Informatik I

Wintersemester 2003/2004

http://graphics.ethz.ch/37-847

Dr. Ronald Peikert
Institut für wissenschaftliches Rechnen
ETH - Zentrum, IFW C27.2
8092 Zürich

peikert@inf.ethz.ch
01 63 25569

Ziele der Vorlesung

Einführung ins *Programmieren*, orientiert an den Bedürfnissen in Mathematik und Naturwissenschaften

- Wichtigste Elemente der Sprache C++
- Praktische Programm-Beispiele
- Einbindung von Software-Bibliotheken

Algorithmen und *Datenstrukturen*

- Implementation in C++
- Korrektheit
- Analyse von Rechenzeit und Speicherbedarf

Warum C++?

Entwicklung der Programmiersprachen:

- Assembler – Programmierung auf der Stufe einzelner Maschineninstruktionen, symbolische Namen.
- Fortran('57), Cobol('59), Basic('64) – Frühe Programmiersprachen, Anweisungen statt Instruktionen.
- Algol ('60), Pascal('71), C ('72) – Strukturierte Programmierung, Sicherheit, Effizienz.
- Modula-2('78), Ada('79), C++('86), Java ('95) – Modulare resp. objektorienterte Programmierung, Module, Schnittstellen, Wiederverwendbarkeit, Eignung für grosse Programmsysteme.

- Lisp('58), Prolog ('71) – Spezielle Sprachen, Funktionale Programmierung, Logik-Programmierung.

C++ und Java sind recht ähnliche Sprachen. Java ist unbestritten die modernere und auch "schönere" Sprache. Es gibt aber mehrere Gründe, die für C++ als Basis dieser Vorlesung sprechen:

- *Verbreitung* und *Beständigkeit:* Die heutigen Betriebssysteme (Windows, Unix, Linux, MacOS, etc.) wurden fast ausschliesslich in C oder C++ entwickelt.
- *Kompatibilität* mit vielen Software-Bibliotheken für Numerik, Computer-Graphik, etc.
- *Effizienz,* speziell beim wissenschaftlichen Rechnen ein wichtiges Kriterium.

Die Betriebssystem-Umgebung

Betriebssysteme sind nicht Gegenstand dieser Vorlesung. C++ selber ist unabhängig von der Wahl des Betriebssystems. Mit den Programm-Bibliotheken und dem Compiler kommt aber eine gewisse Systemabhängigkeit ins Spiel.

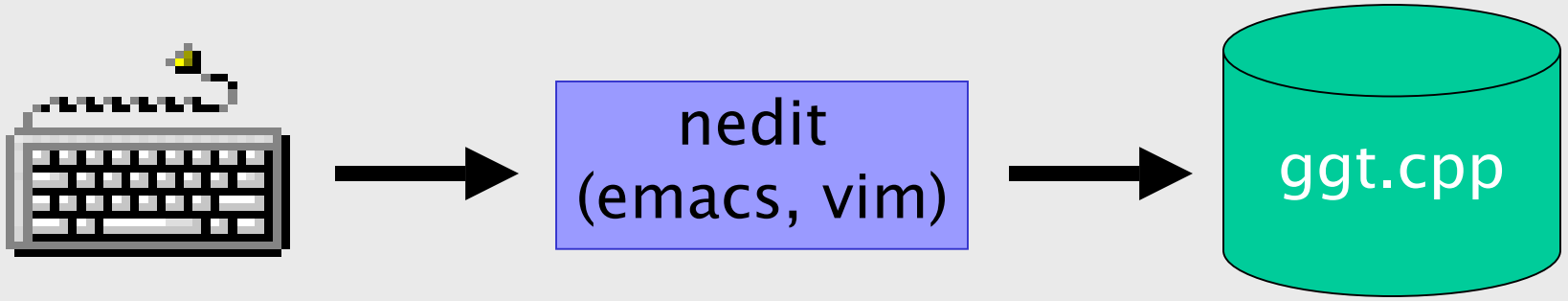
In Vorlesung und Übungen wird deshalb folgende Umgebung zugrundegelegt:

- Eine Unix-Shell als *command line interface* zum Betriebssystem.
- Der Compiler g++ vom GNU Project.

Diese "schlanke" Umgebung erlaubt, auf allen Unix-Systemen sowie den PC-Betriebssystemen Linux und Windows auf recht einheitliche Weise zu arbeiten.

Programmierung und Programme

