout << "Ergebnis: " << a << endl;

  return 0;
};
```

Aufgabe: Wie heißt obiger Algorithmus ?

Die Benutzung von Vektoren $v = \begin{pmatrix} v_0 \\ \vdots \\ v_5 \end{pmatrix} \in int^6$

```
// partsum0.cc

#include <numeric>
#include <iostream>

using namespace std;

int main()
{
  const int dim(6);
  int numbers[dim] = { 1, 2, 3, 4, 5, 6 };
  int result[dim];

  partial_sum(numbers, numbers + dim, result);
  for (int i = 0; i < dim; i ++)
    cout << result[i] << ' ';
  cout << endl;

  return 0;
}
```

Oder ohne Hilfe von `<numeric>` selbst programmiert:

```
// partsum0a.cc

#include <iostream>

using namespace std;

int main()
{
  const int dim(6);
  int numbers[dim] = { 1, 2, 3, 4, 5, 6 };
  int result(0);

  for (int i = 0; i < dim; i ++)
    result += numbers[i];

  cout << result << endl;

  return 0;
}
```

Die Berechnung der euklidischen Norm eines Vektors

```
// partsum0b.cc

#include <iostream>
#include <cmath>

using namespace std;

int main()
{
  const int dim(6);
  int numbers[dim] = { 1, 2, 3, 4, 5, 6 };
  double result(0);

  for (int i = 0; i < dim; i ++)
    result += pow(numbers[i], 2);

  result = sqrt(result);

  cout << result << endl;

  return 0;
}
```

bzw. die Berechnung des arithmetischen Mittelwertes der Komponenten eines Vektors:

```
// partsum0c.cc

#include <iostream>
#include <cassert>

using namespace std;

double average(double v[], int dim)
{
  assert(dim > 0);
  double result(0);

  for (int i = 0; i < dim; i ++)
    result += v[i];

  return result/dim;
};
```

```
int main()
{
  const int dim(6);
  double numbers[dim] = { 1, 2, 3, 4, 5, 6 };

  cout << average(numbers, dim) << endl;

  return 0;
}
```

Zum Abschluß eine Ausarbeitung der Klasse `comp` für komplexe Zahlen:

```
////////////////////////////////
// Datei:   comp1a.cc
// Version: 1.0a
// Zweck:  globaler Operator +, Observatoren
// Autor:  Hans-Juergen Buhl
// Datum: 17.09.1998
////////////////////////////////

#include      <iostream>
#include      <iomanip>

using namespace std;

class comp {

  double re, im;

public:

  comp( const double a = 0, const double b = 0 ) : re(a), im(b) {};

  double getRe() const { return re; };
  double getIm() const { return im; };

  void print() const { cout << re << " + i * " << im; };

};

inline comp operator+ (const comp& c1, const comp& c2)
{
  return comp(c1.getRe() + c2.getRe(), c1.getIm() + c2.getIm());
};

int main()
{
  comp x;
  comp x2(2.5, 1.5);

  x2 = 1 + x2;
  x2.print();
  cout << endl;

  return 0;
}
```

1.1.4 Standardmäßig vorhandene Methoden:

Vom Compiler automatisch erzeugt werden folgende Methoden:

```
////////////////////////////////
// Datei:    empty.cc
// Version: 1.0
// Zweck:   "fast" leere Klasse
// Autor:   Hans-Juergen Buhl
// Datum:  28.09.1998
////////////////////////////////

using namespace std;

class Leer {

  int i;

public:

};

int main()
{
  Leer Zahl;
  // Leer Zahl2(2);
  Leer Zahl2(Zahl);

  Leer Zahl3;
  Zahl3 = Zahl;

  Leer* zahl_ptr;
  zahl_ptr = & Zahl2;

  const Leer Zahl4(Zahl2);
  const Leer* zahl_ptr2;
  zahl_ptr2 = & Zahl4;

  return 0;
}

////////////////////////////////
// Leer()
// Leer(const Leer&)
// Leer& operator= (const Leer&)
// Leer* operator& ()
// const Leer* operator& () const
////////////////////////////////
```

Hinweis:
Falls mindestens ein Konstruktor selbst definiert wird, gelten **nur** noch die selbst definierten Konstruktoren!

1.2 Hilfsmittel der strukturierten Algorithmenbeschreibung

Zur Steuerung des Kontrollflusses können Sie benutzen:

- Sequenz
- Fallunterscheidung (`if`, `if - else`, `switch (break)`)
- Wiederholung (`for`, `while`, `do while (break, continue)`)
- Delegation (Verbundanweisung, Block, Funktionsaufruf, Methodenaktivierung)
- Test und Abbruch (`assert, Exceptions`)

Wir werden diese Möglichkeiten im folgenden durch sog. Nassi-Shneidermann-Diagramme (andere Bezeichnung: Struktogramme) und durch Beispielprogramme näher erläutern.

A) Sequenz:

Aktion 1
Aktion 2
Aktion 3

```
k = i % j;
i = j;
j = k;
```

Spezialfall:

–

```
;   // leere Anweisung
```

B) Fallunterscheidung:

Bedingung: true	Bedingung: false
Aktion 1	Aktion 2

```
if (a<0)
  abs_a = −a;
else
  abs_a = a;
```

Bedingung: true	Bedingung: false
Aktion 1	–

```
min_a_b = a;

if (b<a)
  min_a_b = b;
```

Fallauswahl: Fall 1	Fall 2	Fall 3	Fall 4	···
Akt. 1	Akt. 2	Akt. 3	Akt. 4	···

```
cout << "Bitte wählen Sie:\n";
char answer='\n';
cin >> answer;

switch (answer) {
 case 'f' :
   activate_file_menu();
   break;
 case 'e' :
   activate_edit_menu();
   break;
 case 'q' :
   exit_program();
   break;
 default:
   cout << "unverständliche Eingabe\n";
}
```

Bemerkungen:

- Labels müssen konstante Ausdrücke und eindeutig sein!
- Ohne `break` würde die Anweisungsfolge nach unten weiter fortgesetzt!
- Bei Nichtauftreten und ohne `default` wird `switch` verlassen.

Das eben aufgeführte Beispielprogramm zur Mehrfachfallunterscheidung könnte man auch folgendermaßen realisieren:

```
if (answer == 'f')
  activate_file_menu();
else if (answer == 'e')
  activate_edit_menu();
else if (answer == 'q')
  exit_program();
else cout << "unverständliche Eingabe\n";
```

```
////////////////////////////////
// Datei:    quadGlei.cc
// Version: 1.0
// Zweck:   Loesung von quadratischen Gleichungen
//              Ax^2 + Bx + C == 0.0
// Autor:   Hans-Juergen Buhl
// Datum:  02.10.1998
////////////////////////////////


#include <iostream>
#include <complex>

using namespace std;

// typedef complex complexd;
typedef complex<double> complexd;

class quadGleichung{

  double A, B, C;

public:

  enum {KEINE, EINE, ZWEI, VIELE, UNGUELTIG} erg_typ;
  complexd x1,x2;

  quadGleichung( double a, double b, double c ):
    A(a), B(b), C(c), erg_typ(UNGUELTIG) {};

  void print() { cout << A << B << C << endl; };

  void solve() {

    // ...
    // berechne und setze Werte von erg_typ, x1, x2
    // ...

  }

};

int main()
{
  quadGleichung q1(1.0, 0.0, 1.0);
  q1.solve();
  if (q1.erg_typ != quadGleichung::UNGUELTIG) {
    switch (q1.erg_typ) {
    case quadGleichung::KEINE : cout << " . . . " << endl;
```

```
                                    break;
    case quadGleichung::EINE : cout << " . . . " << endl;
                                    break;
                                    // . . .
    default:                        cout << " . . . " << endl;
    };
  };

  return 0;

}
```

Aufgaben:

a) Ergänzen Sie die Methode `print()` um geeigneten erkärenden Text.

b) Programmieren Sie die Methode `solve()`.

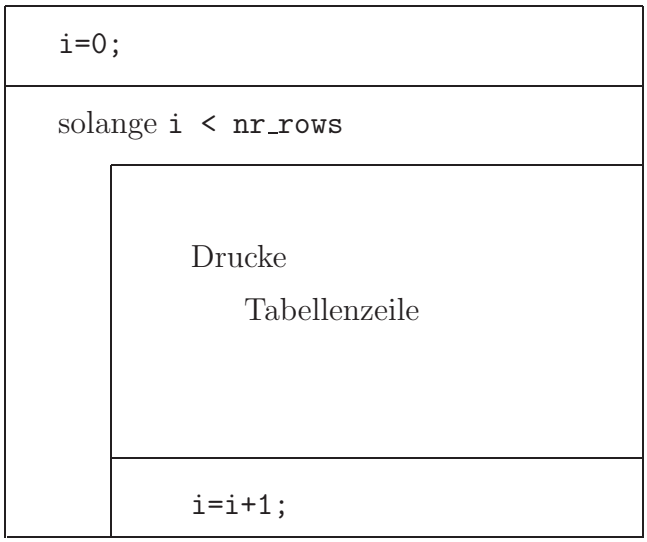
c) Vervollständigen Sie die globale Methode `main()` um interaktive Eingabe der Koeffizienten `A,B,C` inklusive von Eingabeaufforderungen (prompts).

d) Testen Sie `solve()` auch für Eingaben mit `A==0.0`.

C) Schleifen / Wiederholung

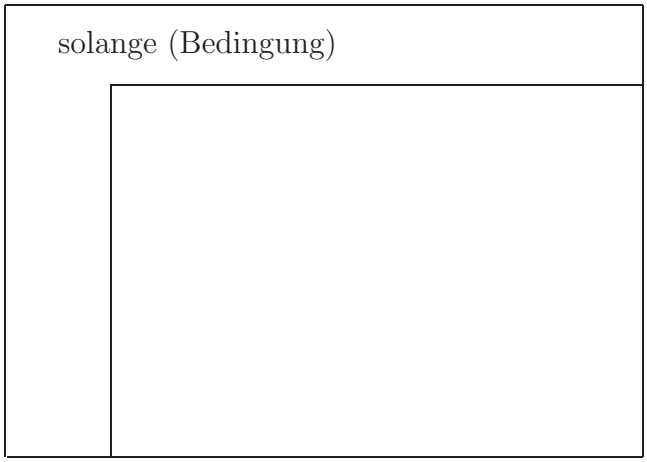
In C++ werden mehrere Schleifenkonzepte zur Verfügung gestellt, nämlich:

- Zählschleife




```
for (int i=0;
     i < nr_rows;
     i++ )
{ ... };

// oder

for (factorial=n, i=n-1;
     i>=1; i--)
     factorial *= i;
```

- abweisende Schleife




```
power = 1.0;
while (i>0) {
  power *= x;
  i--;
};
```

- annehmende Schleife


```
do {
  k = i % j;
  i = j;
  j = k;
} while (j != 0);
```

Schleife und leere Anweisung:

```
for (long i=0; i<100000000; i++); // leere Anweisung
```

Um Wartezeiten zu realisieren verwendet man besser

```
#include <unistd.h>

  ⋮

sleep(2);
```

unter `UNIX` bzw.

```
#include <windows.h>

  ⋮

Sleep(2000);
```

unter `Windows`.

`break` und `continue`:

Mittels `break` kann eine Schleife vorzeitig verlassen werden:

```
for (i=0; i<10; i++){
  cin >> x;
  if (x < 0.0) {
    cout << "Fertig" << endl;
    break;
  }
  cout << sqrt(x) << endl;
}
```

Mittels `continue` kann im Schleifenkörper vorzeitig ans Ende desselben gesprungen werden, um mit dem nächsten Iterationsschritt fortzufahren:

```
for (i=0; i<MAX; i++) {
    ⋮

    if (ist_fertig(c) )
      continue;

    ⋮
  // hierher von continue
}
```

Aufgaben:
Testen Sie die Beispiele dieses Kapitels, indem Sie sie in Funktionen

- `int ggt(int,int)`
- `double abs(double)`
- `double min(double,double)`
- `int factorial(int)`
- `double power(double, int)`
- `void sqrt_table()`

benutzen und diese in einem entsprechenden `main`-Testrahmen testen.

Auch ein `return` in einer Schleife kann diese vorzeitig beenden (es beendet das ganze (Unter-)Programm) :

```
double Partsum(const double M[], const int dim)
{
  double Sum(0.0);
  assert (dim > 0);
  for (int i=0; i<dim; i++)
    {
      if ( M[i] == 0.0 ) return Sum;
      Sum += M[i];
    };
  return Sum;
}
```

Das Programm berechnet und liefert die Summe der Komponenten von `M` bis zur ersten Komponente mit dem Wert `0.0`.

```
////////////////////////////////
// Datei:   power.cc
// Version: 1.0
// Autor:  Hans-Juergen Buhl
// Datum: 17.09.1998
////////////////////////////////

#include  <iostream>
#include  <iomanip>

using namespace std;

double power2(double x, int exp)
{
  double erg(1.0);

  if (exp < 0)
    throw "negativer Exponent bei power2 nicht erlaubt!";
  while ( exp > 0 ) {
    if ((exp % 2 ) != 0) {
      erg *= x;
      exp--;
    } else {
      x = x*x;
      exp = exp/2;
    }
  };
  return erg;
};
```

```
int main()
{
  cout << setprecision(10) << power2(13.5, 3) << endl;

  return 0;
}
```

Aufgaben:

a) Testen Sie obigen Algorithmus durch schrittweise Ausführung und Beobachtung der Werte von `erg`, `exp` und `x`.

b) Benutzen Sie eine Funktion

```
inline bool odd(const int n)
{
  return ( (n % 2) != 0 );
};
```

Das folgende Programm realisiert Funktionstabellen durch eine Funktion `void printtable(...)`:

```
////////////////////////////////
// Datei:   table.cc
// Version: 1.0
// Zweck:  for-Schleife, if
// Autor:  Hans-Juergen Buhl
// Datum: 20.09.1998
////////////////////////////////

#include <iostream>
#include <cmath>

using namespace std;

typedef double (*FKT)(double);

void printtable(const double start_x,
                const double delta_x,
                const int nr_rows,
                FKT f)
{
  double x(start_x);
  if (nr_rows > 0)
    for (int i = 0; i < nr_rows; i++) {
```

```
      cout << x << " " << f(x) << endl;
      x = x + delta_x;
    }
};

int main()
{
  printtable ( 0.0, 0.1, 5, sin);
}
```

Aufgaben:

a) Warum ist kein `assert(nr_rows>0)` nötig?

b) Vergleiche

- `x = x + delta_x;`
- `x = start_x + i * (end_x - start_ x) / (nr_rows - 1);`
- `x = ( (nr_rows-1-i) * start_x + i * end_x )`
  `/ (nr_rows-1);`

Die nun folgende Version realisiert Funktionstabellen als Objekte / Klassen:

```
////////////////////////////////
// Datei:   table2.cc
// Version: 2.0
// Zweck:  for-Schleife, if
// Autor:  Hans-Juergen Buhl
// Datum: 20.09.1998
////////////////////////////////

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

using namespace std;

typedef double (*FKT)(double);

class Table {

  double start_x;
  double delta_x;
  int nr_rows;

  FKT fkt;

public:

  Table(double start, double delta, int nr, FKT f): start_x(start),
                                                     delta_x(delta),
                                                     nr_rows(nr),
                                                     fkt(f)
    {};

  void print() const
    {
      double x(start_x);
      if (nr_rows > 0)
        for (int i = 0; i < nr_rows; i++) {
          cout << x << "   " << fkt(x) << endl;
          x = x + delta_x;
        }
    };
};

int main()
{
  Table SinTable(0.0, 0.1, 5, sin);
  SinTable.print();
}
```

Aufgaben:

a) Testen Sie beide Programme

b) Ergänzen Sie die Klasse `Tabelle` um Methoden

```
void set_start_x(double)
double get_start_x()
     ⋮
void set_fkt(FKT)
```

und testen Sie erneut.

D) Delegation:

- Verbundanweisung / Block:
  Blöcke erlauben es, Anweisungssequenzen zu benutzen, wo nur eine Anweisung erlaubt ist (`do-while`, `while`, `for` ...). Innerhalb von Blöcken konstruierte Klassenexemplare bzw. Elementardatentyp-Variablen existieren genau bis zum Blockende:

```
⋮
{ ...
   Matrix Hilfsmatrix;
   ...
   ...
}
⋮
```

- Funktionsaufruf:
  Funktionen wie `printtable()` erlauben es, `main` übersichtlich zu gestalten und insbesondere bei mehrfacher Benutzung desselben Algorithmusses mit jeweils anderen Parametern, jeweils nur leicht modifizierte Algorithmen zu vermeiden. Es ist jedoch immer zu überlegen, ob globale Funktionen nicht besser als Methoden von Objekten definiert werden sollten, insbesondere wenn die Funktionen auf viele Parameter oder auf globale Daten zugreifen würden.

- Methodenaktivierung
  Folgende Typen von Methoden sollten bedacht werden:
  - Konstruieren bzw. Zerstören von Objekt-Exemplaren
  - Bericht über den Wert/Status der Objekt-Attribute (Observatoren)
  - Ändern des Status der Objekt-Attribute (Modifikatoren)

Eine Klasse `Function`:

```
////////////////////////////////
// Datei:   Function.cc
// Version: 1.0
// Zweck:  lazy evaluation
// Autor:  Hans-Juergen Buhl
// Datum: 16.12.1998
////////////////////////////////

#include <iostream>
#include <string>
#include <cmath>
#include <cassert>

using namespace std;

typedef double (*FKT)(double);

class Function{

  double start_x, end_x;
  int nr_intervals;           // >0
  FKT fkt;

  bool last_int_value_valid;
  double last_int_value;    // valid iff (last_int_value_valid == true)

  bool last_slope_value_valid;
  double last_slope_value; // valid iff (last_slope_value_valid == true)

  // ...

public:


  // nicht moeglich:
  //     Function(double x_l = 0.0, double x_r = 2.0, int n = 8, FKT f) ...

  Function(FKT f, double x_l = 0.0, double x_r = 2.0, int n = 8):
    start_x(x_l), end_x(x_r), fkt(f),
    last_int_value_valid(false),
    last_slope_value_valid(false)
    {
      assert(n>0);
      nr_intervals = n;
    };

  void set_nr_intervals(const int n)
```

```
  {
    assert(n>0);
    nr_intervals = n;
    last_int_value_valid = false;
    last_slope_value_valid = false;
  };

int get_nr_intervals() const
  {
    return nr_intervals;
  };

double get_integral_value()
  {
    if (!last_int_value_valid)
      {
        // algorithm for numerical computation of integral value,
        // last_int_value = algorithm result;

        last_int_value_valid = true;
      };
    return last_int_value;
  };

double p(const double x)
  {
    return ((15.0 * x + 4.0) * x + 3.5) * x + 2.25;
  };

int main()
  {
    Function sin0_2pi(sin, 0.0, 2.0*M_PI, 16);
    // test usage of Function methods

    Function poly(p);
    // further tests ...

    return 0;

  }
```

Bemerkung:
Lazy Evaluation statt automatischer Neuberechnung bei jeder Attributsänderung ist vorteilhaft, da

- nicht alle abgeleiteten Attribute nach jeder Parameteränderung sondern nur bei Bedarf erneut berechnet werden.
- Zwischenergebnisse so lange wie gültig unverändert bleiben.

Funktionen ohne Rückgabewert (Prozeduren):

a) Beispiel:

```
#include <cstdlib>
#include <ctime>

void printDateTime()
{
  cout << time(0) << "Sekunden seit 1. Januar 1970 "
       << "0 Uhr GMT" << endl;
};
```

Benutzung: `printDateTime();`
(nicht in arithmetischen Ausdrücken)

b) Bei Aufruf von Fuktionen mit Rückgabewert kann man diesen Rückgabewert ignorieren:

```
#include <cstdlib>
#include <ctime>

time_t DateTime()
{
  time_t walltime(time(0));
  cout << walltime << endl;
  return walltime;
}

  ⋮

  int main(){
    time_t Start(DateTime());
      ⋮
    time_t Zeit(DateTime()−Start);
      ⋮
    DateTime();
      ⋮
}
```

Semantik der Funktionsparameter:

```
double power2(double x, int exp)
{
  double erg(1.0);

  if (exp < 0)
    throw "negativer Exponent bei power2 nicht erlaubt!";
  while ( exp > 0 ) {
    if ((exp % 2 ) != 0) {
      erg *= x;
      exp--;
    } else {
      x = x*x;
      exp = exp/2;
    }
  };
  return erg;
};

int main()
{
  double x(2.0);
  cout << power(x,3) << endl;
  cout << x << endl;
}
```

druckt

```
8.0
2.0
```

Modifikation der Funktionssignatur führt zu anderen Ergebnissen:

a) `double power2(const double x, const int exp)`

   gibt die Fehlermeldung:

   `The operand x cannot be assigned to.`

b) `double power2(double &x, int exp)`

   druckt

```
8.0
4.0
```

c) `double power2(double &x, int &exp)`

gibt die Fehlermeldung

`Warning : Anachronism; Temporary created for exp.`

Funktionen mit „mehreren“ Ergebniswerten:

a) Klassen-Rückgabewert

```
////////////////////////////////
// Datei:   Point.cc
// Version: 1.0
// Zweck:  mehrere Funktionsergebnisse
// Autor:  Hans-Juergen Buhl
// Datum: 17.09.1998
////////////////////////////////

#include <iostream>

using namespace std;

class Point{

  double x,y;

public:

  Point(double a = 0.0, double b = 0.0):
    x(a), y(b) {};

  void print()
    {
      cout << "( " << x << " , " << y << " )";
    };

  // ...
};

Point giveOrigin()      // returns two double values
{
  // ... an algorithm ...
  return Point(1.0, 2.5);
};

int main()
{
  Point x(2.0);
  x.print(); cout << endl;
  x = giveOrigin();
  x.print(); cout << endl;
  return 0;
}
```

b) Referenz-Parameter von Klassentyp:

```
////////////////////////////////
// Datei:   Point2.cc
// Version: 1.0
// Zweck:  mehrere Funktionsergebnisse
// Autor:  Hans-Juergen Buhl
// Datum: 17.09.1998
////////////////////////////////

#include <iostream>

using namespace std;

class Point{

  double x,y;

public:

  Point(double a = 0.0, double b = 0.0):
    x(a), y(b) {};

  void print()
    {
      cout << "( " << x << " , " << y << " )";
    };

  double get_x() const
    {
      return x;
    };

  double get_y() const
    {
      return y;
    };

  // ...

};

void mirrorPoint(Point& p)          // returns function parameter p
{
  p.print(); cout << endl;
  p = Point(-p.get_y(), p.get_x());
  p.print(); cout << endl;
};
```

```
int main()
{

  Point x(2.0, 3.0);
  x.print(); cout << endl;

  mirrorPoint(x);
  x.print(); cout << endl;

  return 0;
}
```

Matrizen noch nicht als Klassen:

```
//////////////////////////////////////
// Datei:   matrix.cc
// Version: 1.0
// Zweck:  primitive (non class) matrix
// Autor:  Hans-Juergen Buhl
// Datum: 17.09.1998
//////////////////////////////////////

#include <iostream>
#include <cstdlib>

using namespace std;

const int M(3);
const int N(2);

double Norm(const double B[][N], const int m)
{
  double accu(0.0);

  for (int i=0; i < m; i++)
    for (int j = 0; j < N; j++)
      accu += abs(B[i][j]);
  return accu;
};

int main()
{
  double A[M][N] = { 1, -2, 3, 4, -5, -6};
  cout << Norm(A, M) << endl;

  return 0;
}
```

Bemerkung:
Funktionen sind nicht verschachtelbar.
Klassen sind dies jedoch!

E) Test und Abbruch:

Neben

`break` (`switch` und Schleifen),
`continue` (Schleifen),
`return` (in `main` oder Funktionen)
`exit`

kann der Algorithmen-Kontrollfluß in Ausnahmesituationen auch beeinflußt werden durch

- `assert`

  ```
  #include <cassert>

  ...

  double power2(double x, int exp)
  {
    double erg(1.0);
    assert( exp >= 0 ); // Vorbedingung

    ...

  }
  ```

  Zusicherung : „`exp >= 0`“ wird ausgewertet und bei Evaluierung zu `false` wird eine Fehlermeldung ausgegeben sowie das Programm terminiert.

- `Exceptions`

  ```
  ...

  double power2(double x,int exp)
  {
    double erg(1.0);
    if ( exp < 0 ) throw(exp);

    ...

  }
  ```

Falls „`exp<0`“ wird je nach System eine Meldung ähnlich

```
Runtime exception error:
Current exception: int
Abort (core dumped) No handler for exception
```

ausgegeben und das Programm terminiert.

Wenn ein System nur

```
Abbrechen (Speicherabbild geschrieben)
```

meldet, kann mittels des Debuggers `dbx` genaueres erfragt werden:

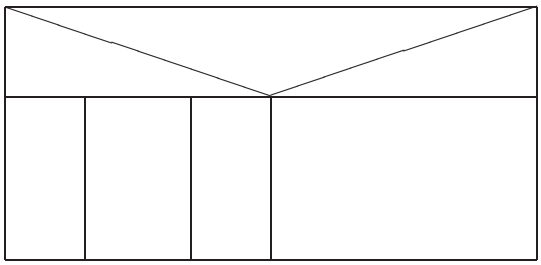
```
% adb power core
Reading power
core file header read successfully
Reading ld.so.1
...
program terminated by signal ABRT (Abbrechen)
Current function is power2
   19           throw "negativer Exponent bei power2 nicht erlaubt!";
(dbx) exit
```

Schöner ist es jedoch, wenn man die Exception am Ende von `main` abfängt und selbst eine Fehlermeldung erzeugen kann:

```
////////////////////////////////
// Datei:   power1.cc
// Version: 1.1
// Zweck:  while-Schleife, catch
// Autor:  Hans-Juergen Buhl
// Datum: 17.09.1998
////////////////////////////////

#include      <iostream>
#include      <iomanip>

using namespace std;

double power2(double x, int exp)
{
  double erg(1.0);

  if (exp < 0)
```

```
        throw "negativer Exponent bei power2 nicht erlaubt!";
  while ( exp > 0 ) {
        if ((exp % 2 ) != 0) {
                erg *= x;
                exp--;
        } else {                // hier ist exp gerade
                x = x*x;
                exp = exp/2;
        }
  };
  return erg;
};

int main()
{
  try{
        cout << setprecision(10) << power2(0.5, -2) << endl;
        return 0;
  } catch (const char* err){
        cerr << endl << "Fehler: "<< err << endl << endl;
        return 1;
  }
}
```

Die vorgestellten Strukturblöcke können ineinander eingesetzt werden (beliebige Verschachtelung), sofern die Sequenz der Abarbeitung klar bleibt, weshalb zum Beispiel



nicht erlaubt ist!

Der Ausgabeoperator << für eigene Klassen:

```
////////////////////////////////
// Datei:   comp4.cc
// Version: 4.0
// Zweck:  << Operator
// Autor:  Hans-Juergen Buhl
// Datum: 17.09.1998
////////////////////////////////

#include <iostream>

using namespace std;

class comp {

  double re, im;

public:

  comp( const double& a = 0, const double& b = 0 ) : re(a), im(b) {};

  friend comp operator+ (const comp&, const comp&);

  void print() const { cout << re << " + i * " << im; };

  friend ostream& operator<<(ostream& os, const comp z);

};

comp operator+ (const comp& c1, const comp& c2)
{
  comp Erg(c1.re + c2.re, c1.im + c2.im);
  return Erg;
};

ostream& operator<<(ostream& os, const comp z)
{
  if ((z.re != 0.0) || (z.im == 0.0))
    os << z.re;
  if (z.im < 0.0)
    os << " - " << -z.im << " * i";
  else if (z.im > 0.0){
    if (z.re == 0.0)
      os << z.im << " * i";
    else
      os << " + " << z.im << " * i";
  };
  return os;
```

```
};

int main()
{
  comp x(1.1, 1.2);
  comp x2(2.5, 1.5);

  x2.print(); cout << endl;

  x2 = 1 + x;
  cout << x2 << endl << 1 + x << endl << 1 + comp(1.1, 1.2) << endl;

  cout << comp() << endl << comp(1.1) << endl << comp(0.0, 1.1)
       << endl << comp(1.1, 1.2) << endl;
  cout << comp(−1.1) << endl << comp(0.0, −1.1)
       << endl << comp(−1.1, −1.2) << endl;
  cout << comp(−1.0, 1.2) << endl << comp(1.1, −1.3) << endl;

  return 0;
}
```

1.3 Elementare Datentypen

Die skalaren Datentypen

Keine Klassen sind die Datentypen:

A) `bool`

B) `short`, `int`, `long`

C) `float`, `double`, `long double`

D) `char`

E) `enum`

Wir beschreiben sie trotzdem ähnlich wie Klassen durch

- ihre Literale
- „Konstruktoren“ = Variablendeklaration und
  Regeln für (autom.) Typkonversion
- Funktionen und Operatoren

Dabei werden die Operatoren `<<` und `>>` nicht explizit aufgeführt.

A) bool

Literale	`true,false`
Variablendeklaration	`bool success;` `bool trial = ( a == b)`

Operatoren	`!  ==  !=`
	`&&  and`
	`||  or`
	`b ? a1 : a2`
	`new, delete, &`

Beispiele:

// Beispiel a)

```
max = (a>b) ? a : b;
```

// Beispiel b)

```
if ((b != 0) && (a/b)>0) cout << "vorzeichengleich";
  else cout << "b==0 oder Vorzeichenwechsel";
```

// Beispiel c)

```
if ((j<dim) && v[j]==0.0) // ...
  else cout << "Indexfehler";
```

// Beispiel d)

```
if ((j>dim) || v[j]<0.0) cout << "Indexfehler oder v[j] negativ";
  else cout << "v[j] >= 0.0";
```

B) short, int, long

Häufig gilt:

16 Bit lang	32 Bit lang	64 Bit lang
`short`	`int`	`long`
`signed short`	`signed int`	`signed long`
`unsigned short`	`unsigned int`	`unsigned long`

Literale

0	1234	976	...	(dezimal)
00	02	011	...	(oktal)
0x0	0x2	0x11	...	(sedezimal)

`12L`	`long`
`12U`	`unsigned`

Deklaration:

```
int i;          ← Startwert undefiniert!
long j(15);
⋮
```

Wie das Beispiel der Funktion

```
long Fakultaet(long l){
  assert (l >= 0);
  long result, i;
  if ( l == 0 ) return 1;
  for (result=l, i=l-1; i >= 1; i--)
     result *= i;
  return result;
}
```

zeigt, sind die Ergebnisse von Integer-Arithmetikoperationen nicht immer richtig:

```
 0                          1
 1                          1
 2                          2
 3                          6
 4                         24
 5                        120
 6                        720
 7                       5040
 8                      40320
 9                     362880
10                    3628800
11                   39916800
12                  479001600
13                 1932053504
14                 1278945280
15                 2004310016
16                 2004189184
17                 -288522240
...                       ...
```

Ab dem Argumentwert 13 stimmen die Ergebnisse nicht mehr, es wird jedoch keinerlei Warnung oder Fehler angezeigt.

Alle elementaren Arithmetikoperatoren liefern nur richtige Ergebnisse, wenn das mathematisch korrekte Ergebnis im darstellbaren Bereich des entsprechenden Integertyps liegt, bei `LONG` also zwischen `LONG_MIN` und `LONG_MAX` (bei uns also - vgl. `climits`:

```
...

#define SHRT_MIN        (-32768)        /* min value of a "short int" */
#define SHRT_MAX        32767           /* max value of a "short int" */

#define INT_MIN         (-2147483647-1) /* min value of an "int" */
#define INT_MAX         2147483647      /* max value of an "int" */

#define LONG_MIN        (-2147483647L-1L)
                                        /* min value of a "long int" */
#define LONG_MAX        2147483647L     /* max value of a "long int" */


...
```

) zwischen -2147483648 und 2147483647. In allen anderen Fällen wird ein implementierungsabhängiger anderer Ergebniswert ohne jede Warnung geliefert.

Wollen Sie also zuverlässige Programme mit Integer-Arithmetik programmieren, so sollten Sie entweder wissen, daß jede einzelne Arithmetikoperation bei allen möglichen Programmläufen den entsprechenden Wertebereich nicht verläßt oder vor jeder Operation sicherstellen, daß jede solche Überschreitung eine Exception auslöst:

Seien also $a,\ b \in \mathbb{Z} \cap \{LONG\_MIN, LONG\_MIN+1, ..., LONG\_MAX\} =: long$ und — wie heute in allen üblichen Computersystemen — entweder $|LONG\_MIN| = LONG\_MAX$ oder $|LONG\_MIN| = LONG\_MAX + 1$.

0) Unärer Operator - (Vorzeichenwechsel)

$$-a \notin long \ \Leftrightarrow \ a < \underbrace{-LONG\_MAX}_{\in\ long}$$

1) Binärer Operator +

$$\begin{aligned} & a + b \notin long \\ & \qquad \Leftrightarrow \quad a + b > LONG\_MAX \ \vee \ a + b < LONG\_MIN \\ & \qquad \Leftrightarrow \qquad (a > 0 \wedge b > 0 \wedge a > \underbrace{LONG\_MAX - b}_{\in\ long}) \qquad \vee \\ & \qquad\qquad\qquad (a < 0 \wedge b < 0 \wedge a < \underbrace{LONG\_MIN - b}_{\in\ long}) \end{aligned}$$

2) Binärer Operator -

$$\begin{aligned} & a - b \notin long \\ & \qquad \Leftrightarrow \quad a - b > LONG\_MAX \ \vee \ a - b < LONG\_MIN \\ & \qquad \Leftrightarrow \qquad (a > 0 \wedge b < 0 \wedge a > \underbrace{LONG\_MAX + b}_{\in\ long}) \qquad \vee \\ & \qquad\qquad\qquad (a < 0 \wedge b > 0 \wedge a < \underbrace{LONG\_MIN + b}_{\in\ long}) \end{aligned}$$

3) Binärer Operator /

$$
\begin{aligned}
& a/b \notin long \\
\Leftrightarrow \quad & (b = 0) \qquad \vee \\
& (b = -1 \wedge a < \underbrace{-LONG\_MAX}_{\in\, long})
\end{aligned}
$$

4) Binärer Operator *

$$
\begin{aligned}
& a * b \notin long \\
\Leftrightarrow \quad & (a > 0 \wedge b > 0 \wedge a > \underbrace{LONG\_MAX/b}_{\in\, long}) \quad \vee \\
& (a < 0 \wedge b < 0 \wedge a < \underbrace{LONG\_MAX/b}_{\in\, long}) \quad \vee \\
& (a < 0 \wedge b > 0 \wedge a < \underbrace{LONG\_MIN/b}_{\in\, long}) \quad \vee \\
& (a > 0 \wedge b < 0 \wedge a > \underbrace{LONG\_MIN/b}_{\in\, long})
\end{aligned}
$$

Funktionen / Operatoren:

`i++`	`i--`	erst Wert berechnen, dann `i` inkrementieren / dekrementieren
`++i`	`--i`	erst `i` inkrementieren / dekrementieren, dann Wert berechnen
`+i`	`-i`	
`*`	`/`	`%`
`+`	`-`	
`<`	`<=`	
`>`	`>=`	
`==`	`!=`	
`=`		
`+=`	`-=`	
`*=`	`/=`	`%=`

Die Operatoren sind nach absteigender Priorität angeordnet.

Weitere (bitorientiere) Operatoren :

`~`		Einerkomplement
`<<`	`>>`	links bzw. rechts shift
`&`		und bitweise
↑		exklusives oder bitweise
`|`		oder bitweise
`>>=`	`<<=`	
`&=`	↑=	
`|=`		

Operatoren mit Nebeneffekt (`++`, `--`, `=`, ...) dürfen für jede Variable / jedes Klassenexemplar in einem Ausdruck höchstens einmal benutzt werden.

Beispiele:

```
(a++) * 2 + (a--)
```

ist undefiniert. Selbst

```
a = a++
```

ist undefiniert, da `=` einen Nebeneffekt auf `a` hat.

C) float, double, long double

Häufig gilt:

32 Bit lang	64 Bit lang	≥ 64, z.B. 128
`float`	`double`	`long double`

Literale:

1.23	.23	0.23	1.
1.2e10	1.2e-99		
3.14159265f	2.0f	4.1F	
0.3333L	0.3l		

Deklaration:

```
float x;
double y(1.0);
```

Funktionen / Operatoren

```
+x  -x
*   /
+   -
<   <=  >   >=
==  !=
=   +=  -=  *=  /=
```

```
<cmath>
```

Konstanten:

```
M_E
⋮
M_SQRT1_2
MAXFLOAT
```

Funktionen

acos
asin
atan
atan2
cos
sin
tan
cosh
sinh
tanh
exp
frexp -- berechne x$\in [0.5, 1)$ und e des Wertes x*pow(2,e)
ldexp -- berechne d*pow(2,i)
log
log10
modf
pow
sqrt
ceil
fabs
floor
erf
erfc
gamma
$\vdots$

D) char

`char`	meist ISO-8859, 8 Bit mit $-128 \ldots 127$ oder $0 \ldots 255$
`signed char`	mit $-128...127$
`unsigned char`	mit $0 \ldots 255$

Literale: 'a', 'ä', '\n'

Operatoren:
`char` ist ein `integer`-Typ, weshalb die Operatoren aus B) auch hier gültig sind (inkl. bitweiser Logik-Operatoren) !

`wchar_t` z.B. für Unicode (16 Bit / 32 Bit)

Literale:
'\u1200', '\u1a2b', '\U1a2b3a4f', ...

E) enum (Aufzählungstypen)

= definierte Sammlungen benannter ganzzahliger Konstanten

Hier einige typische Beispiele:

enum {off, on}

enum color {red, blue, white, green}

enum {BOTTOM=80, TOP=100, OVER}

Bemerkung:

A) Für `switch` gut geeignet.

B) Für Bitfelder geeignet.

C) `enum` in arithmetischen Ausdrücken werden in `int` konvertiert.

D) Es können (wie bei Klassen gesehen) auch eigene Operatoren `++`, `<<` und ähnliche definiert werden!

1.3.1 Codierung von Zeichen, Dokumenten und Seiten

1.3.1.1 Zeichen und Schrift

Zeichensätze Der ASCII-Code (*American Standard Code for Information Interchange*) ist auch heute noch Grundlage vieler Zeichencodes:

00	NUL	01	SOH	02	STX	03	ETX	04	EOT	05	ENQ	06	ACK	07	BEL
08	BS	09	HT	0A	NL	0B	VT	0C	NP	0D	CR	0E	SO	0F	SI
10	DLE	11	DC1	12	DC2	13	DC3	14	DC4	15	NAK	16	SYN	17	ETB
18	CAN	19	EM	1A	SUB	1B	ESC	1C	FS	1D	GS	1E	RS	1F	US
20	SP	21	!	22	"	23	#	24	$	25	%	26	&	27	´
28	(	29	)	2A	*	2B	+	2C	,	2D	-	2E	.	2F	/
30	0	31	1	32	2	33	3	34	4	35	5	36	6	37	7
38	8	39	9	3A	:	3B	;	3C	<	3D	=	3E	>	3F	?
40	@	41	A	42	B	43	C	44	D	45	E	46	F	47	G
48	H	49	I	4A	J	4B	K	4C	L	4D	M	4E	N	4F	O
50	P	51	Q	52	R	53	S	54	T	55	U	56	V	57	W
58	X	59	Y	5A	Z	5B	[	5C	\	5D	]	5E	^	5F	_
60	`	61	a	62	b	63	c	64	d	65	e	66	f	67	g
68	h	69	i	6A	j	6B	k	6C	l	6D	m	6E	n	6F	o
70	p	71	q	72	r	73	s	74	t	75	u	76	v	77	w
78	x	79	y	7A	z	7B	{	7C	|	7D	}	7E	~	7F	DEL

Tabelle 1.5: ASCII-Code

Da sieben Bit, also 128 Zeichen nicht für landesspezifische Sonderzeichen ausreichten, entstanden die landesspezifischen Varianten durch Zeichenersetzung (vgl. Tabelle 1.6).

ISO Nr.	Zeichensatz	Dezimalform												
		ID	35	36	64	91	92	93	94	96	123	124	125	126
6	ANSI ASCII	0U	#	$	@	[	\	]	^	`	{	|	}	~
11	Schweden: Namen	0S	#		É	Ä	Ö	Å	Ü	`	ä	ö	å	ü
10	Schweden	3S	#		@	Ä	Ö	Å	^	`	ä	ö	å	
17	Spanien	2S	£	$	§	¡	Ñ	¿	^	`	°	ñ	ç	~
69	Frankreich	1F	£	$	à	°	ç	§	^	μ	é	ù	è	
21	Deutschland	1G	#	$	§	Ä	Ö	Ü	^	`	ä	ö	ü	ß
4	Großbritannien	1E	£	$	@	[	\	]	^	`	{	|	}	
16	Portugal	4S	#	$	§	Ã	Ç	Õ	^	`	ã	ç	õ	°
60	Norwegen 1	0D	#	$	@	Æ	Ø	Å	^	`	æ	ø	å	
61	Norwegen 2	1D	§	$	@	Æ	Ø	Å	^	`	æ	ø	å	|
2	IRV		#		@	[	\	]	^	`	{	|	}	
15	Italien	0I	£	$	§	°	ç	é	^	`	à	ò	è	ì

Tabelle 1.6: ISO-Austauschtabelle

Alternativ wurden acht Bit (256 Zeichen) für landesspezifische Sonderzeichen, mathematische Symbole, graphische Symbole zum Tabellendruck bzw. für Sonderzwecke (Spiele, ...) besetzt, etwa im Industriestandard PC-8 Zeichensatz (Tabelle 1.7)[1].

0	16	32	0	@	P	‘	p	Ç	É	á				α	$\equiv$
			48	64	80	96	112	128	144	160	176	192	208	224	240
1	17	!	1	A	Q	a	q	ü	æ	í				ß	$\pm$
		33	49	65	81	97	113	129	145	161	177	193	209	225	241
2	$\updownarrow$	"	2	B	R	b	r	é	Æ	ó		$\top$		Γ	$\geq$
	18	34	50	66	81	98	114	130	146	162	178	194	210	226	242
3	!!	#	3	C	S	c	s	â	ô	ú	|	$\vdash$		π	$\leq$
	19	35	51	67	82	99	115	131	147	163	179	195	211	227	243
4	¶	$	4	D	T	d	t	ä	ö	ñ	$\dashv$			$\sum$	
	20	36	52	68	83	100	116	132	148	164	180	196	212	228	244
♣	§	%	5	E	U	e	u	à	ò	Ñ				σ	
5	21	37	53	69	84	101	117	133	149	165	181	197	213	229	245
♠		&	6	F	V	f	v	å	û	^{a}				μ	$\div$
6	22	38	54	70	85	102	118	134	150	166	182	198	214	230	246
		’	7	G	W	g	w	ç	ù	^{o}				τ	$\approx$
7	23	39	55	71	86	103	119	135	151	167	183	199	215	231	247
	$\uparrow$	(	8	H	X	h	x	ê	ÿ	¿				Φ	$^\circ$
8	24	40	56	72	87	104	120	136	152	168	184	200	216	232	248
◯	$\downarrow$	)	9	I	Y	i	y	ë	Ö					Θ	$\cdot$
9	25	41	57	73	88	105	121	137	153	169	185	201	217	233	249
	$\rightarrow$	*	:	J	Z	j	z	è	Ü	¬	$\|\|$			Ω	$\cdot$
10	26	42	58	74	89	106	122	138	154	170	186	202	218	234	250
	$\leftarrow$	+	;	K	[	k	{	ï	ç					δ	$\surd$
11	27	43	59	75	90	107	123	139	155	171	187	203	219	235	251
		,	<	L	\	l	|	î	£					∞	^{n}
12	28	44	60	76	91	108	124	140	156	172	188	204	220	236	252
	$\leftrightarrow$	-	=	M	]	m	}	ì		¡				ϕ	^{2}
13	29	45	61	77	92	109	125	141	157	173	189	205	221	237	253
		.	>	N	^	n	~	Ä	Pt	$\ll$				ϵ	
14	30	46	62	78	93	110	126	142	158	174	190	206	222	238	254
		/	?	O	_	o		Å	f	$\gg$				$\cap$	
15	31	47	63	79	94	111	127	143	159	175	191	207	223	239	255

Tabelle 1.7: PC-8 Zeichensatz

[1] In dieser – wie auch in einigen der folgenden Tabellen – sind leere Codestellen entweder unbesetzt oder wegen Problemen beim Satz der entsprechenden Zeichen in diesem Skript freigelassen worden.

In Windows 3.x wurden jedoch andere Codierungen genutzt (Tabelle 1.8). Der inzwischen verabschiedete Standard der *International Standardization*

NUL 0	 16	 32	0 48	@ 64	P 80	` 96	p 112	 128	 144	 160	° 176	À 192	 208	à 224	 240
 1	 17	! 33	1 49	A 65	Q 81	a 97	q 113	 129	‘ 145	¡ 161	± 177	Á 193	Ñ 209	á 225	ñ 241
 2	 18	" 34	2 50	B 66	R 81	b 98	r 114	 130	’ 146	ç 162	2 178	Â 194	Ò 210	â 226	ò 242
 3	 19	# 35	3 51	C 67	S 82	c 99	s 115	 131	 147	£ 163	3 179	Ã 195	Ó 211	ã 227	ó 243
 4	 20	$ 36	4 52	D 68	T 83	d 100	t 116	 132	 148	 164	´ 180	Ä 196	Ô 212	ä 228	ô 244
 5	 21	% 37	5 53	E 69	U 84	e 101	u 117	 133	 149	 165	μ 181	Å 197	Õ 213	å 229	õ 245
 6	 22	& 38	6 54	F 70	V 85	f 102	v 118	 134	 150	| 166	¶ 182	Æ 198	Ö 214	æ 230	ö 246
BEL 7	 23	′ 39	7 55	G 71	W 86	g 103	w 119	 135	 151	§ 167	· 183	Ç 199	× 215	ç 231	÷ 247
BS 8	 24	(40	8 56	H 72	X 87	h 104	x 120	 136	 152	¨ 168	 184	È 200	Ø 216	è 232	ø 248
HT 9	 25	) 41	9 57	I 73	Y 88	i 105	y 121	 137	 153	© 169	1 185	É 201	Ù 217	é 233	ù 249
LF 10	 26	* 42	: 58	J 74	Z 89	j 106	z 122	 138	 154	a 170	o 186	Ê 202	Ú 218	ê 234	ú 250
VT 11	ESC 27	+ 43	; 59	K 75	[90	k 107	{ 123	 139	 155	≪ 171	≫ 187	Ë 203	Û 219	ë 235	û 251
FF 12	 28	, 44	< 60	L 76	\ 91	l 108	| 124	 140	 156	¬ 172	$\frac{1}{4}$ 188	Ì 204	Ü 220	ì 236	ü 252
CR 13	 29	- 45	= 61	M 77	] 92	m 109	} 125	 141	 157	- 173	$\frac{1}{2}$ 189	Í 205	Ý 221	í 237	ý 253
SO 14	 30	. 46	> 62	N 78	^ 93	n 110	~ 126	 142	 158	 174	$\frac{3}{4}$ 190	Î 206	 222	î 238	 254
SI 15	 31	/ 47	? 63	O 79	_ 94	o 111	 127	 143	 159	¯ 175	¿ 191	Ï 207	ß 223	ï 239	ÿ 255

Tabelle 1.8: Zeichensatz für Windows 3.x

Organisation (ISO), der ISO-8859 Latin 1(ECMA-94 Latin 1) Zeichensatz (Tabelle 1.9) setzte sich insbesondere bei Workstations und neueren Hard- und Softwareprodukten durch. Neben der Latin-1 Version existieren auch noch einige andere nationale Sonderformen des ISO 8859 Codes (vgl. Tabelle 1.10).

			0	@	P	‘	p				°	À		à	
0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
		!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ
1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241
		"	2	B	R	b	r				^{2}	Â	Ò	â	ò
2	18	34	50	66	81	98	114	130	146	162	178	194	210	226	242
		#	3	C	S	c	s			£	^{3}	Ã	Ó	ã	ó
3	19	35	51	67	82	99	115	131	147	163	179	195	211	227	243
		$	4	D	T	d	t				´	Ä	Ô	ä	ô
4	20	36	52	68	83	100	116	132	148	164	180	196	212	228	244
		%	5	E	U	e	u				μ	Å	Õ	å	õ
5	21	37	53	69	84	101	117	133	149	165	181	197	213	229	245
		&	6	F	V	f	v			|	¶	Æ	Ö	æ	ö
6	22	38	54	70	85	102	118	134	150	166	182	198	214	230	246
		′	7	G	W	g	w			§	·	Ç	×	ç	÷
7	23	39	55	71	86	103	119	135	151	167	183	199	215	231	247
		(	8	H	X	h	x			¨		È	Ø	è	ø
8	24	40	56	72	87	104	120	136	152	168	184	200	216	232	248
		)	9	I	Y	i	y			©	^{1}	É	Ù	é	ù
9	25	41	57	73	88	105	121	137	153	169	185	201	217	233	249
		*	:	J	Z	j	z			^{a}	^{o}	Ê	Ú	ê	ú
10	26	42	58	74	89	106	122	138	154	170	186	202	218	234	250
		+	;	K	[	k	{			≪	≫	Ë	Û	ë	û
11	27	43	59	75	90	107	123	139	155	171	187	203	219	235	251
		,	<	L	\	l	|			¬	$\frac{1}{4}$	Ì	Ü	ì	ü
12	28	44	60	76	91	108	124	140	156	172	188	204	220	236	252
		-	=	M	]	m	}			-	$\frac{1}{2}$	Í	Ý	í	ý
13	29	45	61	77	92	109	125	141	157	173	189	205	221	237	253
		.	>	N	^	n	~				$\frac{3}{4}$	Î		î	
14	30	46	62	78	93	110	126	142	158	174	190	206	222	238	254
		/	?	O	_	o				¯	¿	Ï	ß	ï	ÿ
15	31	47	63	79	94	111	127	143	159	175	191	207	223	239	255

Tabelle 1.9: ISO-8859 Latin 1(ECMA-94 Latin 1) Zeichensatz

Code Set	Name	Coverage	Approved
8859-1	Latin-1	western Europe	15 February 1987
8859-2	Latin-2	eastern Europe	15 February 1987
8859-3	Latin-3	Maltese, Catalan, Galician, Esperanto	15 April 1988
8859-4	Latin-4	Baltic and Nordic region	15 April 1988
8859-5	Cyrillic	Slavic countries	1 December 1988
8859-6	Arabic	Arab countries	15 August 1987
8859-7	Greek	Greece	15 November 1987
8859-8	Hebrew	Israel	1 June 1988
8859-9	Latin-5	8859-1 minus Iceland plus Turkey	15 May 1989

Tabelle 1.10: nationale ISO8859-Varianten

Da die Tastaturen der Bildschirmarbeitsplätze nicht für alle möglichen Zeichen eigene Tasten besitzen (Abbildung 1.6), wird es ermöglicht, durch sogenannte Compose-Tastensequenzen die fehlenden Zeichen zu erzeugen (Tabelle 1.11).

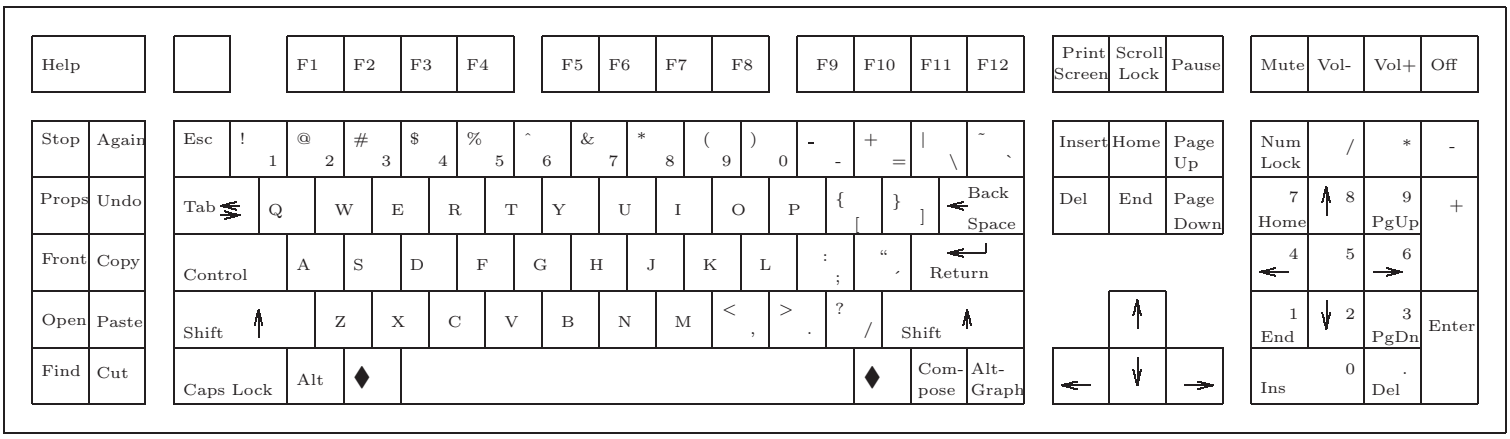


Abbildung 1.6: SunKeyBoard Type 5

Auf PC's existiert daneben noch die Möglichkeit durch gleichzeitiges Drücken der Modifikationstasten `AltGraph`, `Shift`, `AltGraph+Shift` eine Vierfachbelegung einiger Tasten zu realisieren, etwa `AltGraph+Q` für @: 

Damit immer noch nicht erreichbare Zeichen können durch ihre Dezimalcodes unter Benutzung der Alt-Taste erzeugt werden: `<Alt><2><3><6>` (die Alt-Taste muß dabei während der Zifferneingabe niedergedrückt bleiben und die Ziffern müssen im Zifferneingabeblock bei eingeschaltetem Num-Modus betätigt werden).

Zeichen	Compose-Tastenfolge		Zeichen	Compose-Tastenfolge
Á	A ’		Í	I ’
á	a ’		í	i ’
À	A ‘		Î	I ^
à	a ‘		î	i ^
Â	A ^		Ï	I "
â	a ^		ï	i "
Ã	A ~		Ñ	N ~
ã	a ~		ñ	n ~
Ä	A "		Ò	O ‘
ä	a "		ò	o ‘
Å	A *		Ó	O ’
å	a *		ó	o ’
Æ	A E		Ô	O ^
æ	a e		ô	o ^
Ç	C ,		Õ	O ~
ç	c ,		õ	o ~
	d -		Ö	O "
	D -		ö	o "
È	E ‘		Ø	O /
è	e ‘	⋮	ø	o /

Tabelle 1.11: Eingabe von Sonderzeichen / Tastaturen: Compose-Tastensequenzen

1.3.1.2 Dokumentbeschreibungssprachen

In Textverarbeitungssystemen werden Sonderzeichen häufig als internes Speicherformat in Zeichenfolgen codiert, z.B. in SGML, Tabelle 1.12.

HTML = *hypertext markup language* als ein spezieller SGML-Typ sieht etwa wie in Abbildung 1.7 gezeigt aus. Nähere Informationen zu SGML findet der

```
<!DOCTYPE HTML SYSTEM "html.dtd">
<HTML>
<HEAD>
<TITLE> Kommentierte Vorlesungsverzeichnisse FB7 (University of Wuppertal, BUGHW) </TITLE>
</HEAD>
<BODY>
<H2><A
HREF="http://wmwap1.math.uni-wuppertal.de/pub/Mosaic/Mathematics_WWW.html">Fachbereich
Mathematik</A></H2>
<H2>
<IMG ALIGN="top" SRC ="http://wmwap1.math.uni-wuppertal.de/pub/Mosaic/Loewe.xbm">
<A HREF="http://www.uni-wuppertal.de/">Bergische-Universit&auml;t - Gesamthochschule
Wuppertal</A></H2>
<P>
</P>
<HR>
<H1> Kommentierte Vorlesungsverzeichnisse des Fachbereichs Mathematik der Bergischen
Universit&auml;t - Gesamthochschule Wuppertal</H1>
<UL><LI><A HREF="KVV-WS9495.html">Wintersemester 1994/95</A></LI>
<LI>Sommersemester 1995</LI>
<LI><A HREF="kvvws9596.html">Wintersemester 1995/96</A></LI>
<LI><A HREF="kvvss96.html">Sommersemester 1996</A></LI>
</UL>
<P></P>
<HR>
<P></P>
<IMG SRC="http://wmwap1.math.uni-wuppertal.de/pub/Mosaic/hjb.gif"
ALIGN="left">
<P><TT>Wed Feb 7 11:47:35 MET 1996</TT></P>
<ADDRESS>Hans-J&uuml;rgen Buhl (<A HREF="mailto: webmaster@math.uni-wuppertal.de">
webmaster@math.uni-wuppertal.de </A>)</ADDRESS>
<P><A HREF="http://www.webtechs.com/html-val-svc">
<IMG SRC="http://wmwap1.math.uni-wuppertal.de/pub/Mosaic/valid_html.3.0.gif" ALT="HTML 3.0
(Beta) Checked!"></A></P>
</BODY></HTML>
```

Abbildung 1.7: HTML-Beispiel

Leser im Internet etwa unter

http://www.oasis-open.org/cover/general.html

oder

http://www.w3.org/MarkUp/SGML/.

á	`&aacute;`	Small a with acute accent
Á	`&Aacute;`	Capital A with acute accent
ă	`&abreve;`	Small a with breve accent
Ă	`&Abreve;`	Capital A with breve accent
â	`&acirc;`	Small a with circumflex accent
Â	`&Acirc;`	Capital A with circumflex accent
æ	`&aelig;`	Small ae diphtong (ligature)
Æ	`&AElig;`	Capital AE diphtong (ligature)
à	`&agrave;`	Small a with grave accent
À	`&Agrave;`	Capital A with grave accent
ā	`&amacr;`	Small a with macron accent
Ā	`&Amacr;`	Capital A with macron accent
å	`&aring;`	Small a with ring accent
Å	`&Aring;`	Capital A with ring accent
ã	`&atilde;`	Small a with tilde accent
Ã	`&Atilde;`	Capital A with tilde accent
ä	`&auml;`	Small a with umlaut (diaeresis) accent
Ä	`&Auml;`	Capital A with umlaut (diaeresis) accent
č	`&ccaron;`	Small c with caron accent
Č	`&Ccaron;`	Capital C with caron accent
	⋮	
\	`&bsol;`	Backslash (reverse solidus)
$\surd$	`&radic;`	Radical sign
$\coprod$	`&coprod;`	Coproduct (amalgamation) operator
$\prod$	`&prod;`	Product operator
$\int$	`&int;`	Integral operator
$\oint$	`&coint;`	Integral operator with central o
$\sum$	`&sum;`	Summation operator
∇	`&nabla;`	Hamilton operator
⊔	`&sqcup;`	Square union sign
⊓	`&sqcap;`	Square intersection sign
⊑	`&sqsube;`	Square subset of, or equal to, sign
⊒	`&sqsupe;`	Square superset of, or equal to, sign
§	`&sect;`	Section sign
†	`&dag;`	Dagger
‡	`&Dag;`	Double dagger
¶	`&para;`	Paragraph sign (pilcrow)

Tabelle 1.12: SGML = *standard generalized Markup Language*

Eine weit über die Möglichkeiten von HTML hinausgehende SGML-Variante, die im Internet eine gute Zukunft haben dürfte, ist XML (extended markup language, vgl. `http://www.w3.org/XML/notes.html` und `http://www.w3.org/XML/simple-XML.html`). Insbesondere die Datenbankkopplung wird durch XML erleichtert. Schließlich steht in der XML-Anwendung MathML (vgl. `http://www.w3.org/Press/1998/MathML-REC`) endlich ein nutzbares Format für mathematisch-naturwissenschaftliche Veröffentlichungen im WWW zur Verfügung:
Die Formeln

$$\nabla \cdot E = \frac{\rho}{\epsilon_0} \qquad \nabla \times E = -\frac{\partial B}{\partial t} \qquad c^2 \nabla \times B = \frac{\partial E}{\partial t} + \frac{j}{\epsilon_0} \qquad \nabla \cdot B = 0$$

werden in MathML etwa folgendermaßen dargestellt:

```
<mrow>
  <mi fontstyle="normal">&dtri;</mi>
  <mo>&middot;</mo>
  <mi fontstyle="normal">E</mi>
</mrow>
<mo>=</mo>
<mfrac>
  <mi>&rho;</mi>
  <msub>
    <mi>&epsiv;</mi>
    <mn>0</mn>
  </msub>
</mfrac>
<mrow>
  <mi fontstyle="normal">&dtri;</mi>
  <mo>&times;</mo>
  <mi fontstyle="normal">E</mi>
</mrow>
<mo>=</mo>
<mrow>
  <mo>-</mo>
  <mfrac>
    <mrow>
      <mi>&part;</mi>
      <mo>&ApplyFunction;</mo>
      <mi fontstyle="normal">B</mi>
    </mrow>
    <mrow>
      <mi>&part;</mi>
      <mo>&ApplyFunction;</mo>
      <mi>t</mi>
    </mrow>
  </mfrac>
</mrow>
```

⋮

1.3.1.3 Seitenbeschreibungssprachen

Seitenbeschreibungssprachen sind Ausgabeformate von Textverarbeitungssystemen. In der Drucker-Seitenbeschreibungssprache Postscript (PS) wird ein eigener Zeichensatz benutzt:

octal	0	1	2	3	4	5	6	7
\ 00x								
\ 01x								
\ 02x								
\ 03x								
\ 04x		!	"	#	$	%	&	’
\ 05x	(	)	*	+	,	-	.	/
\ 06x	0	1	2	3	4	5	6	7
\ 07x	8	9	:	;	<	=	>	?
\ 10x	@	A	B	C	D	E	F	G
\ 11x	H	I	J	K	L	M	N	O
\ 12x	P	Q	R	S	T	U	V	W
\ 13x	X	Y	Z	[	\	]	^	_
\ 14x	‘	a	b	c	d	e	f	g
\ 15x	h	i	j	k	l	m	n	o
\ 16x	p	q	r	s	t	u	v	w
\ 17x	x	y	z	{	|	}	~	
\ 20x								
\ 21x								
\ 22x								
\ 23x								
\ 24x		¡		£	/		*f*	§
\ 25x		’	“	«			fi	fl
\ 26x		-	†	‡	·		¶	•
\ 27x	,	„	"	»	...			¿
\ 30x		`	´	^	~	¯	˘	˙
\ 31x	¨					"		ˇ
\ 32x	—							
\ 33x								
\ 34x		Æ						
\ 35x		Ø	Œ					
\ 36x		æ				ı		
\ 37x		ø	œ	ß				

octal	0	1	2	3	4	5	6	7
\00x								
\01x								
\02x								
\03x								
\04x		!	∀	#	∃	%	&	∋
\05x	(	)	*	+	,	-	.	/
\06x	0	1	2	3	4	5	6	7
\07x	8	9	:	;	<	=	>	?
\10x	≅	Α	Β	Χ	Δ	Ε	Φ	Γ
\11x	Η	Ι	ϑ	Κ	Λ	Μ	Ν	Ο
\12x	Π	Θ	Ρ	∑	Τ	Υ	ς	Ω
\13x	Ξ	Ψ	Ζ	[	∴	]	⊥	_
\14x	-	α	β	χ	δ	ε	φ	γ
\15x	η	ι	φ	κ	λ	μ	ν	ο
\16x	π	θ	ρ	σ	τ	υ	ϖ	ω
\17x	ξ	ψ	ζ	{	|	}	∼	
\20x								
\21x								
\22x								
\23x								
\24x		ϒ	’	≤	/	∞	ƒ	♣
\25x			♠	↔	←	↑	→	↓
\26x	°	±	"	≥	×	∝	∂	•
\27x	÷	≠	≡	≈	...	|	—	
\30x	ℵ	ℑ	ℜ	℘	⊗	⊕		∩
\31x	∪	⊃	⊇	⊄	⊂	⊆	∈	∉
\32x	∠	∇		©	™	∏	√	·
\33x	¬	∧	∨	⇔	⇐	⇑	⇒	⇓
\34x	◊	⟨		©	™	∑	⎛	
\35x	⎝	⎡	⎪	⎣	⎧	⎨	⎩	
\36x		⟩	∫		⎮		⎞	
\37x	⎠	⎤	⎪	⎦	⎫	⎬	⎭	

char	name	octal	char	name	octal	char	name	octal
	space	040	\	backslash	134	†	dagger	262
!	exclam	041	]	bracketright	135	‡	daggerdbl	263
"	quotedbl	042	^	asciicircum	136	·	periodcentered	264
#	numbersign	043	_	underscore	137	¶	paragraph	266
$	dollar	044	‘	quoteleft	140	•	bullet	267
%	percent	045	a-z	a-z	141-172	‚	quotesinglbase	270
&	ampersand	046	{	braceleft	173	„	quotedblbase	271
’	quoteright	047	|	bar	174	"	quotedblright	272
(	parenleft	050	}	braceright	175	»	guillemotright	273
)	parenright	051	~	asciitilde	176	...	ellipsis	274
*	asterisk	052	¡	exclaimdown	241		perthousand	275
+	plus	053	¢	cent	242	¿	questiondown	277
,	comma	054	£	sterling	243	`	grave	301
–	hyphen	055	/	fraction	244	´	acute	302
.	period	056		yen	245	ˆ	circumflex	303
/	slash	057	*f*	florin	246	˜	tilde	304
0-9	zero-nine	060-071	§	section	247	¯	macron	305
:	colon	072		currency	250	˘	breve	306
;	semicolon	073	’	quotesingle	251	˙	dotaccent	307
<	less	074	"	quotedblleft	252	¨	dieresis	310
=	equal	075	«	guillemotleft	253	°	ring	312
>	greater	076		guilsinglleft	254	¸	cedilla	313
?	question	077		guilsinglright	255	˝	hungarumlaut	315
@	at	100	fi	fi	256		ogonek	316
A-Z	A-Z	101-132	fl	fl	257	ˇ	caron	317
[	bracketleft	133	–	endash	261	—	emdash	320
	⋮			⋮			⋮	

Tabelle 1.13: Standard Text Characters

```
%!PS-Adobe-2.0
%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/reencsmalldict 12 dict def
/ReEncodeSmall
{

reencsmalldict begin
/newcodesandnames              exch def
/newfontname                   exch def
/basefontname                  exch def

/basefontdict basefontname findfont def
/newfont basefontdict maxlength dict def
basefontdict
{ exch dub /FID ne
        { dub /Encoding eq
                { exch dup length array copy
                        newfont 3 1 roll put }
        }
        { pop pop }
        ifelse
} forall

newfont /FontName newfontname put
newcodesandnames aload pop
newcodesandnames length 2 idiv
{ newfont /Encoding get 3 1 roll put } repeat
newfontname newfont definefont pop
end
} def

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/germanvec[
8#321 /Adieresis % Ae
8#322 /Odieresis % Oe
8#323 /Udieresis % Ue

8#331 /adieresis % Ae
8#332 /odieresis % Oe
8#333 /udieresis % Ue
8#334 /germandbls % ss
] def

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/TimesBoldItalic /TimesBoldItalic-German germanvec ReEncodeSmall
/TimesboldItalic-German findfont 160 scalefont setfont
612 0 translate
90 rotate
80 350 moveto
(PR\323FUNG) show
/ZapfDingbats findfont 40 scalefont setfont
130 210 moveto
(\053) show
/TimesBoldItalic-German findfont 40 scalefont setfont
( Bitte nicht st\/322ren! ) show
/ZapfDingbats findfont 40 scalefont setfont
-1 1 scale

(\053) show
showpage
```

Abbildung 1.8: Ein Beispiel

Tabelle 1.14: Euler Fraktur medium weight — `eufm10`

	’0	’1	’2	’3	’4	’5	’6	’7	
’00x	𝔡	𝔡	ﬀ	ﬁ	ﬂ	ﬃ	ﬄ	ſt	"0x
’01x									
’02x			‘	’					"1x
’03x									
’04x		!					&	'	"2x
’05x	(	)	*	+	,	−	.	/	
’06x	0	1	2	3	4	5	6	7	"3x
’07x	8	9	:	;		=		?	
’10x		𝔄	𝔅	ℭ	𝔇	𝔈	𝔉	𝔊	"4x
’11x	ℌ	ℑ	𝔍	𝔎	𝔏	𝔐	𝔑	𝔒	
’12x	𝔓	𝔔	ℜ	𝔖	𝔗	𝔘	𝔙	𝔚	"5x
’13x	𝔛	𝔜	ℨ	[		]	^		
’14x		𝔞	𝔟	𝔠	𝔡	𝔢	𝔣	𝔤	"6x
’15x	𝔥	𝔦	𝔧	𝔨	𝔩	𝔪	𝔫	𝔬	
’16x	𝔭	𝔮	𝔯	𝔰	𝔱	𝔲	𝔳	𝔴	"7x
’17x	𝔵	𝔶	𝔷			"		1	
	"8	"9	"A	"B	"C	"D	"E	"F	

1.3.1.4 Zeichenattribute

Zeichenattribute (*kursiv*, **fett**,...) werden z.B. in TEX in Form eigener Zeichensätze realisiert, wie sie die Tabellen 1.14, 1.15, 1.16 und 1.17 zeigen. TEX ist ein mächtiges Typsatzprogramm insbesondere für mathematisch-naturwissenschaftliche Publikationen.

Tabelle 1.15: Euler cursive (roman) medium weight — `eurm10`

	’0	’1	’2	’3	’4	’5	’6	’7	
’00x	Γ	Δ	Θ	Λ	Ξ	Π	Σ	Υ	"0x
’01x	Φ	Ψ	Ω	α	β	γ	δ	ϵ	
’02x	ζ	η	θ	ι	κ	λ	μ	ν	"1x
’03x	ξ	π	ρ	σ	τ	υ	ϕ	χ	
’04x	ψ	ω	ε	ϑ	ϖ			φ	"2x
’05x									
’06x	0	1	2	3	4	5	6	7	"3x
’07x	8	9	.	,	<	/	>		
’10x	∂	A	B	C	D	E	F	G	"4x
’11x	H	I	J	K	L	M	N	O	
’12x	P	Q	R	S	T	U	V	W	"5x
’13x	X	Y	Z						
’14x	ℓ	a	b	c	d	e	f	g	"6x
’15x	h	i	j	k	l	m	n	o	
’16x	p	q	r	s	t	u	v	w	"7x
’17x	x	y	z	ı	ȷ	℘			
	"8	"9	"A	"B	"C	"D	"E	"F	

Tabelle 1.16: Extra symbols, group 1, medium weight – `msam10`

	’0	’1	’2	’3	’4	’5	’6	’7	
’00x	⊡	⊞	⊠	□	■	▪	◊	⧫	"0x
’01x	↻	↺	⇌	⇋	⊟	⊩	⊪	⊨	
’02x	↠	↞	⇇	⇉	↑↑	↓↓	↾	⇂	"1x
’03x	↿	⇃	⇀	↼	⇆	⇄	↰	↱	
’04x	⇝	⇜⇝	↫	↬	≑	≳	≳	≳	"2x
’05x	⊸	∴	∵	÷	≜	≲	≲	≲	
’06x	⩽	⩾	≼	≽	≺	≦	⩽	≦	"3x
’07x	∖	-	≔	≕	≻	≧	⩾	≧	
’10x	⊏	⊐	▷	◁	⊵	⊴	★	⟃	"4x
’11x	▼	▶	◀	→	←	△	▲	▽	
’12x	≓	≦≧	≧≦	≦≦≧	≧≧≦	¥	⇛	⇚	"5x
’13x	✓	≚	≙	≘	∠	⦟	◁	∝	
’14x	⌣	⌢	⋐	⋑	⋓	⋒	⅄	⅄	"6x
’15x	⟩	⟨	⊆	⊇	≏	≎	⋘	⋙	
’16x	⌜	⌝	®	Ⓢ	⫛	⟊	∽	∾	"7x
’17x	⌞	⌟	✠	☾	⊤	⊙	⊛	⊖	
	"8	"9	"A	"B	"C	"D	"E	"F	

Tabelle 1.17: Extra symbols, group 2, medium weight – `msbm10`

	’0	’1	’2	’3	’4	’5	’6	’7	
’00x	$\leqslant$	$\geqslant$	$\nleqslant$	$\ngeqslant$	$\lneq$	$\gneq$	$\lneqq$	$\gneqq$	"0x
’01x	$\leqslant$	$\geqslant$	$\nleqslant$	$\ngeqslant$	$\lneq$	$\gneq$	$\lneqq$	$\gneqq$	
’02x	$\lesssim$	$\gtrsim$	$\lesssim$	$\gtrsim$	$\nleqslant$	$\ngeqslant$	$\lneqq$	$\gneqq$	"1x
’03x	$\lessapprox$	$\gtrapprox$	$\lessapprox$	$\gtrapprox$	$\precnsim$	$\succnsim$	$\diagup$	$\diagup$	
’04x	$\subsetneqq$	$\supsetneqq$	$\nsubseteqq$	$\nsupseteqq$	$\subsetneq$	$\supsetneq$	$\subsetneq$	$\supsetneq$	"2x
’05x	$\subsetneqq$	$\supsetneqq$	$\nsubseteqq$	$\nsupseteqq$	$\nparallel$	$\nmid$	$\shortmid$	$\shortparallel$	
’06x	$\nvdash$	$\nvDash$	$\nVdash$	$\nVDash$	$\ntriangleleft$	$\ntriangleright$	$\ntriangleleft$	$\ntriangleright$	"3x
’07x	$\leftarrowtail$	$\rightarrowtail$	$\nleftarrow$	$\nrightarrow$	$\nLeftarrow$	$\nRightarrow$	$\divideontimes$	$\varnothing$	
’10x	$\nexists$	$\mathbb{A}$	$\mathbb{B}$	$\mathbb{C}$	$\mathbb{D}$	$\mathbb{E}$	$\mathbb{F}$	$\mathbb{G}$	"4x
’11x	$\mathbb{H}$	$\mathbb{I}$	$\mathbb{J}$	$\mathbb{K}$	$\mathbb{L}$	$\mathbb{M}$	$\mathbb{N}$	$\mathbb{O}$	
’12x	$\mathbb{P}$	$\mathbb{Q}$	$\mathbb{R}$	$\mathbb{S}$	$\mathbb{T}$	$\mathbb{U}$	$\mathbb{V}$	$\mathbb{W}$	"5x
’13x	$\mathbb{X}$	$\mathbb{Y}$	$\mathbb{Z}$	$\frown$	$\frown$	$\smile$	$\smile$		
’14x	$\Finv$	$\Game$					$\eth$	$\eth$	"6x
’15x	$\backsimeq$	$\beth$	$\gimel$	$\daleth$	$\dashleftarrow$	$\dashrightarrow$	$\ltimes$	$\rtimes$	
’16x	$\shortmid$	$\shortparallel$	$\diagdown$	$\sim$	$\approx$	$\approx$	$\approxeq$	$\approxeq$	"7x
’17x	$\curvearrowleft$	$\curvearrowright$	$\digamma$	$\varkappa$	$\Bbbk$	$\hbar$	$\hbar$	$\backepsilon$	
	"8	"9	"A	"B	"C	"D	"E	"F	

Die Formeln

$$\nabla \cdot E = \frac{\rho}{\epsilon_0} \qquad \nabla \times E = -\frac{\partial B}{\partial t} \qquad c^2 \nabla \times B = \frac{\partial E}{\partial t} + \frac{j}{\epsilon_0} \qquad \nabla \cdot B = 0$$

werden in TEX folgendermaßen codiert:

```
$$ \nabla \cdot E = \frac{\rho}{\epsilon_0}
\quad\quad \nabla \times E = - \frac{\partial B}{\partial t}
\quad\quad c^2 \nabla \times B = \frac{\partial E}{\partial t}
+ \frac{j}{\epsilon_0}
\quad\quad \nabla \cdot B = 0 $$
```

Ähnliches gilt für Fenstersysteme wie z.B. X-Windows:

```
⋮
-urw-itc avant garde-medium-o-normal-sans-10-100-72-72-p-57-iso8859-1
-urw-itc avant garde-medium-o-normal-sans-12-120-72-72-p-68-iso8859-1
-urw-itc avant garde-medium-o-normal-sans-14-140-72-72-p-79-iso8859-1
-urw-itc avant garde-medium-o-normal-sans-6-60-72-72-p-34-iso8859-1
-urw-itc avant garde-medium-o-normal-sans-8-80-72-72-p-45-iso8859-1
-urw-itc avant garde-medium-r-normal-sans-0-0-0-0-p-0-iso8859-1
-urw-itc avant garde-medium-r-normal-sans-0-0-72-72-p-0-iso8859-1
-urw-itc avant garde-medium-r-normal-sans-10-100-72-72-p-57-iso8859-1
-urw-itc avant garde-medium-r-normal-sans-12-120-72-72-p-68-iso8859-1
-urw-itc avant garde-medium-r-normal-sans-14-140-72-72-p-79-iso8859-1
-urw-itc avant garde-medium-r-normal-sans-6-60-72-72-p-34-iso8859-1
-urw-itc avant garde-medium-r-normal-sans-8-80-72-72-p-45-iso8859-1
-urw-itc bookman-demi-i-normal--0-0-0-0-p-0-iso8859-1
-urw-itc bookman-demi-i-normal--0-0-72-72-p-0-iso8859-1
-urw-itc bookman-demi-i-normal--10-100-72-72-p-61-iso8859-1
-urw-itc bookman-demi-i-normal--12-120-72-72-p-72-iso8859-1
-urw-itc bookman-demi-i-normal--14-140-72-72-p-85-iso8859-1
-urw-itc bookman-demi-i-normal--6-60-72-72-p-36-iso8859-1
-urw-itc bookman-demi-i-normal--8-80-72-72-p-48-iso8859-1
-urw-itc bookman-demi-r-normal--0-0-0-0-p-0-iso8859-1
-urw-itc bookman-demi-r-normal--0-0-72-72-p-0-iso8859-1
-urw-itc bookman-demi-r-normal--10-100-72-72-p-60-iso8859-1
-urw-itc bookman-demi-r-normal--12-120-72-72-p-73-iso8859-1
-urw-itc bookman-demi-r-normal--14-140-72-72-p-84-iso8859-1
-urw-itc bookman-demi-r-normal--6-60-72-72-p-36-iso8859-1
-urw-itc bookman-demi-r-normal--8-80-72-72-p-48-iso8859-1
-urw-itc bookman-light-i-normal--0-0-0-0-p-0-iso8859-1
-urw-itc bookman-light-i-normal--0-0-72-72-p-0-iso8859-1
⋮
```

Tabelle 1.18: Beispiele für Zeichensätze in X-Windows

Zusammenfassend eine interessante Tabelle[2]:

Symbol	ISO 8879	Adobe name	Adobe Hex	Unicode	Unicode name
ISO Latin 1 Entities					
“	quot	quotedbl	22	0022	QUOTATION MARK
&	amp	ampersand	26	0026	AMPERSAND
<	lt	less	3C	003C	LESS-THAN-SIGN
>	gt	greater	3E	003E	GREATER-THAN-SIGN
	nbsp		A0	00A0	NON-BREAKING-SPACE
¡	iexcl	exclamdown	A1	00A1	INVERTED EXCLAMATION MARK
	cent	cent	A2	00A2	CENT SIGN
£	pound	sterling	A3	00A3	POUND SIGN
	curren	currency	A4	00A4	CURRENCY SIGN
	yen	yen	A5	00A5	YEN SIGN
	brvbar	brokenbar	A6	00A6	BROKEN VERTICAL BAR
§	sect	section	A7	00A7	SECTION SIGN
¨	die	dieresis	A8	00A8	SPACING DIAERESIS
©	copy	copyright	A9	00A9	COPYRIGHT SIGN
^{a}	ordf	ordfeminine	AA	00AA	FEMININE ORDINAL INDICATOR
≪	laquo	guillemotleft	AB	00AB	LEFT POINTING GUILLEMET
¬	not	logicalnot	AC	00AC	NOT SIGN
-	shy	hyphen	AD	00AD	SOFT HYPHEN
®	reg	registered	AE	00AE	REGISTERED TRADE MARK SIGN
¯	macron	macron	AF	00AF	SPACING MACRON
°	degree	degree	B0	00B0	DEGREE SIGN
±	plusmn	plusminus	B1	00B1	PLUS-OR-MINUS SIGN
^{2}	sup2	twosuperior	B2	00B2	SUPERSCRIPT DIGIT TWO
^{3}	sup3	threesuperior	B3	00B3	SUPERSCRIPT DIGIT THREE

⋮

[2] http://www.w3.org/MarkUp/HTMLPlus/htmlplus_table.ps

1.3.1.5 Die Zukunft durch Zeichenvielfalt

Eine weltweite Anwendbarkeit kann erreicht werden, falls 16Bit für die Codierung zur Verfügung stehen: *UNICODE* , der in neuen Programmiersprachen wie etwa *JAVA* schon benutzt wird. **Statistik der Version 1.0**

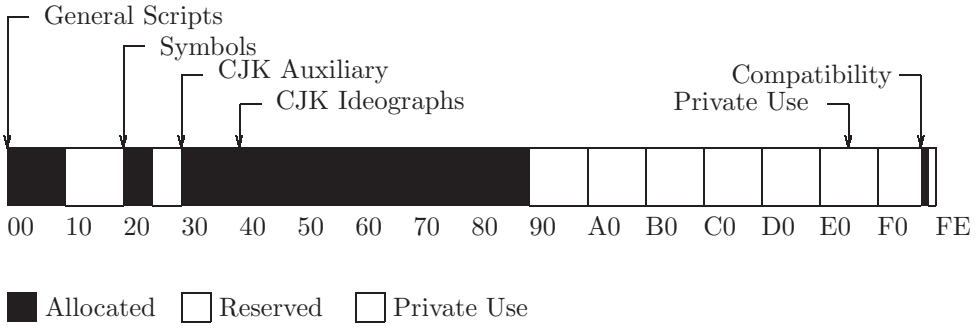


Abbildung 1.9: Verteilung UNICODE

Die folgende Tabelle zeigt die Anteile des gesamten im UNICODE zur Verfügung stehenden Platzes, die verschiedenen Schrifttypen in der Version 1.0 bereits zugeteilt wurden:

	Allocated	*Unassigned*	*% Assigend*	
General	2336	5856	29%	
Symbols	1290	2806	31%	
CJK symbols	763	261	75%	
Hangul	2350	450	84%	
Han Compatibility	268	4	99%	(Volume 2)
Ideographic & other	20733	22275	48%	(Volume 2)
User Space	5632	N/A	N/A	
Compatibility Zone	362	133	73%	
Special	1	13		
FEFF	1	0		
FFFE, FFFF	N/A	2		
Totals	28706 (assigned)			
	+ 5632 (private use)			
	= 34338 (allocated)			
	52%			

Mit noch über 30000 unbenutzten Character Positionen besitzt der UNICODE auch für die Zukunft noch genug Raum für weitere Expansionen.

Die Zukunft des UNICODES
In Zukunft wird der UNICODE Standard um weniger verbreitete und veraltete Schrifttypen erweitert. Schrifttypen dieser Art werden jedoch nicht in ihrer ursprünglichen Form eingebunden, da sich ihr Nutzen schwer einschätzen läßt. So wird bei vielen dieser Schriften eine ausführliche Diskussion nötig sein, bis ein zufriedenstellendes Codierungsschema vorliegt. Die fünf Schriftarten *Ethiopian, Burmese, Khmer, Sinhala* und *Mongolian* werden zum Standard UNICODE hinzugefügt, sobald zuverlässige Informationen über sie vorliegen. Weitere Schriftarten, die für eine mögliche Aufnahme vorgesehen sind, sind

- *Inuktitut/Cree Syllabary:* Das Kommunikationsministerium von Kanada untersucht Standardisierungen von verschiedenen Dialektarten, die von Cree und/oder Inuktitut gesprochen werden und sucht Codierungsschemen.
- *Egyptian Hieroglyphics:* Ein einheitliches Codierungsschema existiert und wird vorangetrieben.
- *Korean Hangul Syllables:* Eventuell werden noch weitere Korean Hangul Dialekte hinzugefügt.

Der Unterschied zwischen der logischen Anordung von Zeichen und der Anordnung auf dem Bildschirm zeigt die Abbildung 1.10

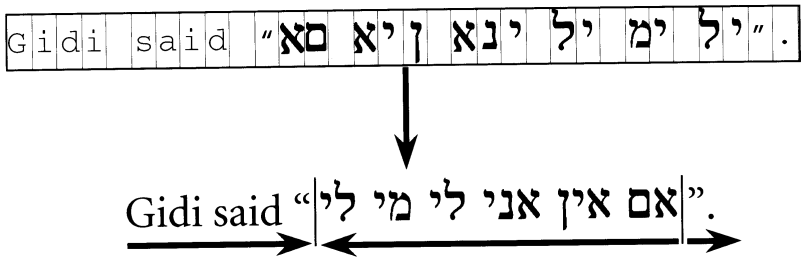


Abbildung 1.10: Bidirectional Ordering

General Scripts
Im *General Scripts*-Bereich des UNICODEs sind alle lateinischen und nichtideographischen Schriftzeichen codiert:

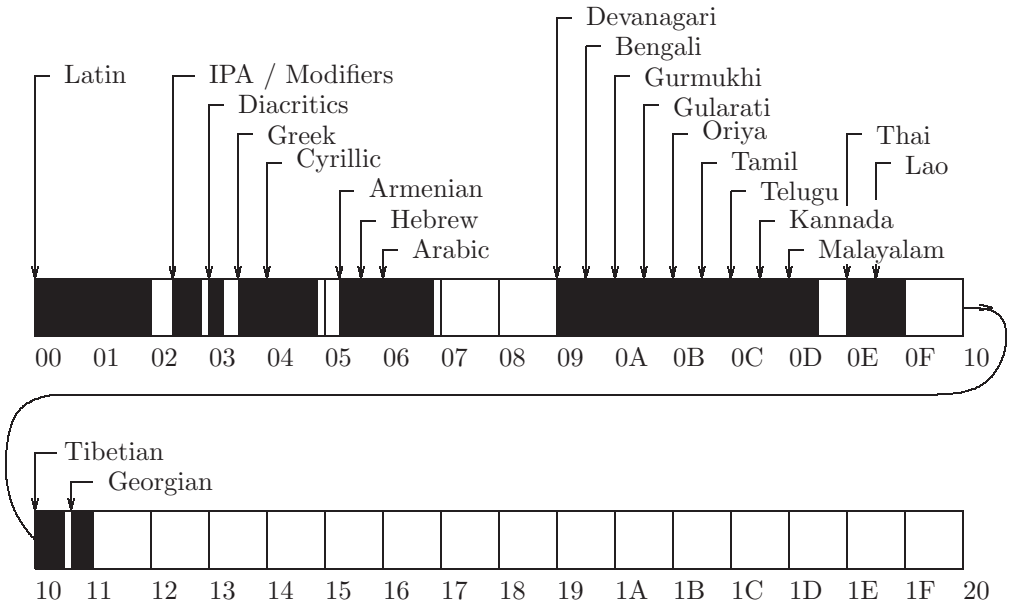


Abbildung 1.11: General Scripts

Control		**ASCII**						**Control**		**Latin 1**					
000	001	002	003	004	005	006	007	008	009	00A	00B	00C	00D	00E	00F
NUL	DLE		0	@	P	‘	p	CTRL	CTRL		°	À		à	
SOH	DC1	!	1	A	Q	a	q	CTRL	CTRL	¡	±	Á	Ñ	á	ñ
STX	DC2	"	2	B	R	b	r	CTRL	CTRL		^{2}	Â	Ò	â	ò
ETX	DC3	#	3	C	S	c	s	CTRL	CTRL	£	^{3}	Ã	Ó	ã	ó
EOT	DC4	$	4	D	T	d	t	CTRL	CTRL		’	Ä	Ô	ä	ô
ENQ	NAK	%	5	E	U	e	u	CTRL	CTRL		μ	Å	Õ	å	õ
ACK	SYN	&	6	F	V	f	v	CTRL	CTRL	|	¶	Æ	Ö	æ	ö
BEL	ETB	’	7	G	W	g	w	CTRL	CTRL	§	·	Ç	×	ç	÷
BS	CAN	(	8	H	X	h	x	CTRL	CTRL	¨	¸	È	Ø	è	ø
HT	EM	)	9	I	Y	i	y	CTRL	CTRL	©	^{1}	É	Ù	é	ù
LF	SUB	*	:	J	Z	j	z	CTRL	CTRL	^{a}	^{o}	Ê	Ú	ê	ú
VT	ESC	+	;	K	[	k	{	CTRL	CTRL	≪	≫	Ë	Û	ë	û
FF	FS	,	<	L	\	l	|	CTRL	CTRL	¬	$\frac{1}{4}$	Ì	Ü	ì	ü
CR	GS	-	=	M	]	m	}	CTRL	CTRL	-	$\frac{1}{2}$	Í	Ý	í	ý
SO	RS	.	>	N	^	n	~	CTRL	CTRL		$\frac{3}{4}$	Î		î	
SI	US	/	?	O	_	o	DEL	CTRL	CTRL	¯	¿	Ï	ß	ï	ÿ

Tabelle 1.19: UNICODE Version 1.0, Character Blocks 0000-00FF

2000..206F	Zeichen für die Zeichensetzung: ‚.:;“„ ...
2070..209F	Subscripts und Superscripts: ^{2},^{3} ,^{4}, ...
20A0..20CF	Währungssymbole: £,$, ...
20D0..20FF	diakretische Zeichen: $\leftarrow$, $\rightarrow$, ...
2100..214F	buchstabenähnliche Zeichen: $\mathcal{F}$,°F, ...
2150..218F	Zahlen: $\frac{1}{3}$, `I`, `VII`, ...
2190..21FF	Pfeile: $\uparrow$, $\mapsto$, ...
2200..22FF	mathematische Sonderzeichen: $\forall$, $\exists$, $\in$, ...
2300..23FF	verschiedene technische Sonderzeichen: #, ...
2400..243F	Symbole für Control-Zeichen: `NUL`,`ESC`, ...
2440..245F	OCR-Zeichen
2460..24FF	eingerahmte alphanumerische Zeichen: ©, ...
2500..257F	Formular- und Diagrammzeichen: $\vdash$, $\dashv$, $\|$, ...
2580..259F	Blockgraphik-Zeichen
25A0..25FF	graphische Symbole
2600..26FF	verschiedene Dingbats
2700..27BF	Zapf-Dingbats
...	
3000..303F	CJK-Symbole
3040..309F	Hiragana
30A0..30FF	Katakana
...	

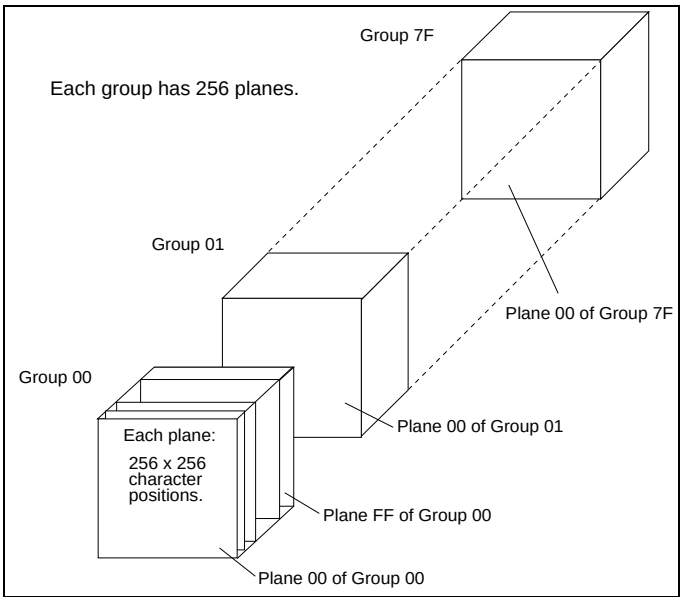
Tabelle 1.20: Weitere Zeichenbereiche

Ein Problem ist die Zeichendarstellung auf dem Bildschirm. Es gibt unter anderem die Methoden des pixelmäßigen Aufbau und des Zeichenaufbaus aus Linien und Kreisegmenten. Ist ein vektorielles Zeichen erst einmal aufgebaut, so ist es ohne Qualitätsverlust beliebig vergrößerbar.

Zeichenaufbau durch Pixel bzw. Linien, Kreissegmente, ...

Der UNICODE stellt weitgehende Kompatibilität zu bestehenden Codes durch (verschobenes) Einfügen oder Bereitstellen von Code-Umwandlungstabellen her: 0000..007F entspricht ASCII. Für andere Codes werden UNICODE-Übersetzungstabellen bereitgestellt, z.B. für UNICODE zu SGML (Tabelle 1.22), UNICODE zu Postscript (Tabelle 1.21) oder UNICODE zu MacIntosh (Tabelle 1.23). Analoge Tabellen gibt es zur Übersetzung von UNICODE zu Microsoft Windows, zu PC Code Page Mappings (Latin, Greek,...), zu EBCDIC Code Page Mappings und weiteren.

Die Notwendigkeit, Control-Codes anderer Codierungen auch verfügbar zu haben, und die Weigerung von Japan und Korea, die vereinheitlichte CJK-Codierung (die mit 19000 statt insgesamt über 31000 Codepositionen ausgekommen wäre) zu akzeptieren, führt zum 32Bit Zeichencode *ISO 10646*, der in seinen ersten 65536 Codes die Zeichen des Unicodes benutzt:



Näheres zum ISO- bzw. Unicode lese man bei

http://dns.hti.umich.edu/htistaff/pubs/1997/janete.01/,
http://www.indigo.ie/egt/standards/iso10646/bmp-today-table.html,
http://wwwinfo.cern.ch/asdoc/WWW/publications/ictp99/ictp99N2705.html

und

http://www.unicode.org/

nach. Die nun zur Verfügung stehenden 4294967296 Codes dürften voraussichtlich für eine Codierung auch der ausgefallensten (ausgestorbenen) Schriften ausreichen.

UNIC	ISO StdEnc	Latin1	ZapfDB Symbol	Adobe glyph name	Unicode character name
0020	20	20	20	space	SPACE
0021	21	21	21	exclam	ECLAMATION MARK
0022	22	22		quotedbl	QUOTATION MARK
0023	23	23	23	numbersign	NUMBERSIGN
0024	24	24		dollar	DOLLAR SIGN
0025	25	25	25	percent	PERCENT SIGN
0026	26	26	26	ampersand	AMPERSAND
0027	A9			quotesingle	APOSTROPHE-QUOTE
0028	28	28	28	parenleft	OPENING PARENTHESIS
0029	29	29	29	parenright	CLOSING PARENTHESIS
002A	2A	2A		asterisk	ASTERISK
002B	2B	2B	2B	plus	PLUS SIGN
002C	2C	2C	2C	comma	COMMA
002D	2D	AD		hyphen	HYPHEN-MINUS
002D		2D		minus	HYPHEN-MINUS
002E	2E	2E	2E	period	PERIOD
002F	2F	2F	2F	slash	SLASH
0030	30	30	30	zero	DIGIT ZERO
0031	31	31	31	one	DIGIT ONE
0032	32	32	32	two	DIGIT TWO
0033	33	33	33	three	DIGIT THREE
0034	34	34	34	four	DIGIT FOUR
0035	35	35	35	five	DIGIT FIVE
0036	36	36	36	six	DIGIT SIX
0037	37	37	37	seven	DIGIT SEVEN
0038	38	38	38	eight	DIGIT EIGHT
0039	39	39	39	nine	DIGIT NINE
003A	3A	3A	3A	colon	COLON
003B	3B	3B	3B	semicolon	SEMIKOLON

Tabelle 1.21: UNICODE to Adobe Standard Mappings

UNIC	6862.2	SGML	Unicode character name
0021		excl	EXCLAMATION MARK
0023		num	NUMBER SIGN
0024		dollar	DOLLAR SIGN
0025		percnt	PERCENT SIGN
0026		amp	AMPERSAND
0027		quot	APOSTROPHE-QUOTE
0028		lpar	OPENING PARENTHESIS
0029		rpar	CLOSING PARENTHESIS
002A		ast	ASTERISK
002B	05.00	plus	PLUS SIGN
002C		comma	COMMA
002D		hyphen	HYPHEN-MINUS
002E		period	PERIOD
002F		sol	SLASH
003A		colon	COLON
003B		semi	SEMICOLON
003C		lt	LESS-THAN SIGN
003D		equals	EQUALS SIGN
003E		gt	GREATER-THAN SIGN
003F		quest	QUESTION MARK
0040		commat	COMMERCIAL AT
005B		lsqb	OPENING SQUARE BRACKET
005C		bsol	BACKSLASH
005D		rsqb	CLOSING SQUARE BRACKET
005E		circ	SPACING CIRCUMFLEX
005F		lowbar	SPACING UNDERSCORE
0060		grave	SPACING GRAVE
007B		lcub	OPENING CURLY BRACKET
007C		verbar	VERTICAL BAR
007D		rcub	CLOSING CURLY BAR
007E		tilde	TILDE
00A0		nbsp	NON-BREAKING SPACE
00A1		iexcl	INVERTED EXCLAMATION MARK
00A2		cent	CENT SIGN
00A3		pound	POUND SIGN

Tabelle 1.22: The UNICODE to SGML (ISO DIS 6862.2) Mappings

UNIC	ROM	SYM	GRK	GK2	HEB	ARB	NAME
0020	20	20	20	20	20/A0	21/A1	SPACE
0021	21	21	21	21	21/A1	21/A1	ECLAMATION MARK
0022	22		22	22	22/A2	22/A2	QUOTATION MARK
0023	23	23	23	23	23/A3	23/A3	NUMBERSIGN
0024	24		24	24	24/A4	24/A4	DOLLAR SIGN
0025	25	25	25	25	25/A5	25	PERCENT SIGN
0026	26	26	26	26	26	26/A6	AMPERSAND
0027	27		27	27	27/A7	27/A7	APOSTROPHE-QUOTE
0028	28	28	28	28	28/A8	28/A8	OPENING PARENTHESIS
0029	29	29	29	29	29/A9	29/A9	CLOSING PARENTHESIS
002A	2A		2A	2A	2A/AA	2A/AA	ASTERISK
002B	2B	2B	2B	2B	2B/AB	2B/AB	PLUS SIGN
002C	2C	2C	2C	2C	2C/AC	2C	COMMA
002D	2D		2D	2D	2D/AD	2D/AD	HYPHEN-MINUS
002E	2E	2E	2E	2E	2E/AE	2E/AE	PERIOD
002F	2F	2F	2F	2F	2F/AF	2F/AF	SLASH
0030	30	30	30	30	30/B0	30	DIGIT ZERO
0031	31	31	31	31	31/B1	31	DIGIT ONE
0032	32	32	32	32	32/B2	32	DIGIT TWO
0033	33	33	33	33	33/B3	33	DIGIT THREE
0034	34	34	34	34	34/B4	34	DIGIT FOUR
0035	35	35	35	35	35/B5	35	DIGIT FIVE
0036	36	36	36	36	36/B6	36	DIGIT SIX
0037	37	37	37	37	37/B7	37	DIGIT SEVEN
0038	38	38	38	38	38/B8	38	DIGIT EIGHT
0039	39	39	39	39	39/B9	39	DIGIT NINE
003A	3A	3A	3A	3A	3A/BA	3A/BA	COLON
003B	3B	3B	3B	3B	3B/BB	3B	SEMICOLON
003C	3C	3C	3C	3C	3C/BC	3C/BC	LESS-THAN SIGN
003D	3D	3D	3D	3D	3D/BD	3D/BD	EQUALS-SIGN
003E	3E	3E	3E	3E	3E/BE	3E/BE	GREATER-THAN SIGN
003F	3F	3F	3F	3F	3F/BF	3F	QUESTION MARK
0040	40		40	40	40	40	COMMERCIAL AT
0041	41		41	41	41	41	LATIN CAPITAL LETTER A
0042	42		42	42	42	42	LATIN CAPITAL LETTER B
0043	43		43	43	43	43	LATIN CAPITAL LETTER C
0044	44		44	44	44	44	LATIN CAPITAL LETTER D
0045	45		45	45	45	45	LATIN CAPITAL LETTER E
0046	46		46	46	46	46	LATIN CAPITAL LETTER F
0047	47		47	47	47	47	LATIN CAPITAL LETTER G
⋮							

Tabelle 1.23: UNICODE to Macintosh Mappings

1.3.2 Codierung von Zahlen

1.3.2.1 Dezimalziffern

Um die zehn Ziffern des Dezimalsystems binär zu codieren, braucht man (mindestens) 4 Bit; im *BCD-Code* oder *8-4-2-1–Code* wird nun jede Dezimalziffer durch die Bitkombination dargestellt, die im Stellenwertsystem der Basis 2 den entsprechenden Wert hat:

$$\sum_{i=0}^{3} m_i \cdot 2^i, \quad m_i \in \{0, 1\}$$

In einem Byte kann man dann zwei Dezimalstellen, in 4 Byte (32 Bit)

Wert	m_3	m_2	m_1	m_0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

Tabelle 1.24: Zifferndarstellung im BCD-Code

8 Dezimalstellen bzw. in 8 Byte (64 Bit) 16 Dezimalstellen unterbringen. Abgesehen davon, daß in 4 Byte im BCD-Code die Zahlen

$$00.000.000 \ldots 99.999.999,$$

im Dualsystem jedoch die (im Binärstellenwertsystem zu interpretierenden Zahlen)

$$0 \ldots \sum_{i=0}^{31} 1 \cdot 2^i,$$

also

$$0 \ldots 2^{32} - 1 = 0 \ldots 4.294.967.295$$

codiert werden können, ist auch das Rechnen mit BCD-codierten Zahlen aufwendiger: es benötigt mehr Schaltfunktionen / Gatter:

$$
\begin{array}{rr}
 & 9_{10} \\
+ & 3_{10} \\
\hline
 & 12_{10}
\end{array}
\qquad
\begin{array}{rr}
 & 1001_2 \\
+ & 0011_2 \\
\hline
 & 1100 \\
+ & 0110 \\
\hline
0001 & 0010
\end{array}
\quad \leftarrow \text{kein BCD-Code, deshalb Korrektur um 6}
$$

Ziffernaddition: 1 Addition, 1 Vergleich > 9 und ggf. 1 weitere Addition

Schließlich gibt es Binärmuster, die nicht als BCD-codierte Zahl auftreten können, die Abbildung ist also nicht bijektiv. Aus Gründen der besseren Ökonomie benutzen moderne Computer deshalb fast ausschließlich binär codierte Zahlen.

1.3.2.2 Vorzeichenlose ganze Zahlen

Wortlänge		Zahlenwerte
Bit		0...1
Byte	8 Bit	$0...2^8 - 1 = 255$
Halbwort	16 Bit	$0...2^{16} - 1 = 65535$
Wort	32 Bit	$0...2^{32} - 1 = 4\,294\,967\,295$
Doppelwort	64 Bit	$0...2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$
Quadwort	128 Bit	$0...2^{128} - 1$
		$= 340\,282\,366\,920\,938\,463\,463\,374\,607\,431\,768\,211\,455$

Kurzschreibweise für lange Binärfolgen:

$1011\,1010_2$	dual
\o272	oktal
\xBA	hexadezimal

Es besteht die Möglichkeit, vorzeichenlose ganze Zahlen im Stellenwertsystem zur Basis b darzustellen. Dazu sei $x \in \mathbb{N}$. Dann gilt für beliebiges $b \in \mathbb{N} \setminus \{0, 1\}$:

$i := 0;$
Wiederhole
  $z_i := x$ rem $b;$
  $x := x$ div $b;$
  $i :=$ succ $(i);$
bis ($x = 0$)

liefert die Stellen von x zur Basis b:

$$x = \sum_{i=0}^{k} z_i b^i$$

Aufgaben:

a) Beweise die Korrektheit des angegebenen Verfahrens.

b) Wie berechnet man den Wert von $(z_N \cdots z_2 z_1 z_0)_b$? (Hinweis: Man benutze das Horner-Schema

$$(\ldots((z_k \cdot b + z_{k-1}) \cdot b + z_{k-2}) \cdot b + \ldots) \cdot b + z_0$$

und gebe den entsprechenden Pseudocode an.)

1.3.2.3 Addition statt Subtraktion

Wie berechnet man $1236 + 995$? Nach einer Kopfrechenregel rechnet man

$$1236 + \underbrace{1000 - 5}_{=995} = 2236 - 5 = 2231,$$

d.h. man rechnet

$$\underbrace{1236 - 5}_{\text{Subtraktion}} = \underbrace{1236 + 995}_{\text{Addition}} - \underbrace{1000}_{\text{Korrektur}} .$$

Eine noch bessere Methode – insbesondere für den Computer – ist die Benutzung der Zehnerkomplements, d.h. man rechnet

$$1236 - 5 = \underbrace{1236}_{4stellig} + \underbrace{9995}_{4stellig} - 10000$$

und kann wegen des Korrekturterms den Übertrag in der 5. Stelle einfach vergessen.

Definition 1.1

Das ((N + 1)-stellige) 10er-Komplement von $\sum_{i=0}^{N} y_i \cdot 10^i, y_i \in \{0, \ldots, 9\}$ *ist definiert als*

$$10^{N+1} - \sum_{i=0}^{N} y_i \cdot 10^i$$

Es kann entweder durch eine $(N + 2)$*-stellige Subtraktion oder durch Bestimmung des 9er Komplements mit anschließender Korrekturaddition mit* 1 *bestimmt werden.*

Definition 1.2
Das ((N + 1)-stellige) 9er-Komplement von $\sum_{i=0}^{N} y_i \cdot 10^i, y_i \in \{0, \ldots, 9\}$ *ist definiert als*

$$\sum_{i=0}^{N} 9 \cdot 10^i - \sum_{i=0}^{N} y_i \cdot 10^i = \sum_{i=0}^{N} (9 - y_i) \cdot 10^i$$

Es kann durch $(N + 1)$ *1-ziffrige Subtraktionen bestimmt werden.*

Bemerkung 1.1
9er-Komplement und 10er-Komplement sind selbstinvers.

Bemerkung 1.2

$$\textit{9er-Komplement}(\ \sum_{i=0}^{N} y_i \cdot 10^i\ ) = \textit{10er-Komplement}(\ \sum_{i=0}^{N} y_i \cdot 10^i\ ) - 1$$

Mit Hilfe des 10er Komplements kann also im Beispiel der Subtraktion wie folgt vorgegangen werden:

$$\sum_{i=0}^{N} x_i \cdot 10^i - \sum_{i=0}^{N} y_i \cdot 10^i = \sum_{i=0}^{N} x_i \cdot 10^i + \left(\sum_{i=0}^{N} (9 - y_i) \cdot 10^i + 1 \underbrace{- \sum_{i=0}^{N} 9 \cdot 10^i - 1}_{= -10^{N+1}} \right),$$

d.h. man addiert das 9er Komplement von y, korrigiert um $+1$ und *vergißt* einen eventuell auftretenden Überlauf in der $(N + 2)$-ten Stelle. Codiert man $-y$ durch das Zehnerkomplement von y, so benötigt die Vorzeichenwechseloperation $(N + 1)$ einstellige Subtraktionen und eine $(N + 1)$-stellige Korrekturaddition, die Subtraktion $x - y = x + (-y)$ braucht jedoch nur eine $(N + 1)$-stellige Addition.

Es stellt sich die Frage, welche N-stelligen Zahlen nun negativ und welche positiv interpretiert werden müssen.
Nach Abbildung 1.12 ergibt sich für die zweistelligen Zahlen im 9er Komplement ein dargestellter Zahlenbereich von

$$\{-49, -48, \ldots, -0, 0, 1, \ldots, 49\} = \{-(10^N \textit{ div } 2 - 1), \ldots, -0, +0, \ldots, 10^N \textit{ div } 2 - 1\}$$

und im 10er Komplement

$$\{-50, -49, \ldots, -1, 0, 1, \ldots, 49\} = \{-(10^N \, div \, 2), \ldots, 0, \ldots, 10^N \, div \, 2 - 1\}$$

Im 10er Komplement ergibt sich also

$$Wert(x_N \ldots x_1 x_0) = \begin{cases} \sum_{i=0}^{N} x_i 10^i & \text{, falls } x_N \in \{0,1,2,3,4\} \\ -10^{N+1} + \sum_{i=0}^{N} x_i 10^i & \text{, falls } x_N \in \{5,6,7,8,9\} \end{cases}$$

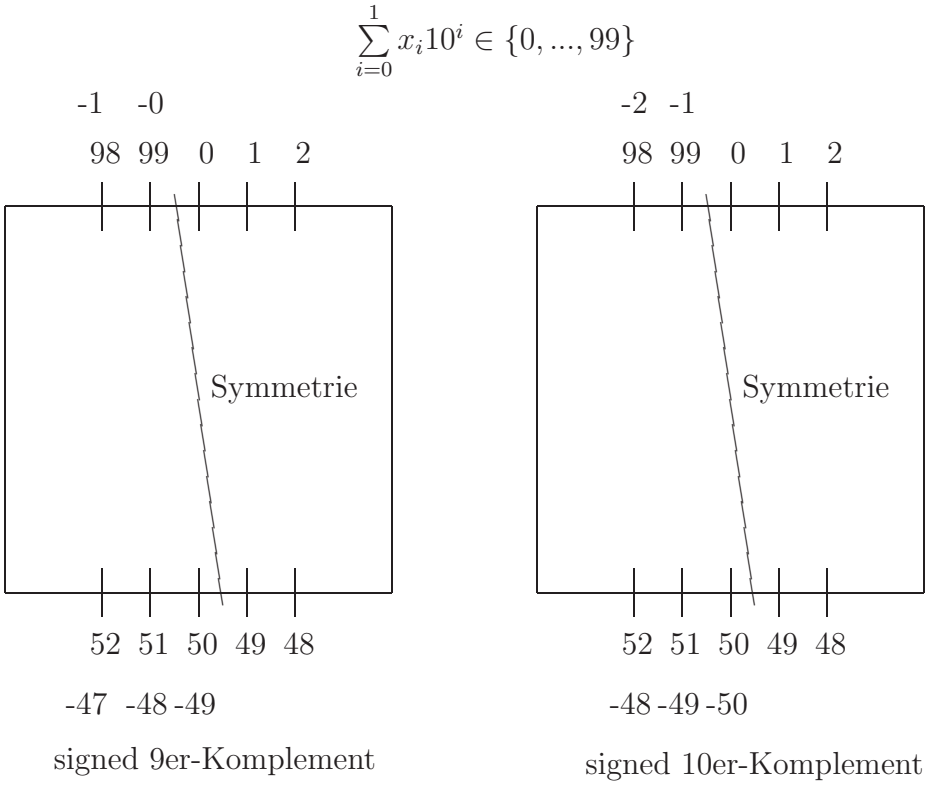


Abbildung 1.12: Darstellung von Zahlen im 9er bzw. 10er Komplement

Wie ist nun für zwei vorzeichenbehaftete (*signed*) 10er Komplement-Zahlen a, b der Wert $W := Wert(a_N \dots a_0) + Wert(b_N \dots b_0)$ zu berechnen?

A) $a_N, b_N \in \{0,1,2,3,4\}$:

$$\begin{aligned} W &= \sum_{i=0}^{N} a_i 10^i + \sum_{i=0}^{N} b_i 10^i \\ &= \sum_{i=0}^{N} (a_i + b_i) 10^i \\ &= \sum_{i=0}^{N} \underbrace{\left(a_i + b_i + \left(\sum_{j=0}^{i-1} (a_j + b_j) 10^j \right) \text{ div } 10^i \right)}_{=:z_i} \cdot 10^i \end{aligned}$$

Falls $z_N \geq 5$ ergibt sich ein Overflow, d.h. W ist nicht im signed 10er Komplement darstellbar! Die Rechnung leistet besser der folgende iterative Algorithmus:

```
uebertrag:= 0;
i:= 0;
Wiederhole
   z_i := x_i + y_i + uebertrag;
   uebertrag:= z_i div 10;
   z_i := z_i mod 10;
   i:=i+1
bis (i≥ N + 1);
falls (uebertrag≠ 0) ...
```

B) $a_N, b_N \in \{5,6,7,8,9\}$: der Algorithmus funktioniert ähnlich, ist $z_N \in \{0,1,2,3,4\}$ so ergibt sich ein Underflow!

C) $a_N \in \{0,1,2,3,4\}, b_N \in \{5,6,7,8,9\}$ oder $a_N \in \{5,6,7,8,9\}, b_N \in \{0,1,2,3,4\}$: wieder ähnlich, aber keine Probleme mit eventuellem Over- oder Underflow.

1.3.2.4 Negative Zahlen im Dualsystem

Auf Großrechnern werden negative Zahlen zuweilen im 1er-Komplement dargestellt, meist jedoch ist eine Darstellung im 2er-Komplement üblich:

Bit 0..1	$-1, 0$
Byte $00..FF$	$-128, .., +127$
	Dabei: oberstes Bit$= 1 \Leftrightarrow$ Wert < 0
	Eigentlich: $0, .., 127, -128, -127, .., -1$
Halbwort (16 Bit)	$-2^{15} = -32768, .., +32767 = 2^{15} - 1$
Wort (32 Bit)	$-2^{31}, .., 2^{31} - 1$
Doppelwort (64 Bit)	$-2^{63}, .., 2^{63} - 1$
Quadwort (128 Bit)	$-2^{127}, .., 2^{127} - 1$

Ein Vorzeichenwechsel wird dabei durch

$$x \mapsto \overline{x} + 1$$

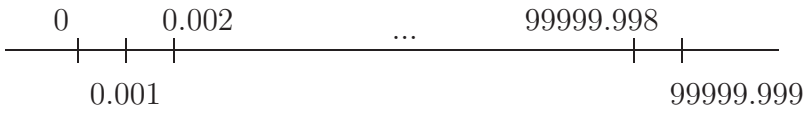
realisiert, wobei $\overline{x}$ das bitweise Komplement von x darstellt, also $0 \leftrightarrow 1$. Dies funktioniert allerdings bei der kleinsten negativen Zahl nicht. Bei der Addition tritt ein Überlauf genau dann auf, wenn im obersten Bit ein Übertrag stattfindet oder (xor) aus dem obersten Bit heraus ein Übertrag erfolgt. Leider stellen viele Programmiersprachen (z.B. C++) einen solchen Überlauf nicht fest!
Die Multiplikation zweier Zahlen erfolgt analog zum schriftlichen Rechenverfahren. Gleiches gilt für die Division. Spezialfunktionen wie *log* oder *arctan* werden durch abbrechende Taylorreihen, Newtoniteration, Kettenbrüche, Funktionaltheoreme o.ä. realisiert. Dies geschieht meist in Software.

1.3.2.5 Festkommazahlen

Festkommazahlen besitzen immer eine feste Anzahl von *Nachkommaziffern*, z.B. 5 Vor- und 3 Nachkommastellen wie bei kg:

$$\mathcal{F}_3^5 := \left\{ (-1)^s \sum_{i=-3}^{5} a_i \cdot 10^i : a_i \in \{0, ..9\}; s \in \{0, 1\} \right\}$$

Veranschaulichung am Zahlenstrahl: $\mathcal{F}_3^5$



Eigenschaften:

- fester Gitterabstand zwischen je zwei benachbarten Zahlen (im Beispiel 10^{-3})
- Addition bis auf Über-/Unterlauf exakt ausführbar
- Multiplikation mit ganzen Zahlen bis auf Über-/Unterlauf exakt ausführbar
- Multiplikation von echt gebrochenen Zahlen mit Rundungsfehlern behaftet

Überlegungen zeigen, daß z.B. die Realisierung von DM-Beträgen mit `real`-Variablen ungünstig ist, denn

$$1\text{Pf} = \frac{1}{2^2 5^2}\text{DM} = 1.\overline{47ae1}_{16}$$

Man rechnet also besser mit Pfennigbeträgen:

```
typedef integer Pfennige;   // z.B. 32 Bit
Pfennige preis;             // also:  -21 474 836,48 .. +21 474 836,47 DM
```

Basisumwandlung von Nachkommastellen:

Es sei $x \in [0, 1)$. Dann gilt für beliebiges $b \in \mathbb{N} \setminus \{1\}$:

$i = -1;$
`Wiederhole`
  $z_i = \text{int}(x * b);$ `//` $\in [0, .., b-1]$
  $x = x * b - z_i;$ `//` $\in [0, 1)$
  i`--`
`bis((x=0) || (`$i < -$`maxStellenzahl))`

liefert mit

$$\sum_{j=\max\{i+1,-maxStellenzahl\}}^{-1} z_j b^j$$

eine *maxStellenzahl*-Näherungsdarstellung von x zur Basis b Sei zum Beispiel $x := \frac{9}{64}$ und $b = 2$. Dann arbeitet der Algorithmus wie folgt:

$$\begin{aligned}
\frac{9}{64} \cdot 2 = \frac{9}{32} &\Rightarrow z_{-1} = 0 \\
\frac{9}{32} \cdot 2 = \frac{9}{16} &\Rightarrow z_{-2} = 0 \\
\frac{9}{16} \cdot 2 = \frac{9}{8} = 1 + \frac{1}{8} &\Rightarrow z_{-3} = 1 \\
\frac{1}{8} \cdot 2 = \frac{1}{4} &\Rightarrow z_{-4} = 0 \\
\frac{1}{4} \cdot 2 = \frac{1}{2} &\Rightarrow z_{-5} = 0 \\
\frac{1}{2} \cdot 2 = 1 &\Rightarrow z_{-6} = 1
\end{aligned}$$

Damit ergibt sich

$$\frac{9}{64} = (0.001001)_2$$

Aufgaben:

a) Berechne $(\frac{1}{100})_{10}$ zur Basis $b = 16$.

b) Wie ist der Algorithmus so zu ändern, daß der Fehler

$$\left| x - \sum_{j=i+1}^{-1} z_j b^j \right| \leq \frac{1}{2} \cdot b^{i+1}$$

und nicht lediglich

$$\left| x - \sum_{j=i+1}^{-1} z_j b^j \right| \leq b^{i+1}$$

ist? (Runden statt Abschneiden)

Die Summe $\sum_{j=i+1}^{-1} z_j b^j$ berechnet man übrigens am besten mit dem Horner-Schema:

$$((\ldots(z_{i+1}b^{-1} + z_i) + \ldots + z_{-2})b^{-1} + z_{-1})b^{-1}$$

1.3.2.6 Gleitkommazahlen

Festkommazahlen sind für finanzielle Rechnungen sehr angebracht, solange man mit den Operatoren $+, -, *, /$ auskommt. Für Rechnungen, die Funktionen wie $\log, \exp, \ldots$ benötigen, muß man im allgemeinen auf die Gleitkommazahlen ausweichen. Die Idee für Gleitkommazahlen kommt aus den Ingenieur- und Naturwissenschaften:

$$\mathcal{F}_3^4 = \{\min \mathcal{F} := -9999.999, -9999.998, \ldots, 0, \ldots, 9999.998, 9999.999 =: \max \mathcal{F}\}$$

stellt bis zu 7 gültige Ziffern im Bereich $\mathcal{F} = [\min \mathcal{F}, \max \mathcal{F}]$ zur Verfügung; der Fehler für jede in $\mathcal{F}$ liegende reelle Zahl x durch ihre Approximation $\text{trunc}(x)$ bzw. durch ihre Rundung $\text{round}(x)$ ist absolut

$$\begin{aligned} |x - \text{trunc}(x)| &\leq 10^{-3} \\ |x - \text{round}(x)| &\leq \frac{1}{2} \cdot 10^{-3}, \end{aligned}$$

also z.B. $|\frac{2}{3} - 0.666|$ bzw. $|\frac{2}{3} - 0.667|$. Dagegen kann der relative Fehler

$$\left|\frac{x - \text{trunc}(x)}{x}\right| \text{ bzw. } \left|\frac{x - \text{trunc}(x)}{\text{trunc}(x)}\right|$$

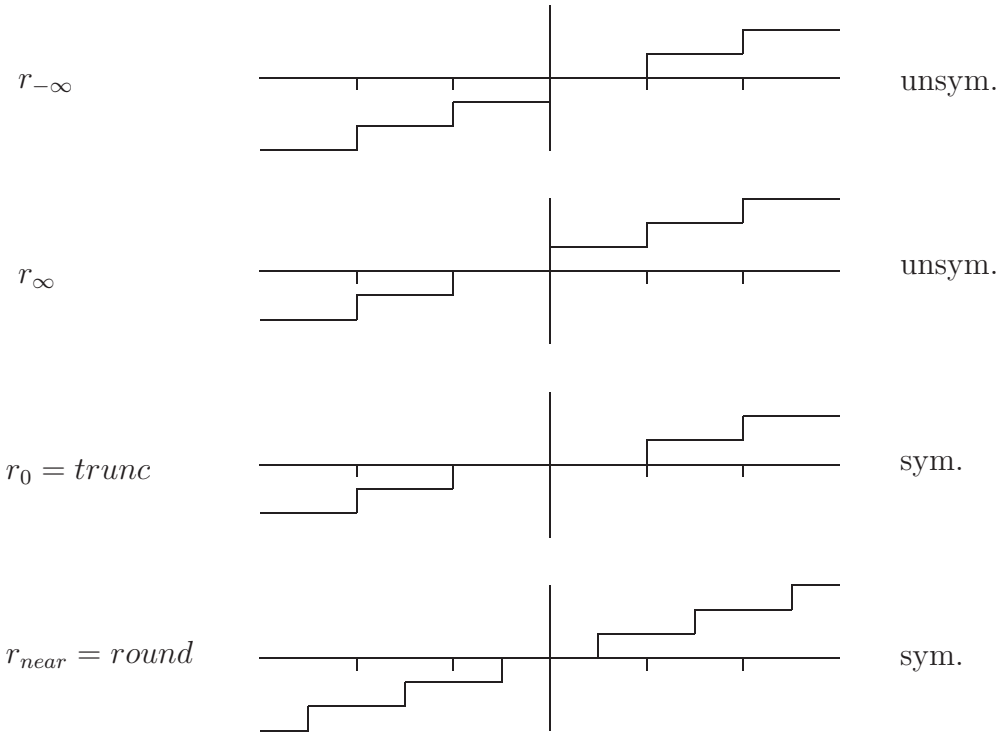
beliebig groß werden, es kann sogar passieren, daß keine Ziffer der Näherung stimmt! Für viele naturwissenschaftlichen Rechnungen reichen etwa obige 7 Stellen (7 *richtige* führende Stellen) aus, aber der abgedeckte Zahlenbereich erstreckt sich über viele Größenordnungen wie μV,...,kV, also 10^{-6}V,...,10^3V, obwohl viele analoge Voltmeter nur $\pm 1\%$ Ablesegenauigkeit haben.

Definition 1.3 *Eine Abbildung* $\circ : \mathbb{R} \to \mathcal{F}_n^\nu$ *mit den Eigenschaften*

$$\begin{aligned} &\textit{Projektion :} && \circ|_{\mathcal{F}_n^\nu} = id \\ &\textit{Monotonie :} && x \leq y \Leftarrow \circ x \leq \circ y \end{aligned}$$

heißt Rundung .

IEEE stellt 4 Rundungen zur Verfügung:



Sei $\tilde{x} \in \mathcal{F}_n^\nu$ eine Näherung von x, z.B. $\tilde{x} = \circ x$ mit $\circ \in \{r_{-\infty}, r_\infty, r_0, r_{near}\}$, so bezeichnet man

$$\Delta\tilde{x} := |\tilde{x} - x|$$

als *absoluten Fehler* des Näherungswertes $\tilde{x}$,

$$\bar{\varepsilon}_{\tilde{x}} := \left|\frac{\tilde{x} - x}{x}\right|$$

für $x \neq 0$ als *relativen Fehler* von $\tilde{x}$. Da x häufig nicht bekannt ist, man also nur weiß, daß $\tilde{x}$ maximal $\Delta\tilde{x}$ von x entfernt ist, benutzt man

$$\varepsilon_{\tilde{x}} := \left|\frac{\tilde{x} - x}{\tilde{x}}\right|$$

für $\tilde{x} \neq 0$ als relativen Fehler. Weiß man (z.B. bei $\tilde{x} = \circ x$ gerundet), daß $\tilde{x}$ genau zwischen zwei Rasterpunkten von $\mathcal{F}_3^4$ liegt, so kann man grob abschätzen:

$$\begin{aligned} \Delta\tilde{x} &\leq 10^{-3} \\ \bar{\varepsilon}_{\tilde{x}} &\leq \frac{10^{-3}}{x} \quad \forall x \neq 0 \\ \varepsilon_{\tilde{x}} &\leq \frac{10^{-3}}{\tilde{x}} \quad \forall \tilde{x} \neq 0 \end{aligned}$$

Für naturwissenschaftliche/technische Anwendungen sind meist etwa nur zwei führende Ziffern sinnvoll. Im Beispiel heißt das

$$(-1)^s \cdot \underbrace{(z_1 10^{-1} + z_2 10^{-2})}_{\text{norm. Mantisse}} \cdot \underbrace{10^e}_{\text{Basis}}$$

mit $s \in \{0,1\}, z_{1/2} \in \{0,\dots,9\}, z_1 \neq 0$ und einem Exponenten $e \in \{-6,-5,\dots,5\}$.

Allgemein ergibt sich

$$(-1)^s \cdot \left(\sum_{i=1}^{N} z_i b^{-i}\right) \cdot b^e, \quad z_1 \neq 0 \qquad (\circ)$$

oder

$$(-1)^s \cdot \left(\sum_{i=0}^{N} z_i b^{-i}\right) \cdot b^e, \qquad z_0 \neq 0$$

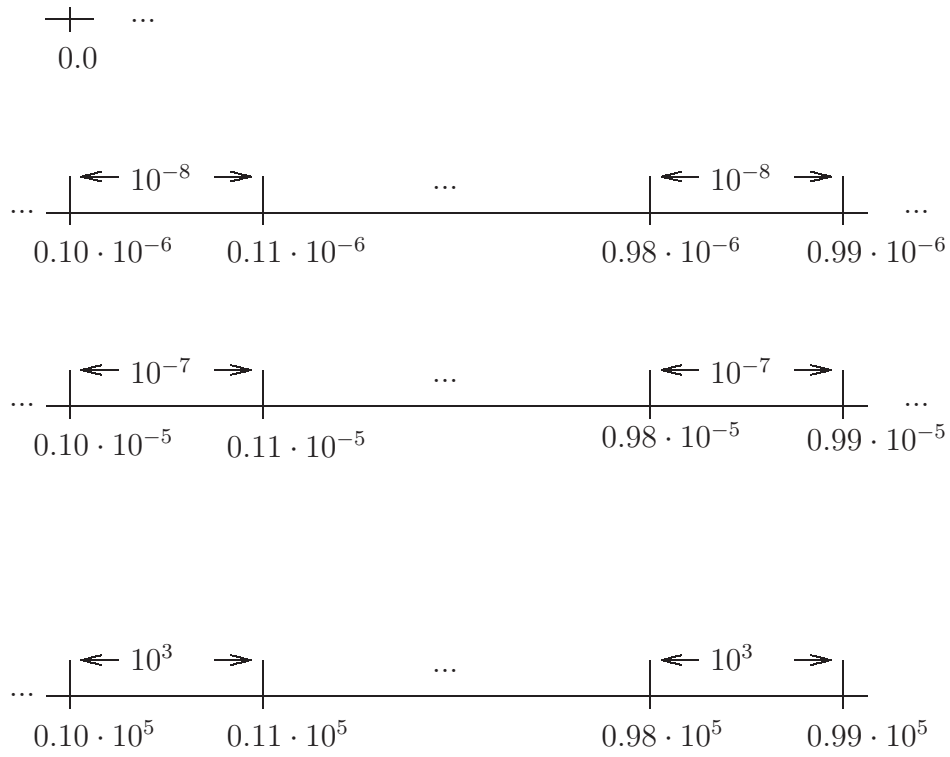
Zur Untersuchung des Zahlenbereichs der durch $(\circ)$ gegebenen Zahlen sei

$$\tilde{\mathcal{G}} := \left\{ (-1)^s \cdot \left(\sum_{i=1}^{N} z_i b^{-i}\right) \cdot b^e | s \in \{0,1\}, z_i \in \{0,1,\dots,b-1\}, z_1 \neq 0, e \in \{e_{min},\dots,e_{max}\} \right\}$$

Man beachte, daß $\tilde{\mathcal{G}}$ <u>nicht</u> die 0 enthält! Man setzt daher $\mathcal{G} := \tilde{\mathcal{G}} \cup \{0\}$, $\max \mathcal{G} := \max \tilde{\mathcal{G}}$ und $\min \mathcal{G} := \min \tilde{\mathcal{G}}$. Für $\tilde{\mathcal{G}}$ erhält man z.B. mit $b = 10, N = 2$ und $e \in \{-6,\dots,5\}$:

$$\begin{array}{rcl}
\{-0.99 \cdot 10^5, -0.98 \cdot 10^5, & \dots & , -0.11 \cdot 10^5, -0.10 \cdot 10^5, \\
-0.99 \cdot 10^4, -0.98 \cdot 10^4, & \dots & , -0.11 \cdot 10^4, -0.10 \cdot 10^4, \\
 & \dots & \\
-0.99 \cdot 10^{-6}, -0.98 \cdot 10^{-6}, & \dots & , -0.11 \cdot 10^{-6}, -0.10 \cdot 10^{-6}, \\
0.10 \cdot 10^{-6}, 0.11 \cdot 10^{-6}, & \dots & , 0.98 \cdot 10^{-6}, 0.99 \cdot 10^{-6}, \\
0.10 \cdot 10^{-5}, 0.11 \cdot 10^{-5}, & \dots & , 0.98 \cdot 10^{-5}, 0.99 \cdot 10^{-5}, \\
 & \dots & \\
0.10 \cdot 10^4, 0.11 \cdot 10^4, & \dots & 0.98 \cdot 10^4, 0.99 \cdot 10^4, \\
0.10 \cdot 10^5, 0.11 \cdot 10^5, & \dots & 0.98 \cdot 10^5, 0.99 \cdot 10^5\}
\end{array}$$

Für die Rundungsfehler ergibt sich im Beispiel mit r_0 folgendes Bild:



1.3.2.7 IEEE-Gleitkommazahlen

IEEE-Gleitkommazahlen im 64 Bitformat sind von der Form

$$x_d \quad = \quad s \cdot 2^e \cdot \sum_{k=1}^{53} f_k \cdot 2^{-k} \tag{1.1}$$

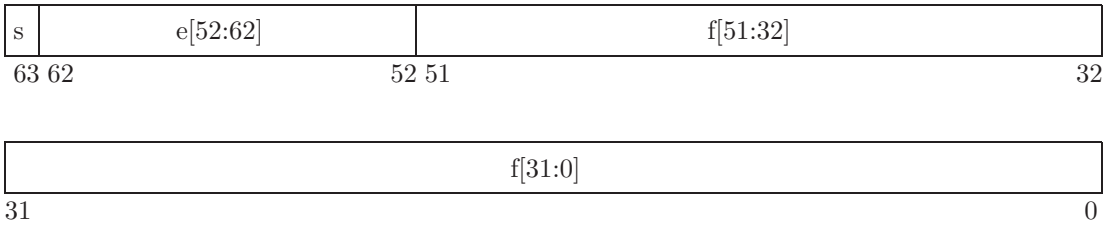
mit genau 53 signifikanten Dualstellen, wobei $s = \pm 1, -1021 \leq e \leq 1024$ und $f_i \in \{0, 1\}$ mit $f_1 \neq 0$ oder

$$x_d \quad = \quad s \cdot 2^{-1021} \cdot \sum_{k=2}^{53} f_k \cdot 2^{-k} \tag{1.2}$$

(inklusive der 0). Zusätzlich sind folgende drei Ausdrücke vordefiniert:

NaN	not a number
$+infty$	$+\infty$
$-infty$	$-\infty$

Man beachte, daß sich die Zahl $\frac{1}{3}$ nicht exakt im IEEE-Format darstellen läßt.
Die Darstellung einer Zahl im IEEE-Format erfolgt dabei durch das folgende Bitmuster:



Werte werden dabei entsprechend der folgenden Tabelle angenommen:

Double-Format Bitmuster	**Wert**
$0 < e < 2047$	$(-1)^s \cdot 2^{e-1023} \cdot 1.f$ (normal nu
$e = 0, f \neq 0$	$(-1)^s \cdot 2^{-1022} \cdot 0.f$ (subnormal
(mindestens ein Bit in f ist ungleich 0)	
$e = 0, f = 0$	$(-1)^s \cdot 0.0$ (signed zero)
(alle Bits in f sind 0)	
$s = 0, e = 2047, f = .000\cdots0$	+INF (positive infinity)
(alle Bits in f sind 0)	
$s = 1, e = 2047, f = .000\cdots0$	-INF (negative infinity)
(alle Bits in f sind 0)	
$s = u, e = 2047, f \neq 0$	NaN (not a number)
(mindestens ein Bit in f ist ungleich 0, u steht für *undefined*)	

- **Maximum positive normal number** — die größte im IEEE Double-Format darstellbare endliche Zahl
- **Minimum positive normal number** — die kleinste im IEEE Double-Format darstellbare positive Zahl, bei der sicher ist, daß nur die letzte Binärstelle unsicher ist.
- **Maximum subnormal number** — die größte im IEEE Double-Format darstellbare Zahl, bei der es möglich ist, daß mehr als die letzte Binärstelle unsicher ist.
- **Minimum positive subnormal number** — die kleinste im IEEE Double-Format darstellbare positive Zahl; hier ist es möglich, daß mehr als die letzte Binärstelle unsicher ist.

Es ergeben sich dabei folgende Beispiele:

Üblicher Name	**Bitmuster (Hex)**	**Äquivalenter Wert**
$+0$	`00000000 00000000`	0.0
-0	`80000000 00000000`	-0.0
1	`3ff00000 00000000`	1.0
2	`40000000 00000000`	2.0
max normal number	`7fefffff ffffffff`	1.7976931348623157e+308
min positve normal number	`00100000 00000000`	2.2250738585072014e−308
max subnormal number	`000fffff ffffffff`	2.2250738585072009e−308
min positve subnormal number	`00000000 00000001`	4.9406564584124654e−324
$+\infty$	`7ff00000 00000000`	Infinity
$-\infty$	`fff00000 00000000`	−Infinity
Not-a-Number	`7ff80000 00000000`	NaN

Im IEEE 32 Bitformat (single precision) gilt analog:

Gleitkomma Single-Precision Format	
s = sign (1 bit) e = biased exponent (8 bits) f = fraction (23 bits) u = undefined	
Normalized value ($0 < e < 255$)	$(-1)^s \cdot 2^{e-127} \cdot 1.f$
Subnormal value ($e = 0$)	$(-1)^s \cdot 2^{-126} \cdot 0.f$
Zero ($e = 0$)	$(-1)^s \cdot 0$
Signalling NaN	$s = u, e = 255\,(\text{max})\, f = .0uu \cdots uu$ (mindenstens ein Bit in $f \neq 0$)
Quiet NaN	$s = u, e = 255\,(\text{max})\, f = .1uu \cdots uu$
$-\infty$	$s = 1, e = 255\,(\text{max})\, f = .000 \cdots 00$
$+\infty$	$s = 0, e = 255\,(\text{max})\, f = .000 \cdots 00$

Dazu noch einmal vergleichend das 64 Bitformat

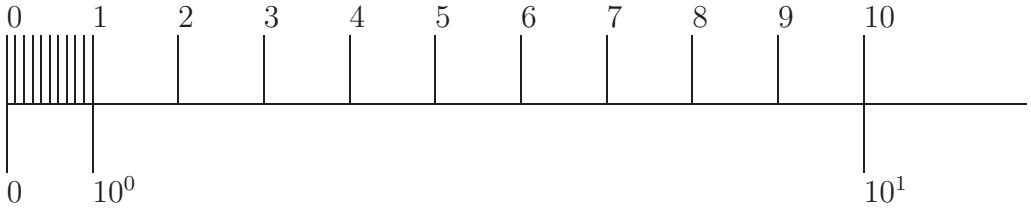
Gleitkomma Double-Precision Format	
s = sign (1 bit) e = biased exponent (11 bits) f = fraction (52 bits) u = undefined	
Normalized value ($0 < e < 2047$)	$(-1)^s \cdot 2^{e-1023} \cdot 1.f$
Subnormal value ($e = 0$)	$(-1)^s \cdot 2^{-1022} \cdot 0.f$
Zero ($e = 0$)	$(-1)^s \cdot 0$
Signalling NaN	$s = u, e = 2047\,(\max)\, f = .0uu \cdots uu$ (mindenstens ein Bit in $f \neq 0$)
Quiet NaN	$s = u, e = 2047\,(\max)\, f = .1uu \cdots uu$
$-\infty$	$s = 1, e = 2047\,(\max)\, f = .000 \cdots 00$
$+\infty$	$s = 0, e = 2047\,(\max)\, f = .000 \cdots 00$

und das 128 Bitformat:

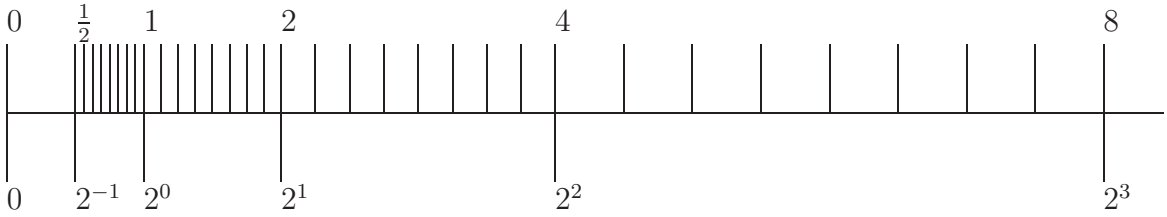
Gleitkomma Quad-Precision Format	
s = sign (1 bit) e = biased exponent (15 bits) f = fraction (112 bits) u = undefined	
Normalized value ($0 < e < 32767$)	$(-1)^s \cdot 2^{e-16383} \cdot 1.f$
Subnormal value ($e = 0$)	$(-1)^s \cdot 2^{-16382} \cdot 0.f$
Zero ($e = 0$)	$(-1)^s \cdot 0$
Signalling NaN	$s = u, e = 32767\,(\max)\, f = .0uu \cdots uu$ (mindenstens ein Bit in $f \neq 0$)
Quiet NaN	$s = u, e = 32767\,(\max)\, f = .1uu \cdots uu$
$-\infty$	$s = 1, e = 32767\,(\max)\, f = .000 \cdots 00$
$+\infty$	$s = 0, e = 32767\,(\max)\, f = .000 \cdots 00$

Zur Veranschaulichung der absoluten Abstände in Gleitkomma-Zahlensystemen:

Decimal Representation



Binary Representation



Die Lücken zwischen darstellbaren Single-Precision Format Gleitkommazahlen:

x	**Nachbarzahl von x in Richtung $+\infty$**	**Lücke**
0.0	1.4012985e-45	1.4012985e-45
1.1754944e-38	1.1754945e-38	1.4012985e-45
1.0	1.0000001	1.1920929e-07
2.0	2.0000002	2.3841858e-07
16.000000	16.000002	1.9073486e-06
128.00000	128.00002	1.5258789e-05
1.0000000e+20	1.0000001e+20	8.7960930e+12
9.9999997e+37	1.0000001e+38	1.0141205e+31

Die üblichen Gleitkomma-Zahlensysteme im Vergleich:

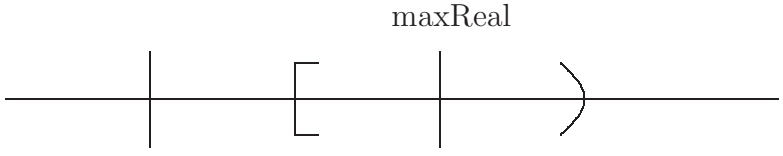
Wertebereich und Genauigkeit				
Format	**Signifikante Binärstellen**	**min positiv normal number**	**max normal number**	**Signifikante Dezimalstellen**
single	24	$1.175\ldots10^{-38}$	$3.402\ldots10^{+38}$	6-9
double	53	$2.225\ldots10^{-308}$	$1.797\ldots10^{+308}$	15-17
double extended (SPARC)	113	$3.362\ldots10^{-4932}$	$1.189\ldots10^{+4932}$	33-36
double extended (x86)	64	$3.362\ldots10^{-4932}$	$1.189\ldots10^{+4932}$	18-21

Format	**Underflow-Schwelle**	
Single	Smallest Normal Number	1.17549435e-38
	Largest Subnormal Number	1.17549421e-38
Double	Smallest Normal Number	2.2250738585072014e-308
	Largest Subnormal Number	2.2250738585072009e-308
Double extended (SPARC)	Smallest Normal Number	3.362103143112093506262677817321752 6e-4932
	Largest Subnormal Number	3.362103143112093506262677817321752 0e-4932
Double extended (x86)	Smallest Normal Number	3.3621031431120935062e-4932
	Largest Subnormal Number	3.3621031431120935059e-4932

IEEE double precision entspricht dem Zahlenbereich $S(2, 53, -1021, +1024)$.
Damit ist die größte darstellbare Zahl

$$maxReal = \sum_{k=1}^{53} 2^{-k} \cdot 2^{1024} = (1 - 2^{-53}) \cdot 2^{1024} = (1 - 2^{-p}) \cdot 2^{emax}$$

und für $[maxReal, +\infty)$ ergibt sich bei der Rundungsart r_{near} und $b = 2$:



1.3.2.8 Beispiele für gute IEEE-Algorithmen

A) **Invarianz von Eigenschaften in $\mathbb{R}$ auch in `double`:**

```
int signum(double x){
  if (x > 0.0)
    return 1;
  else if (x == 0.0)
    return 0;
  else if (x < 0.0)
    return -1;
  else // NaN
    throw "undefined: argument is NaN";
};

double arithMittel(double x, double y){
  if (signum(x) == signum(y))
    return x + (y-x)/2.0;
  else
    return (x+y)/2.0;
};
```

Dieser Algorithmus weist folgende Eigenschaften auf, wobei ohne Einschränkung davon ausgegangen werden kann, daß $x \leq y$ gilt:

(a) Fehler $\leq 1\, ulp$, wobei *ulp* für *unit in last place* steht

(b) Allgemein gilt $x \leq arithMittel \leq y$; falls zwischen x und y ein weiterer Gitterpunkt liegt, so gilt $x < arithMittel < y$.

(c) Es tritt in keinem Fall Over- oder Underflow auf.

Die angegebene zweite Eigenschaft des Algorithmus ist keineswegs selbstverständlich. Es gibt nämlich $x, y; x \neq y$ mit:

$$\frac{x \oplus y}{2} \notin [\min(x,y), \max(x,y)]$$

Man betrachte etwa im 13-stelligen Dezimalsystem die Zahlen

$$\begin{aligned} x &= 0.9\cdots96 \cdot 10^0 \\ y &= 0.9\cdots98 \cdot 10^0 \end{aligned}$$

Dann ist $x \oplus y = 0.19\cdots9 \cdot 10^1$ und somit

$$\frac{x \oplus y}{2} = 0.9\cdots95 \cdot 10^0.$$

Es wird nun der Zahlenbereich $S(10, 2, -5, +5)$ mit $S' := S \cup \{Underflowbereich\}$ und die Operation

$$\frac{x}{2} + \frac{y}{2}$$

behandelt.

(a) Für

$$x = 0.13 \cdot 10^{-5}, \quad y = 0.50 \cdot 10^{-4}$$

erhält man:

$$\begin{aligned}
\frac{x}{2} &= 0.065 \cdot 10^{-5} \overset{in\,S'}{=} \begin{cases} 0.07 \cdot 10^{-5} & \text{bei grad. Underflow} \\ 0.0 & \text{trad.} \end{cases} \\
\frac{y}{2} &= 0.25 \cdot 10^{-4} \\
\frac{x}{2} + \frac{y}{2} &= \begin{cases} 0.257 \cdot 10^{-4} \overset{in\,S'}{=} 0.26 \cdot 10^{-4} & \\ 0.25 \cdot 10^{-4} & \text{trad.} \end{cases}
\end{aligned}$$

(b) Für

$$x = 0.13 \cdot 10^{-5}, \quad y = 0.10 \cdot 10^{-5}$$

erhält man:

$$\begin{aligned}
\frac{x}{2} &\overset{in\,S'}{=} 0.07 \cdot 10^{-5} \\
\frac{y}{2} &\overset{in\,S'}{=} 0.05 \cdot 10^{-5} \\
\frac{x}{2} + \frac{y}{2} &\overset{in\,S'}{=} 0.12 \cdot 10^{-5}
\end{aligned}$$

(c) Sind

$$x = 0.13 \cdot 10^{-5}, \quad y = 0.11 \cdot 10^{-5},$$

so erhält man bei trad. Rechnung $\frac{x}{2} + \frac{y}{2} = 0$ und $x + \frac{y-x}{2} = x$. In S' ist jedoch

$$\begin{aligned}
\frac{x}{2} &\overset{in\,S'}{=} 0.07 \cdot 10^{-5} \\
\frac{y}{2} &\overset{in\,S'}{=} 0.06 \cdot 10^{-5} \\
\frac{x}{2} + \frac{y}{2} &\overset{in\,S'}{=} 0.13 \cdot 10^{-5}
\end{aligned}$$

während

$$\begin{aligned} y - x &\overset{in\,S'}{=} -0.02 \cdot 10^{-5} \\ (y-x)/2 &\overset{in\,S'}{=} -0.01 \cdot 10^{-5} \\ x + (y-x)/2 &\overset{in\,S'}{=} 0.12 \cdot 10^{-5}. \end{aligned}$$

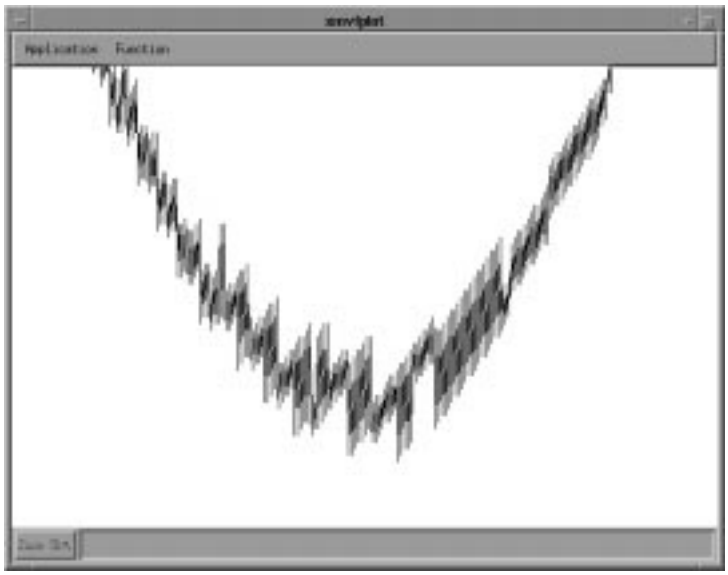
B) **Monotonie** (Voraussetzung für Nullstellensuchverfahren, z.B. Bijektion,...)

Der Graph von

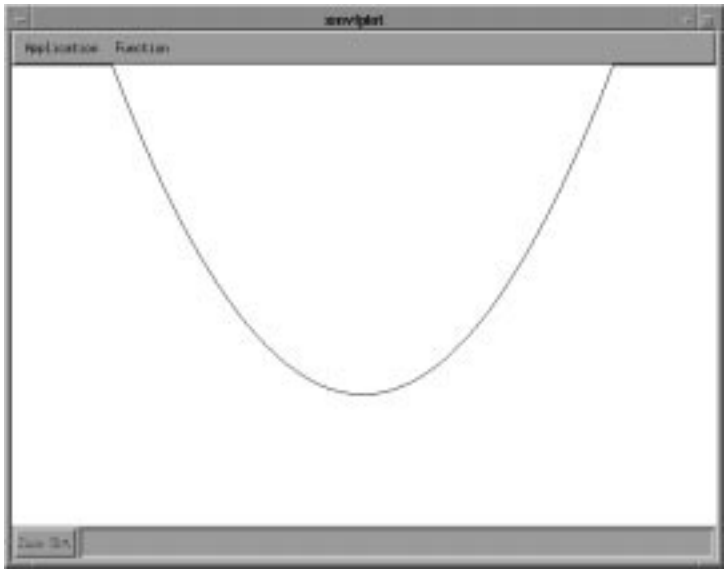
$$p(x) = 170.4x^3 - 356.41x^2 + 168.97x + 18.601$$

liegt in $[1.09160791, 1.09160805] \times [-2 \cdot 10^{-13}, 5 \cdot 10^{-13}]$.

Normale Rechnung mit dem Datentyp `double` liefert



und mittels `ACCURATE` (verifizierte Arithmetik):



In beiden Abbildungen wurde mit 16-stelliger Arithmetik gerechnet bzw. auf ebenso viele Stellen (automatisch) verifiziert. Die beiden positiven Nullstellen des obigen Polynoms liegen sehr nahe zusammen, wie die Zerlegung in Linearfaktoren

$$p(x) = 170.4 \cdot \left(x - \frac{5+\sqrt{35}}{10}\right)\left(x - \frac{5-\sqrt{35}}{10}\right)\left(x - \frac{18601}{17040}\right)$$

und die unter Verwendung von ADE berechneten Einschließungen

$$\frac{5+\sqrt{35}}{10} = 1.091607978309\mathbf{96}_1^2 \quad \text{und} \quad \frac{18601}{17040} = 1.091607981220\mathbf{65}_7^8$$

zeigen.

C) **Lösung einer quadratischen Gleichung (Auslöschung):**

Es wird der Rundungsfehlereinfluß bei 4-stelliger Arithmetik, also bei Gleitkommazahlen mit 4 Mantissenstellen betrachtet. Wir wollen das wohlbekannte Problem

$$ax^2 + bx + c = 0.0$$

nach der Lösungsformel $x_{1/2} = (-b \pm \sqrt{b^2 - 4.0 \cdot a \cdot c})/(2.0 \cdot a)$ mit 4-stelliger Arithmetik (d.h. alle Zwischenergebnisse haben nur 4 signifikante Stellen!) lösen für

$$a = +1.0004, \quad b = -200.01, \quad c = +1.0003 \tag{1.3}$$

1. Fehler: Representation von a, b, c nur vierstellig, also

$$\bar{a} = +1.000, \quad \bar{b} = -200.0, \quad \bar{c} = 1.000$$

Die Rechnung ergibt im 1. Schritt den *2. Fehler*, nämlich einen Rundungsfehler bei der Subtraktion:

$$\bar{b}^2 - 4.0 \cdot \bar{a} \cdot \bar{c} = 40000 - 4.000 = 40000$$

Der 2. Schritt liefert

$$\bar{x}_1 = (200.0 + 200.0)/2.0 = 200.0$$

und im 3. Schritt ergibt sich durch einen *Folgefehler* des 2. Fehlers

$$\bar{x}_2 = (200.0 - 200.0)/2.0 = 0.0$$

Die exakten Lösungen lauten jedoch auf 4 signifikante Ziffern genau

$$\begin{aligned} \bar{x}_1 &= 200.0 \\ \bar{x}_2 &= 0.005000 \end{aligned}$$

Die exakten Lösungen des gestellten Problems (1.3) wären auf 4 Ziffern genau

$$\begin{aligned} x_1 &= 200.0 \\ x_2 &= 0.004999 \end{aligned}$$

gewesen.
Als stabilere Lösungsidee bietet sich für $a \neq 0$ nach der Umformung

$$ax^2 + bx + c = 0 \quad \Leftrightarrow \quad x_{1/2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

im Falle $b \neq 0$ die vorzeichengünstige Wurzel

$$x_1 := \frac{-b + \text{sign}(-b)\sqrt{b^2 - 4ac}}{2a}$$

an. Der entscheidende Vorteil ist, daß die Summanden im Zähler das gleiche Vorzeichen besitzen. Nach dem Satz von Vieta ist dann die zweite Wurzel gegeben durch:

$$x_2 := \frac{c}{a \cdot x_1},$$

wegen $x_1 x_2 = \frac{c}{a}$ und $x_1 + x_2 = -\frac{b}{a}$.

D) **Unvergleichbare IEEE-Zahlen:**

Man betrachte den folgenden Programmausschnitt (Fraktale/Apfelmännchen):

```
⋮
y1 = 0 ;
y2 = 0 ;
for (i=0;i<ITERATIONS;i++)
{
      x1 = y1 ;
      x2 = y2 ;
      y1 = x1*x1-x2*x2+c1 ;
      y2 = 2.0*x1*x2+c2 ;
      n2 = y1*y1+y2*y2 ; // Norm ↑ 2
      if (n2>4.0)
         /* skip */
      else plot(y1,y2) ;
}
```

Geplotten wird, wenn $\|y\|_2 \leq 2.0$, *aber:* Falls y_i sehr groß (nahe $+\infty$), so wird wegen $y_1 = +\infty - \infty + c = NaN$ ebenfalls geplottet! Besser wäre also:

```
      if (n2>4.0)
            ···
      else if unordered(n2,4.0)
            ···
      else ···
```

Beachte also:

```
if (a>b)
      ...
else //a ≤ b ∨ unordered(a, b)
      ...
```

1.3.2.9 Extremsituationen

Ausnahmebedingungen:

IEEE-Ausnahmebedingungen	**Grund des Auftretens**	**Beispiel**	**Ergebnisse ohne Exception-Handler**
Ungültige Operation	Die Operation ist für den auftretenden Operanden nicht definiert. Diese Ausnahmebedingung kann beim x86 auch bei Stack-Fehlern auftreten	$\sqrt{(-x)}$ für $x > 0$ fp_op (signaling_NaN) $0 \cdot \infty, 0/0, \infty/\infty, x$ rem 0 ungeordneter Vergleich unerlaubte Umwandlung	Quiet NaN
Division durch Null	Der Divisor ist gleich Null und der Dividend ist eine endliche Zahl ungleich Null. Allgemeiner: ein exaktes unendliches Ergebnis wird durch eine Operation auf endlichen Operanden geliefert.	$x/0$ für $x \neq 0, \infty$ oder NaN $\log(0)$	∞ mit richtigem Vorzeichen

Tabellenfortsetzung

<table>
<tr>
<td>Überlauf</td>
<td>Ein korrekt gerundetes Ergebnis ist größer als die größte darstellbare Zahl, d.h. der Bereich des Exponenten ist erschöpft.</td>
<td>Double precision:
MAXDOUBLE+1.0e294
exp(709.8)
Single precision:
(float)MAXDOUBLE
MAXFLOAT+1.0e32
expf(88.8)</td>
<td>Abhängig von der Rundungsart (RM) und dem Vorzeichen des Zwischenergebnisses
<table>
<tr><th>RM</th><th>+</th><th>−</th></tr>
<tr><td>RN</td><td>$+\infty$</td><td>$-\infty$</td></tr>
<tr><td>RZ</td><td>+ max</td><td>− max</td></tr>
<tr><td>R−</td><td>+ max</td><td>$-\infty$</td></tr>
<tr><td>R+</td><td>$+\infty$</td><td>− max</td></tr>
</table>
</td>
</tr>
<tr>
<td>Unterlauf</td>
<td>Tritt auf, wenn das exakte Ergebnis von Null verschieden ist, liegt stets im Interval zwischen $\pm 2^{e_{min}}$</td>
<td>Double precision:
nextafter(min_normal,$-\infty$)
nextafter(min_subnormal,$-\infty$)
MINDOUBLE/3.0
exp(-708.5)
Single precision:
(float)MINDOUBLE
nextafterf(MINFLOAT,$-\infty$)
expf(-87.4)</td>
<td>Subnormal oder Null</td>
</tr>
<tr>
<td>Ungenauigkeit</td>
<td>Das gerundete Ergebnis stimmt nicht mit dem exakten Ergebnis überein.</td>
<td>2.0/3.0
(float)1.12345678
log(1.1)
MAXDOUBLE+MAXDOUBLE,
falls kein Überlauf auftritt</td>
<td>Ergebnis der Operation (gerundet), Unter- oder Überlauf</td>
</tr>
<tr>
<td>Unnormalisierter Operand (nur bei x86)</td>
<td>Versuch, eine arithmetische Operation mit unnormalen Operanden durchzuführen</td>
<td>1.0+MINDOUBLE/2.0</td>
<td>Ergebnis der Operation (gerundet), Unter- oder Überlauf</td>
</tr>
</table>

1.3.2.10 IEEE Funktionen:

Funktion	Beschreibung
`math.h`	header file
`copysign(x,y)`	x mit y's Vorzeichen
`fabs(x)`	Absolutbetrag von x
`fmod(x,y)`	Rest bei Division von x durch y
`ilogb(x)`	Exponent zur Basis 2 von x
`nextafter(x,y)`	Nächste darstellbare Zahl von x in Richtung y
`remainder(x,y)`	Rest bei Division von x durch y
`scalbn(x,n)`	$x \times 2^n$

Funktion	Beschreibung
`sunmath.h`	header file
`fp_class(x)`	Klassifikationsfunktion
`isinf(x)`	Klassifikationsfunktion
`isnormal(x)`	Klassifikationsfunktion
`issubnormal(x)`	Klassifikationsfunktion
`iszero(x)`	Klassifikationsfunktion
`signbit(x)`	Klassifikationsfunktion
`nonstandard_arithmetic(void)`	Toggle hardware
`standard_arithmetic(void)`	Toggle hardware

1.3.2.11 IEEE-Werte

IEEE Werte: Single Precision

IEEE Wert	Dezimalwert und hexadezimale Darstellung	C, C++ und FORTRAN
max normal	3.40282347e+38	`r = max_normalf();`
	`7f7fffff`	`r = r_max_normal();`
min normal	1.17549435e−38	`r = min_normalf();`
	`00800000`	`r = r_min_normal();`
max subnormal	1.17549421e−38	`r = max_subnormalf();`
	`007fffff`	`r = r_max_subnormal();`
min subnormal	1.40129846e−45	`r = min_subnormalf();`
	`00000001`	`r = r_min_subnormal();`
∞	Infinity	`r = infinityf();`
	`7f800000`	`r = r_infinity();`
quiet NaN	Nan	`r = quiet_nanf(0);`
	`7fffffff (SPARC, PowerPC,x86)`	`r = r_quiet_nan(0);`
	`7fbfffff (HP700)`	
signaling NaN	NaN	`r = signaling_nanf(0);`
	`7f800001 (SPARC, PowerPC, x86)`	`r = r_signaling_nan(0);`
	`7fc00001 (HP700)`	

IEEE Werte: Double Precision

IEEE Wert	Dezimalwert und hexadezimale Darstellung	C, C++ und FORTRAN
max normal	1.7976931348623157e+308	`x = max_normal();`
	`7feffffff ffffffff`	`x = d_max_normal();`
min normal	2.2250738585072014e−308	`x = min_normal();`
	`00100000 00000000`	`x = d_min_normal();`
max subnormal	2.2250738585072009e−308	`x = max_subnormal();`
	`000fffff ffffffff`	`x = d_max_subnormal();`
min subnormal	4.9406564584214654e−324	`x = min_subnormal();`
	`00000000 00000001`	`x = d_min_subnormal();`
∞	Infinity	`x = infinity();`
	`7ff00000 00000000`	`x = d_infinity();`
quiet NaN	Nan	`x = quiet_nan(0);`
	`7fffffff ffffffff (SPARC, PowerPC,x86)`	`x = d_quiet_nan(0);`
	`7ff7ffff ffffffff (HP700)`	
signaling NaN	NaN	`x = signaling_nan(0);`
	`7ff00000 00000001 (SPARC, PowerPC, x86)`	`x = d_signaling_nan(0);`
	`7ff80000 00000001 (HP700)`	

Function	Range	ulp error	
		f77 1.4	**f77 2.0**
`exp`	$[-665.42,665.42]$	-0.87 ulp	-0.85 ulp
`log`	$[2^{-16.5}, 2^{16.5}]$	$+0.81$ ulp	$+0.81$ ulp
`atan`	$(-\infty, +\infty)$	-0.59 ulp	-0.64 ulp
`sin`	$[0.\pi/2)$	-0.64 ulp	-0.60 ulp
`cos`	$[0.\pi/2)$	$+0.63$ ulp	$+0.63$ ulp

1 ulp $\hat{=}$ 1 unit in the last place

1.3.3 Ausdrücke und Typkonversionen

- boolsche Ausdrücke
- arithmetische Ausdrücke
- Ausdrücke als Anweisungen
- Typkonversionen
  - automatische
  - explizit angeforderte
- Funktionen und Rekursionen

A) Boolsche Ausdrücke:

Aus Wahrheitswerten, Funktionen mit `bool`-Rückgabewert, Vergleichen (`==`, `!=`, `<=`, `>=`, `<`, `>`) mittels der logischen Operationen `!`, `&&`, `||` zusammengesetzte Ausdrücke heißen boolsche Ausdrücke. Sie dienen der Kontrolle von

- Fallunterscheidungen,
- Schleifen und als
- Erfolgsrückgabewerte.

Da `bool` als Spezialfall von `int` angesehen wird, sind boolsche Ausdrücke zugleich auch arithmetische Ausdrücke.

B) Arithmetische Ausdrücke:

Arithmetische Ausdrücke (inkl. boolscher und char-wertiger Ausdrücke) werden gemäß ihrer Klammerung bzw. folgender Hierarchie von Prioritäten abgearbeitet:

Operator	Assoziativität	Beispiel
`::`	links	
`()`, `[]`, `->`, `.`, `++`, `--`, `Typwandlungen`	links	`i++`
`*`, `++`, `--`, `!`, `~`, `&`, `+`, `-`, `casts` , `new`, `delete`	rechts	`++i`
`.*`, `->*`	links	
`*`, `/`, `%`	links	
`+`, `-`	links	
`<<`, `>>`	links	
`<`, `<=`, `>`, `>=`	links	
`==`, `!=`	links	
`&`	links	
`↑`	links	
`|`	links	
`&&`	links	
`||`	links	
`?:`	rechts	
`=`, `+=`, `-=`, `*=`, `andere Zuweisungsoperatoren`	rechts	
`throw`	rechts	
`,`	links	

Bemerkung:

Da der Postfixoperator `++` die Signatur

```
const int int::operator++(int)
```
[3]

hat, ist ein Ausdruck der Form `(i++)++` nicht möglich!

Der Präfixoperator `++` besitzt hingegen die Signatur

```
int& int::operator++(),
```

weshalb Ausdrücke der Form `++(++i)` und selbst `++ ++i` erlaubt sind.

[3] Der `int`-Parameter des Operators `++` in der Signatur dient lediglich dazu, ihn als Postfixoperator zu kennzeichnen.

C) Ausdrücke als Anweisungen:

Ausdrücke können als Anweisungen benutzt werden:

a = b+1;

i++;

a+b;

Dabei wird der Ausdruckswert ignoriert, jeder Nebeneffekt tritt aber auf.

Ausdrücke der Form `expr1, expr2` werden sequentiell nacheinander berechnet. Der Wert des Ausdruckes ist der Wert von `expr2`:

```
sum=0, i=1
```

hat den Wert 1.

D) Typkonversion

Arithmetische Ausdrücke wie etwa

$$x + y$$

haben einen Wert und einen Typ. Wie sehen die Typen aus?

x	y	x+y
int	int	int
short	int	int
⋮	⋮	⋮
short	short	int

In gemischten Ausdrücken `x + y` der Form `short + int` wird eine temporäre Kopie von `x` nach `int` expandiert, um dann z.B.

```
int + int = int
```

zu ergeben.

1) `bool,char,short,enum` → `int` oder `unsigned` (falls `int` nicht repräsentierbar)

2) Falls nach Anwendung von 1) noch ein gemischter Ausdruck existiert, so wird gemäß
`int` → `unsigned` → `long` → `unsigned long` → `float` → `double` → `long double`
eine Typanpassung durchgeführt.

Beispiele:

```
char c;
short s;
int i;
unsigned u;
long l;
float f;
double d;
```

`d - s / i`	`double`
`u * 3.0 - d`	`double`
`c + 1`	`int`
`c + 1.0`	`double`
`u * 3 - i`	`unsigned`
`3 * s * l`	`long`
`d + s`	`double`
...	...

Bei der Wertzuweisung „`=`“ wird ebenfalls nötigenfalls eine automatische Typkonversion stattfinden:

```
d = i;
i = d;    // systemabhaengiges Verhalten
```

Analoges gilt bei der Auswertung von Funktionsargumenten bei Funktionsaufrufen.

Einzig bei `catch`-Anweisungen von `try`-Blöcken findet keine automatische Typkonversion statt, weshalb etwa

```
...
try{
   ...
   if (nenner == 0) throw "Fehler: Nenner == 0!";
   ...
}catch(const string& err){
   cerr << endl << err << endl;
   exit 1;
}
...
```

die Exception vom Typ `const char*` nicht abfängt!

Um einen Konstruktor vor der Benutzung bei der automatischen Typkonversion auszuschließen, kann man ihn als `explicit` kennzeichnen:

```
class Jahr{
  int j;
public:
  explicit Jahr(int i): j(i){};
  ...
};
...
  Jahr d1, d2;
  d1 = Jahr(2001);
  // d2 = 2001;         ist nicht möglcih!
...
```

Explizite Typkonversion kann angefordert werden mittels der Typkonversionsfunktionen/-operatoren:

```
static_cast<double>(i)
```

(Früher `(double)i` oder `double(i)`.)

`const_cast<int>(i)` erlaubt bei Bedarf ein konstantes Objekt `i` an Stellen zu benutzen, wo nichtkonstante Objekte benötigt werden!

```
////////////////////////////
// Datei:  cast3.cc
// Version: 1.0
// Zweck:  cast
// Autor:  Hans-Juergen Buhl
// Datum: 17.09.1998
////////////////////////////

#include <iostream>

using namespace std;

void f(int& i)
{
  cout << i << endl;
};

int main()
{
  const int j(5);

// f(j);
//
//  Warning (Anachronism): Formal argument i of type int&
//        in call to f(int&) is being passed to const int.
//

  f(const_cast<int&>(j));

  return 0;
}
```

Benötigen Sie eine automatische oder auch lediglich nur explizit benutzbare Typkonversion in einen Typ, den Sie nicht selbst als Klasse programmiert haben, können Sie wie folgt vorgehen:

```
class rationalNumber
{
  long Z, N;
public:
  rationalNumber(long z = 0, long n = 1) : Z(z), N(n) { kuerze(); };
  ...
```

```
  operator double() const{
    return Z/(N*1.0);
  };
  ...
}
```

Der neue Typkonversionsoperator `double` ist dann ab sofort zur Typwandlung von Exemplaren von `rationalNumber` nach `double` benutzbar als:

```
double d, s, t, w;
rationalNumber r1(7,10) , r2(5,14);

d = r1;
//                          oder
s = (double) r2;
//                          oder
t = double(r1 + r2);
//                          oder
w = r1.operator double();
...
```

Aufgaben:

Teste:

```
int i; double y;
  ⋮
y = i / static_cast<double>(7);
  ⋮
y = i / 7.0;
  ⋮
y = i / 7;
```

E) Funktionen und Rekursionen:

Funktionen wie zum Beispiel

$$\texttt{odd:}\ \mathbb{Z} \to \mathbb{B} := \{\texttt{true}, \texttt{false}\}$$

sind in `C++` mittels

```
bool odd( int i )
{
  return ( i % 2 != 0 );
}
```

deklarierbar.

Sonderfälle:

1) Funktionen ohne Rückgabewert:

```
void print_square(int i)
{
  cout << i * i << endl;
}
```

2) Funktionen ohne Definitionsbereich / Argument:

```
void print_hello()
{
  cout << "Hello\n";
}
```

3) Referenz-Parameter

```
void swap(int& i, int& j)
{
  int temp(i);
  i = j; j = temp;
}
```

4) Funktionen können als `inline` deklariert werden (Effektivitätssteigerung):

```
inline odd(int i)
{
  ...
}
```

5) Funktionen können Argumente mit Defaultwerten besitzen:

```
inline power(int i, int exp = 2)
{
  ...
}

...

int main()
{
  cout << power(5) << endl;      // aequivalent zu power(5,2)
  cout << power(5,3) << endl;
}
```

Aufgabe:

Teste alle skizzierten Funktionen.

Funktionen können gleiche Namen haben, sofern sie sich durch den Definitionsbereich unterscheiden:

```
double power(double x, int exp);
int power(int i, int exp);
int power(int i);
     ⋮
```

Dies bezieht sich auch auf Operatoren:

```
////////////////////////////////////
// Datei  :  helloDate.cc
// Version : 2.0
// Zweck:    Ausgabe einer Nachricht;
//           <<-Operator, >>-Operator
// Autor:    Hans-Juergen Buhl
// Datum:    17.09.1998
////////////////////////////////////


#include  <iostream>
#include  <string>

using namespace std;

class Nachricht {

  string Text;       // nicht mehr const wegen ***

public:

  Nachricht(const string& str = "") : Text(str) { };

  void print() const { cout << Text; };

  friend ostream& operator << (ostream&, const Nachricht&);

  // friend istream& operator >> (istream&, Nachricht&); nicht noetig

};

ostream& operator << (ostream& os, const Nachricht& na)
{
  os << na.Text;
  return os;
}

istream& operator >> (istream& is, Nachricht& na)
```

```
{
  // string te;
  // is >> te;
  //    holt nur ein Wort

  char te[80];
  is.getline(te, 80);
  Nachricht name(te);
  na=name;                  // ***
  //    ohne string

  // string te;
  // is.getline(te);
  // Nachricht name(te);
  // na = name;
  //     funktioniert   n o c h   nicht

  return is;
}

int main()
{
  Nachricht Begruessung("Willkommen zum Programmierkurs");
  Begruessung.print();

  cout << "-------------------------------------\n\n";
  cout << Begruessung << endl;


  cout << "-------------------------------------\n\n";
  cout << "Bitte einen neuen Nachrichtentext eingeben: ";
  Nachricht InText;
  cin >> InText;
  cout << InText << endl;
  cout << endl;

  return 0;
}
```

Einfache Rekursionen

Funktionen / Methoden können sich selbst direkt oder indirekt aufrufen, zum Beispiel bei Quicksort:

```
void quicksort(T* from, T* to)
{
  T* mid;
  if (from < to−1) {
    mid = partition(from,to);
    quicksort(from,mid);
    quicksort(mid+1,to);
  }
};
```

Die Funktion `partition(from, to)` bestimmt dabei `mid` so, daß alle Elemente von `from` bis `mid` kleiner als alle Elemente von `mid+1` bis `to` sind!

Andere Beispiele:

- Binäre Suche
- „recursive decent“-Parser ...

Schlechtes Beispiel: Fakultät – hier ist eine einfache Schleife viel angebrachter!

Vorsicht:
Achten Sie immer darauf, daß nach **endlich** vielen Selbstaufrufen kein Selbstaufruf mehr stattfindet!

Aufgabe:

a) Wann ist die binäre Suche beendet?

b) Programmieren Sie die binäre Suche!

Die Ackermann-Funktion

```
#include <iostream>
#include <exception>

using namespace std;

int ackermann(int n,        // n >= 0
              int m         // m >= 0
              ){            // Ergebnis >= 0

  if (n<0) throw n;
  if (m<0) throw n;
  if (n==0) return (m+1);
  if (m==0) return ackermann(n-1,1);
  return ackermann(n-1,
                   ackermann(n,m-1));
}

int main(){
  cout << ackermann(1,1) << endl;
  cout << ackermann(1,2) << endl;
  cout << ackermann(2,2) << endl;
  cout << ackermann(3,3) << endl;
}
```

Aufgabe: Was passiert bei „zu großen“ Ergebniswerten?

Wechselrekursionen

Wechselrekursionen treten auf, wenn mehrere Funktionen sich gegenseitig aufrufen.
Müssen sich zwei Funktionen gegenseitig aufrufen, so kommt es zum Konflikt mit der Regel, daß jedes C++-Objekt vor der ersten Benutzung definiert sein muß.
Dies Problem kann dadurch gelöst werden, daß man zunächst eine Funktion **nur** deklariert (ohne Funktionsimplementierung). Dann kann diese schon benutzt werden; die Funktionsdefinition muß jedoch nachgeliefert werden:

```
double f1(double);  // Deklaration f1

double f2(double d) // Definition f2
{
  // ...
  return f1(d);
}

double f1(double x) // Definition von f1
{
  // ...
  x = f2( x / 2.0 );
  // ...
}
```

Auch bei der Wechselrekursion bitte Vorsicht vor Endlosschleifen!

Sei als Beispiel etwa:

$$\forall n > 0$$

$$\begin{aligned} f(n) &= n - m(f(n-1)) \\ m(n) &= n - f(m(n-1)) \end{aligned}$$

$$\begin{aligned} f(0) &= 1 \\ m(0) &= 0 \end{aligned}$$

(vgl. D.R. Hofstädter: Gödel, Escher, Bach, S. 148 f., Klett-Cotta-Verlag, 1985)

```
////////////////////////////////////
// Datei:  wechselRek.cc
// Version: 1.0
// Zweck:  Deklaration, Definition
// Autor:  Hans-Juergen Buhl
// Datum: 13. Nov. 2000
////////////////////////////////////

#include <iostream>

using namespace std;

long m(long);

long f(long n)
{
  if (n == 0)
    return 1;
  else if (n > 0)
    return (n - m(f(n-1)));

  throw n;
}

long m(long n)
{
  if (n == 0)
    return 0;
  else if (n > 0)
    return (n - f(m(n-1)));
  throw n;
}

int main()
{
  for (int i=0; i < 1001; i++)
    cout << i << '\t' << f(i) << '\t' << m(i) << endl;
}
```

1.4 Die Klasse `<complex>`

Beispiel:

```
#include <iostream>
#include <complex>

using namespace std;

int main()
{
  complex<double> c1(1.0);
  complex<double> i(0.0, 1.0);

  cout << c1 * i << endl;

  typedef complex<double> complexd;

  complexd c2(c1);
  const complexd Eins(1.0);

  cout << c2 * Eins << endl;
  cout << exp(Eins + i) << endl;

  return 0;
}
```

`complex` ist eine sogenannte Template-Klasse, also eine generische Klasse, deren Skalarbereich man aus

- `float`
- `double`
- `long double`

wählen kann.

1.5 Die Klasse <string> — Iteratoren

Die Klasse <string> besitzt folgende Methoden:

Konstruktoren	`string()`
	`string(const char*)`
	`string(const string&)`

Destruktor	`~string()`

Klassenmethoden (member functions)	`int length()`
	`string& operator= (const string& s)`
	`string& operator= (const char* p)`
	`string& operator+= (const string& s)`
	⋮
	`char& operator[](int)`
	`const char& operator[](int) const`
	⋮

friend-Funktionen:	`ostream& operator << (ostream &o, const string& s);`
	`istream& operator >> (istream &i, string &s);`
	`bool operator== (const string& s1, const string& s2)`
	⋮

Sie ist ein Prototyp für sogenannte Containertypen, hier für eine angeordnete Sammlung — d.h. eine Folge — von chars.

Ein Beispiel zur Textbearbeitung mit Hilfe von string-Methoden:

```
////////////////////////////////
// Datei:   satz.cc
// Version: 1.0
// Zweck:   string demo
// Autor:   Hans-Juergen Buhl
// Datum:   17.09.1998
////////////////////////////////

#include   <iostream>
#include   <string>

using namespace std;

int main()
```

```
{
  string Satz;

  char* pos;

  Satz = "Eskimos beschreiben Schnee auf ";
  Satz += "23 verschiedene Weisen";

  pos = Satz.begin();
  while (pos < Satz.end()) {
        do {
            cout << *pos;
            ++pos;
        } while ((pos < Satz.end()) && (*pos != ' '));

        // (*pos == ' ') || (pos == Satz.end())

        if (pos < Satz.end()) ++pos;
        cout << endl;
  };

  int anz_e(0);
  for (char* i(Satz.begin()); i < Satz.end(); i++)
        if (*i == 'e') anz_e++;
  cout << anz_e << " e-Vorkommnisse" << endl;

  anz_e = 0;
  for (int j=0; j < Satz.length(); j++)
        if (Satz[j] == 'e') anz_e++;
  cout << anz_e << " e-Vorkommnisse" << endl;

  return 0;
}
```

Statt `char* pos` kann (und sollte) auch `string::iterator pos` benutzt werden. `string.begin()` und `string.end()` liefert einen Iterator. Auf Iteratoren ist die Methode `++` definiert.

Aufgaben:

a) Testen Sie.

b) Schreiben Sie eine Variante, die alle vorkommenden Buchstaben zählt und eine Statistik ausgibt.

Eigentlich ist `string` ebenfalls ein template-Typ (vgl. 1.4), der vorinstantiiert ist.

Zugriff auf `string`-Elemente (`char`'s) über Index bzw. Iteratoren:

	Index	Iterator
akt. Position initialisieren	`int i(0);`	`className::iterator pos(x.begin());`
akt. Position verschieben	`i++;`	`pos++;`
Abbruch	`i < x.length()`	`pos < x.end()`
Element an akt. Stelle	`x[i]`	`*pos`

Bemerkung: `*pos++` ist identisch mit `*(pos++)`

Iteratoren sind die geeigneten abstrakten Hilfmittel, um Exemplare von Containertypen Element für Element (jedes genau einmal) zu bearbeiten.

Aber auch in anderen Fällen bieten sich **iteratorähnliche** Klassen als geeignete Hilfsmittel an: Im Falle der vier verschiedenen Versionen zur Primzahlberechnung (vergleiche Übungsblätter 9, 11, 13 und 14) hätte die Hauptapplikation — die Funktion `main()` — völlig ungeändert bleiben können, hätte man einen „Iterator“ `PrimeCandidates` nach folgendem Muster genutzt:

```
#include <iostream>

using namespace std;

...
int main(){

  int n;
  PrimeCandidates t;  // TeilerKandidat
  PrimeCandidates p;  // PrimzahlKandidat

  cout << "Bitte n eingeben: ";
  cin >> n;
  if (n >= 2)
    {
      p.reset();
      cout << *p++ << endl;
      while (*p <= n)
        {
```

```
          t.reset();
          while ((*p % *t != 0) && (*t * *t <= *p)){
            t++;
          };
          if (*t * *t > *p)
            cout << *p << endl;
          p++;
        }
    }

  return 0;
}
```

Die erste (ineffektivste) Version der Primzahlsuche gemäß

```
class PrimeCandidates{
  long pos;
public:
  PrimeCandidates(): pos(0){};

  void reset(){ pos = 0; };

  long operator*() const{
    if (pos == 0) return 2;
    return (2*pos+1);
  };

  PrimeCandidates operator++(int){
    PrimeCandidates temp = *this;
    pos += 1;
    return temp;
  };

  PrimeCandidates begin() const { return PrimeCandidates(); };
};
```

hätte dann jeweils nur durch die jeweils effektivere ersetzt werden müssen.

1.6 Eigene generische Deklarationen

a) Funktionen:

Um Funktionen nicht für alle möglichen Parametertyp-Kombinationen mittels „cut and paste“ und anschließendem (automatischen) Suchen und Ersetzen selbst codieren zu müssen, gibt es `template`'s:

```
template <class T>
void swap(T& a, T& b)
{
  T temp(a);
  a = b;
  b = temp;
}

...

int main()
{
  int x,y;
  double s,t;
  ...
  swap(x,y);
  ...
  swap(s,t);
  ...
  swap(x,s)
  ...
}
```

- Auch bei Templates werden automatische Typkonversionen durchgeführt!
- Sortieren, Suchen, ... sollten generisch zur Verfügung stehen.

b) Klassen:

Auch eigene Klassen können Sie generisch (als Musterklasse) implementieren (wie zum Beispiel die Standardklasse `complex`):

```
template <class T, int n=3>
class Strecke{
  T start_pkt[n];
  T end_pkt[n];

public:
  ...
  void zeichne();
  void drehe(double Winkel);
  ...
}

template <class T, int n=3>
void Strecke<T,n>::zeichne()
{
 ...
}

...

int main()
{
  Strecke<double,4> a,b;
  Strecke<int>    x,y;
}
```

Bemerkung:
Insbesondere Containerklassen, zum Beispiel:

- Warteschlangen,
- Kellerspeicher,
- Listen,
- Bäume,
- Graphen,

sind gute Kandidaten für Template-Klassen.

Bemerkung: Template-Parameter können nie Templates sein.

Kapitel 2

Weitere Klassen

2.1 Vektoren und Matrizen in C

Vektoren und Matrizen nach C-Syntax:

$v \in double^n$ `double v[n]` v_i `v[i]`	$M \in double^{m \times n}$ `double M[m][n];` M_{ij} `M[i][j]`
`double v[3]={1,2,3};`	`double M[2][3] = { {1,2,3}, {4,5,6} };` `double M[2][3] = { 1,2,3,4,5,6 };`
`double Norm(const double v[], const int n)`	`double Norm(const double M[][n], const int m)`

Dynamische Vektoren nach C-Syntax:

double *v;
v = **new double**[n];

...

delete []v;

2.2 Dynamisch angeforderter Speicher (`new`, `delete`)

```
...
{
 double y(1.5);
 double *x_adr;
 ...
 x_adr = new double(1.1);
 ...
 x_adr = new double(1.2);
 ...
}
```

Dieses Code-Stück verursacht ein „Speicherleck“!

Bemerkung:

a) Durch „`new`“ angeforderter Speicher bleibt solange bestehen, bis er durch „`delete`“ wieder freigegeben wird:

```
delete x_adr;
```

b) Vorsicht:

`double *x, y;` $\Longleftrightarrow$ `double *x; double y;`

Richtig ist

```
double *x, *y;
```

c) An den Wert an der Stelle, dessen Adresse im Zeiger steht, kommt man durch Dereferenzieren :

```
cout << *x_adr;
```

d) Überprüfen Sie bei alten `C++`-Compilern immer den Erfolg von `new`:

```
x_adr = new double(1.5);
if (x_adr == 0) throw "Fehler: Speicherplatzmangel für new!";
...
```

2.3 Dynamische Felder:

```
int vecdim;
cin >> vecdim;
if (vecdim < 1)
{
  cerr << "vecdim too small\n";
  exit(1);
}

...

double* x;
x = new double[vecdim];
if (x==0) throw "Fehler: Nicht genügend Speicherplatz frei!";

x[0] = 1.5;

  ...

delete []x;

  ...
```

Vorsicht:

A) In `int *x[10]` – als `(int *x)[10]` – bezeichnet x ein Feld von 10 `int*`-Elementen.

B) In `int (*x)[10]` ist x ein Zeiger auf ein 10-dim. Feld mit `int`-Elementen.

Bemerkung:
Wird innerhalb von Klassen durch einen Konstruktor `new` benutzt, **muß** ein Destruktor mit `delete` programmiert werden.

```
////////////////////////////////
// Datei:   dynvec.cc
// Version: 1.0
// Autor:   Hans-Juergen Buhl
// Datum: 17.09.1998
////////////////////////////////

#include  <iostream>
#include  <cassert>
#include  <cstdlib>

using namespace std;

int main()
{
  static const int m(3);
  assert( m >= 1 );

  // double* x(new double[m]);
  // oder:

  double* x;
  x = new double[m];
  // if (x==0) throw "Fehler: Nicht genügend Speicherplatz!";

  x[0] = 1.0;
  cout << x[0] << endl;
  // ...
  delete []x;

  double* y[m];
  y[0] = new double(1.5);
  cout << *y[0] << endl;
  delete y[0];

  // double (*z)[m];
  // z = new double[m];
  // Error: Cannot assign double* to double(*)[3]

  return 0;
}
```

Eine Lowlevel-Implementierung von `Nachricht` benutzt dynamische char-Vektoren:

```
#include <iostream>

using namespace std;

class Nachricht {

  char* Text;

public:

  // Nachricht() {};

  Nachricht(const char* t) { Text = new char[strlen(t)+1]; strcpy(Text, t); };

  ~Nachricht() { delete []Text; }

  void print() { cout << Text; };

};

int main()
{
  Nachricht Begruessung("Willkommen zum Programmierkurs");

  Begruessung.print();
  cout << endl;

}
```

In C++ sollte lieber der (dynamische) `string`-Typ benutzt werden!

2.4 Klasenattribute und -methoden

Klassenattribute und -methoden werden in C++ mit dem Keyword `static` gekennzeichnet; sie existieren pro Klasse **nur** einmal und nicht wie nicht-`static` Attribute/Methoden je Klassenexemplar einmal:

```
class numeriertesXXX{

  // ...
  static int Zaehler;

public:

  numeriertesXXX(){
    Zaehler++; // ...
  }
  static int getZaehler() const { return Zaehler; }

}

int numeriertesXXX::Zaehler(0); // Initialisierung des Klassenattributes

// ...

int main() {
  numeriertesXXX s1,s2;
  // ...
  cout << "Objektanzahl= " << numeriertesXXX::getZaehler() << endl;
  // ...
}
```

Klassenmethoden dürfen nur auf Klassenattribute zugreifen, da sie unabhängig von der Existenz von Exemplaren der Klasse aufgerufen werden.

Eine wichtige Nutzanwendung von Klassenattributen ist die klassenweite Dimensionierung eines Vektor-Attributs:

```
...
class ratNumErr{
  static const int MAXNUM(2);
  string Causes[MAXNUM];
  const int aktIndex; // 0..MAXNUM
public:
  ratNumErr(const int i = 0): aktIndex(i)
  {
                           Causes[0]="unspecified error in: rationalNumber";
                           Causes[1]="Nenner == 0 in Konstruktor rationalNumber";
  };
  string what(){ return Causes[aktIndex];}
};
...
class rationalNumber
{
  long Z, N;
public:
  rationalNumber(long z = 0, long n = 1) : Z(z), N(n) {
    if (n == 0) throw ratNumErr(1);
    kuerze();
  };
  ...
};
...
int main()
{
 try{
  rationalNumber r1(7,10) , r2(5,14);
  ...
 }
 catch (ratNumErr& err) {
    cerr << err.what() << endl;
 }
 catch (exception& e){
    cerr << e.what() << endl;        // z.B.: Arithmetic Exception
 }
```

2.5 Vektoren als (sichere) Klasse

vektor
low high double *v;
vektor(const int h, const int l=1, const double d=0.0) vektor(const double x[], int n) ~vektor() vektor& operator= (const vektor&) double& operator() (const int i) double operator() (const int i) const int lo() const int hi() const friend ostream& operator<< (ostream&, const vektor&) double Norm() const ... bool operator== (const vektor&) const ... vektor operator+ (const vektor&) const ...

```
////////////////////////////////
// Datei:  vektor1.cc
// Zweck:  Vektor als Klasse
// Autor:  Hans-Juergen Buhl
////////////////////////////////

#include  <iostream>

using namespace std;

class vektor{

  const int low;       // v(low..high)
  const int high;
  double* v;           // Startadresse fuer dyn. verwaltetes Exemplar

public:

  vektor(const int h, const int l = 1, const double d = 0.0); // v(l..h) = d
  vektor(const double x[], int n);                             // v(1..n) = x[0..n-1]

  ~vektor(){ delete []v; };

  double& operator()(const int i);
  double operator()(const int i) const;

  int lo() const { return low; };
  int hi() const { return high; };

  friend ostream& operator<<(ostream& os, const vektor& v);

};

vektor::vektor(const int h, const int l, const double d) : low(l), high(h)
{
  int size(h-l+1);
  if (size < 1) throw "falsche Vektor-Länge in Konstruktor";
  v = new double[size];
  // if (v == 0) throw "kein freier Speicherplatz mehr verfügbar";
  for (int j=0; j < size; j++)
    v[j] = d;
};

vektor::vektor(const double x[], int n) : low(1), high(n)
{
  if (n < 1) throw "falsche Vektor-Länge in Konstruktor";
  v = new double[n];
  // if (v == 0) throw "kein freier Speicherplatz mehr verfügbar";
  for (int i=0; i < n; i++)
```

```
    v[i] = x[i];
};

double& vektor::operator()(const int i)
{
  if ( (i <low) || (i > high) ) throw "Indexverletzung bei Komponentenzugriff";
  return v[i−low];
};

double vektor::operator()(const int i) const
{
  if ( (i <low) || (i > high) ) throw "Indexverletzung bei Komponentenzugriff";
  return v[i−low];
};

ostream& operator<<(ostream& os, const vektor& w)
{
  os << " ( ";
  os << w(w.lo());
  for (int i=w.lo()+1; i<= w.hi(); i++)
    os << " , " << w(i);
  os << " ) ";
  return os;
};

int main()
{
  vektor y(5, 1, 3.0);
  cout << y.lo() << " " << y.hi() << endl;
  cout << y << endl;
  for (int i=y.lo(); i <= y.hi(); i++)
    y(i) = i*2;
  cout << y << endl;

  vektor x(8, 2);
  cout << x.lo() << " " << x.hi() << endl;
  cout << x << endl;
  for (int k=x.lo(); k <= x.hi(); k++)
    x(k) = k*2;
  cout << x << endl;

  double zh[] = { 1.0, 3.0, 2.0, 4.0 };
  vektor z(zh, 4);
  cout << z.lo() << " " << z.hi() << endl;
  for (int j=z.lo(); j <= z.hi(); j++)
    z(j) *= z(j);
  cout << z << endl;
}
```

2.6 Matrizen als (sichere) Klasse

matrix
low1, high1 low2, high2 double* v
matrix(...) matrix(const matrix&) ~matrix() matrix& operator= (const matrix&) double& operator() (const int i, const int j) double operator()(const int, const int) const int lo1(), hi1(), lo2(), hi2() ...

```
////////////////////////////////
// Datei:   matrix.cc
// Zweck:  matrix als Klasse, ==
// Autor:  Hans-Juergen Buhl
////////////////////////////////

#include      <iostream>

using namespace std;

class matrix{

  int low1;              // v(low1..high1, low2..high2)
  int high1;             // high1 >= low1
  int low2;
  int high2;             // high2 >= low2

  double* v;             // Startadresse fuer dyn. verwaltetes Exemplar

public:

  matrix(const int h1, const int l1, const int h2, const int l2,
         const double d = 0.0);

  ~matrix(){ delete []v; };

  int lo1() const { return low1; };
  int hi1() const { return high1; };
  int lo2() const { return low2; };
  int hi2() const { return high2; };
```

```
  double& operator()(const int i, const int j);
  double operator()(const int i, const int j) const;

};

matrix::matrix(const int h1, const int l1, const int h2, const int l2,
               const double d) : low1(l1), high1(h1), low2(l2), high2(h2)
{
  int size1(h1-l1+1);
  if (size1 < 1) throw "falsche Vektor-Länge in Konstruktor";

  int size2(h2-l2+1);
  if (size2 < 1) throw "falsche Vektor-Länge in Konstruktor";

  v = new double[size1*size2];
  // if (v == 0) throw "kein freier Speicherplatz mehr verfügbar";

  for (int j=l1; j <= h1; j++)
    for (int k=l2; k <= h2; k++)
      v[(j-l1)+(k-l2)*size1] = d;
};

double& matrix::operator()(const int i, const int j)
{
  if ( (i < low1) || (i > high1)) throw "Indexverletzung bei Komponentenzugriff";
  if ( (j < low2) || (j > high2)) throw "Indexverletzung bei Komponentenzugriff";
  int size1(high1-low1+1);

  return v[(i-low1)+(j-low2)*size1];
};

double matrix::operator()(const int i, const int j) const
{
  if ( (i < low1) || (i > high1)) throw "Indexverletzung bei Komponentenzugriff";
  if ( (j < low2) || (j > high2)) throw "Indexverletzung bei Komponentenzugriff";
  int size1(high1-low1+1);

  return v[(i-low1)+(j-low2)*size1];
};

int main()
{
  matrix m1(5,1,7,0,7.0);

  for (int i=m1.lo1(); i <= m1.hi1(); i++)
    for (int j=m1.lo2(); j <= m1.hi2(); j++)
      cout << i << " " << j << " : " << m1(i,j) << endl;
}
```

2.7 Numerische lineare Algebra mit Vektoren und Matrizen

Zu Anwendungen der linearen Algebra benötigt man noch Methoden, die auf Vektoren und Matrizen operieren:

```
class matrix{

    //...

};

class vektor{

    // ...

};

matrix operator*(const matrix&, const matrix&);

vektor operator*(const matrix&, const vektor&);

matrix operator||(const vektor& x, const vektor& y); // dyadisches Produkt

// ...
```

2.8 Numerische Lineare Algebra und die STL

STL = Standard Template Library

Die STL stellt u.a. folgende Klassen zur Verfügung:

Klasse	Zweck
vector	lineare Felder für nichtnumerische Zwecke
valarray	lineare Felder als Vektoren, Matrizen für numerische Zwecke

Ein kurzer Überblick:

```
valarray<double> x1;
valarray<double> x2(1000);  // 1000 Komponenten

min, max
* / % + - acos ... komponentenweise
resize, shift
[]
slice_array  wie z.B. raw, column, ...
...
```

valarray's werden i.a. nicht direkt sondern zur Implementierung von numerische Bibliotheken benutzt, zum Beispiel:

- CXSC: U. Kulisch u.a: C-XSC, Springer-Verlag, 1993;
  http://www.math.uni-wuppertal.de:80/org/WRST/xsc/cxsc.html
- NEWMAT09: http://physuna.phy.uc.edu/~carey/newmatdocs/index.html
- TNT: http://math.nist.gov/tnt/

2.9 Zeiger

- `FKT` und `double(*_)(double)`
- `T x[]` und `const T*`
- `T*`

Zeiger (im Sinne von Adressen):

a) `typedef double(*FKT)(double)`
Alle Funktionsnamen `fkt` von Funktionen der Signatur

```
double fkt(double);
```

haben den Typ `FKT` (Zeiger auf Funktionen von `double` nach `double`; `double(*_)(double)`

b) Feldnamen `x` des Typs `T x[n];` haben den Typ

```
const T*
```

(Zeiger auf das Startelement des Feldes)

c) Variablen `x_adr` des Typs

```
T* x_adr;
```

können „Adressen“ bzw. Zeiger auf (statisch oder dynamisch erzeugte) Objekte des Typs `T` enthalten:

```
T x;
x = y;
x_adr = &x;
```

oder

```
x_adr = new T(y);
```

d) `this` – **ein Zeiger:** `*this` steht innerhalb einer Klassenmethode für das Klassenexemplar selbst, auf das diese Methode gerade angewandt wird. `this` ist also ein Zeiger. Bemerkung: Für `(*this).xxx` schreibt man kürzer: `this->xxx`.

2.10 Structures

Elemente aus $T_1 \times T_2 \times T_3$ können dargestellt werden durch:

```
class T13 {
   public:
        T1 t1;
        T2 t2;
        T3 t3;
}

T13 x;
x.t1 = ... ;

// Komponenten sind dann statt mittels numerischer Indices nur durch
// Attributsnamen zugaenglich !
```

Eine Klasse, deren Default-Sektion `public` ist, wird auch Struktur genannt:

```
struct T13 {
        T1 t1;
        T2 t2;
        T3 t3;
}

...
```

2.11 Unions

Elemente der disjunkten Vereinigung $T_1 \dot\cup T_2 \dot\cup \ldots$ können als `union` modelliert werden:

```
union Ergebnis {
  double x;
  complexd z;
};

Ergebnis erg;
switch (erg_typ) {
 case REAL:
    cout << erg.x << endl;
    break;
 case COMPLEX:
    cout << erg.z << endl;
}
```

Im Speicher stehen `x` oder `z` an derselben Stelle! Warnung: Man muß wissen, was in `erg` abgespeichert ist, um die richtige semantische Interpretation sicherzustellen.

Vorzeichenbit, Exponent und Mantissenbits einer `float`:

```
#include <iostream>

using namespace std;

int main(){

  union floatBit{
        float d;
        struct{
           bool sign:1;
           long exp:8;
           long mant:23;
        } bits;
  };

  floatBit x;
  cout << hex;

  x.d = 4.1234;
  cout << "sign= " << x.bits.sign << endl;
  cout << "exp = " << x.bits.exp << endl;
  cout << "mant= " << x.bits.mant << endl;
  cout << endl;

  x.d = -4.1234;
  cout << "sign= " << x.bits.sign << endl;
  cout << "exp = " << x.bits.exp << endl;
  cout << "mant= " << x.bits.mant << endl;
  cout << endl;

  x.d = 0.0;
  cout << "sign= " << x.bits.sign << endl;
  cout << "exp = " << x.bits.exp << endl;
  cout << "mant= " << x.bits.mant << endl;
  cout << endl;

  x.d = 14.0 / x.d;
  cout << "sign= " << x.bits.sign << endl;
  cout << "exp = " << x.bits.exp << endl;
  cout << "mant= " << x.bits.mant << endl;
  cout << endl;
}
```

2.12 Dynamische Container-Typen

2.12.1 Einfach gelinkte Liste (mit Zeigern)

Eine beliebig lange, beliebig verkürz- und verlängerbare Ansammlung von linear angeordneten Elementen (gleichen Typs) wird **Liste** genannt und kann wie folgt implementiert werden:

```
////////////////////////////
// Datei:  linkedList.cc
// Version: 1.0
// Autor:  Hans-Juergen Buhl
// Datum: 17.09.1998
////////////////////////////

#include <iostream>
#include <string>
#include <cassert>
#include <cstdlib>

using namespace std;

template<class T>
class linkedList{

  struct Knoten{
    T Daten;
    Knoten *next;
    Knoten(const T& daten, Knoten *p):
      Daten(daten), next(p) {};
  } *Anfang;

  void clear_all(Knoten* abStelle){
    if (abStelle != 0) {
      clear_all(abStelle->next);
      delete abStelle;
    };
  };

public:

  linkedList<T>() : Anfang(0) {};

  ~linkedList<T>() { clear_all(Anfang); };

  void push_front(const T& daten){
    Knoten* temp = new Knoten(daten, Anfang);
```

```
    // assert(temp != 0);
    Anfang = temp;
  };

  void pop_front() {
    if (Anfang != 0) {
      Knoten* temp(Anfang);
      Anfang = Anfang->next;
      delete temp;
    };

  }

  class iterator;
  friend class iterator;

  class iterator {
    Knoten* aktuell;
  public:
    iterator(Knoten* init = 0) : aktuell(init) {};
    T& operator*() { return aktuell->Daten; };
    const T& operator*() const {return aktuell->Daten; };

    iterator& operator++(){        // praefix
      if (aktuell != 0) aktuell = aktuell->next;
      return *this;
    };

    iterator operator++(int){      // postfix
      iterator temp = *this;
      ++*this;
      return temp;
    };

    bool operator!=(const iterator& x) const {
      return (aktuell != x.aktuell);
    };
  };

  iterator begin() const { return iterator(Anfang); };
  iterator end() const  { return iterator(); };
};

int main()
{
  {
    linkedList<string> stringList;
    stringList.push_front("Erster eingefuegter Listenknoten");
    stringList.push_front("Zweiter eingefuegter Listenknoten");
```

```
    stringList.pop_front();
    stringList.push_front("Dritter eingefuegter Listenknoten");
    // ...
    stringList.push_front("ENDE!");

    linkedList<string>::iterator pos(stringList.begin());
    for (; pos != stringList.end(); pos++)
      cout << *pos << endl;

  }

  return 0;
}
```

In der Veranstaltung **Algorithmen und Datenstrukturen** werden wir die Alternativen verschiedener Implementierungen solcher Listen und vieler anderer Container-Typen kennenlernen.

2.12.2 Datentyp `list` aus der `STL`

Die „wichtigsten“ Container-Typen stehen in C++ in der `STL` in einer Template-Version schon implementiert zur Verfügung:

```
////////////////////////////
// Datei:  STLlist.cc
// Version: 1.0
// Autor:  Hans-Juergen Buhl
// Datum: 17.09.1998
////////////////////////////

#include <iostream>
#include <string>
#include <list>

using namespace std;

int main()
{
  {
    list<string> stringList;
    stringList.push_front("Erster eingefuegter Listenknoten");
    stringList.push_front("Zweiter eingefuegter Listenknoten");
    // ...
    stringList.push_front("ENDE!");

    list<string>::iterator pos(stringList.begin());
    for(; pos != stringList.end(); pos++)
      cout << *pos << endl;
  };
  {
    typedef list<double> rliste;
    rliste r1;
    r1.push_front(1.1);
    r1.push_front(1.2);
    r1.push_front(1.3);
    r1.push_front(1.4);
    r1.push_front(1.5);
    rliste::iterator pos(r1.begin());
    for(; pos != r1.end(); pos++)
      cout << *pos << endl;
  };

  return 0;
}
```

A) **Containertypen**

`vector`	wahlfreier Zugriff
`list`	wenn häufig in Mitte eingefügt/gelöscht wird
`dequeue`	wenn häufig an beiden Enden eingefügt/gelöscht wird

B) Klassische abstrakte Datentypen

`stack`	Kellerspeicher, Einfügen/Löschen/Zugriff am Ende
`queue`	Warteschlange, Einfügen am Ende, Löschen/Zugriff am Anfang
`priority_queue`	Prioritäts-Warteschlange
`set`	schnelle Elementsuche, wenn Daten und Schlüssel identisch sind
`multiset`	schnelle Elementsuche, wenn Daten und Schlüssel identisch sind
`map`	schnelle Elementsuche, wenn Daten und Schlüssel nicht identisch sind
`multimap`	schnelle Elementsuche, wenn Daten und Schlüssel nicht identisch sind

Bemerkung:
`multiset`'s und `multimap`'s können gleichartige Elemente mehrfach enthalten.

2.13 Parameterübergabe an `main()`

```
int main(int argc, char* argv[])
{
 ...
 for (int i=0; i<argc; i++)
   cout << argv[i];
 ..
}
```

Übersetze das Programm mittels

```
make xxx <cr>
```

Dann gib ein :

```
xxx Hallo du drinnen ! <cr>
```

Was ist das Ergebnis?

Index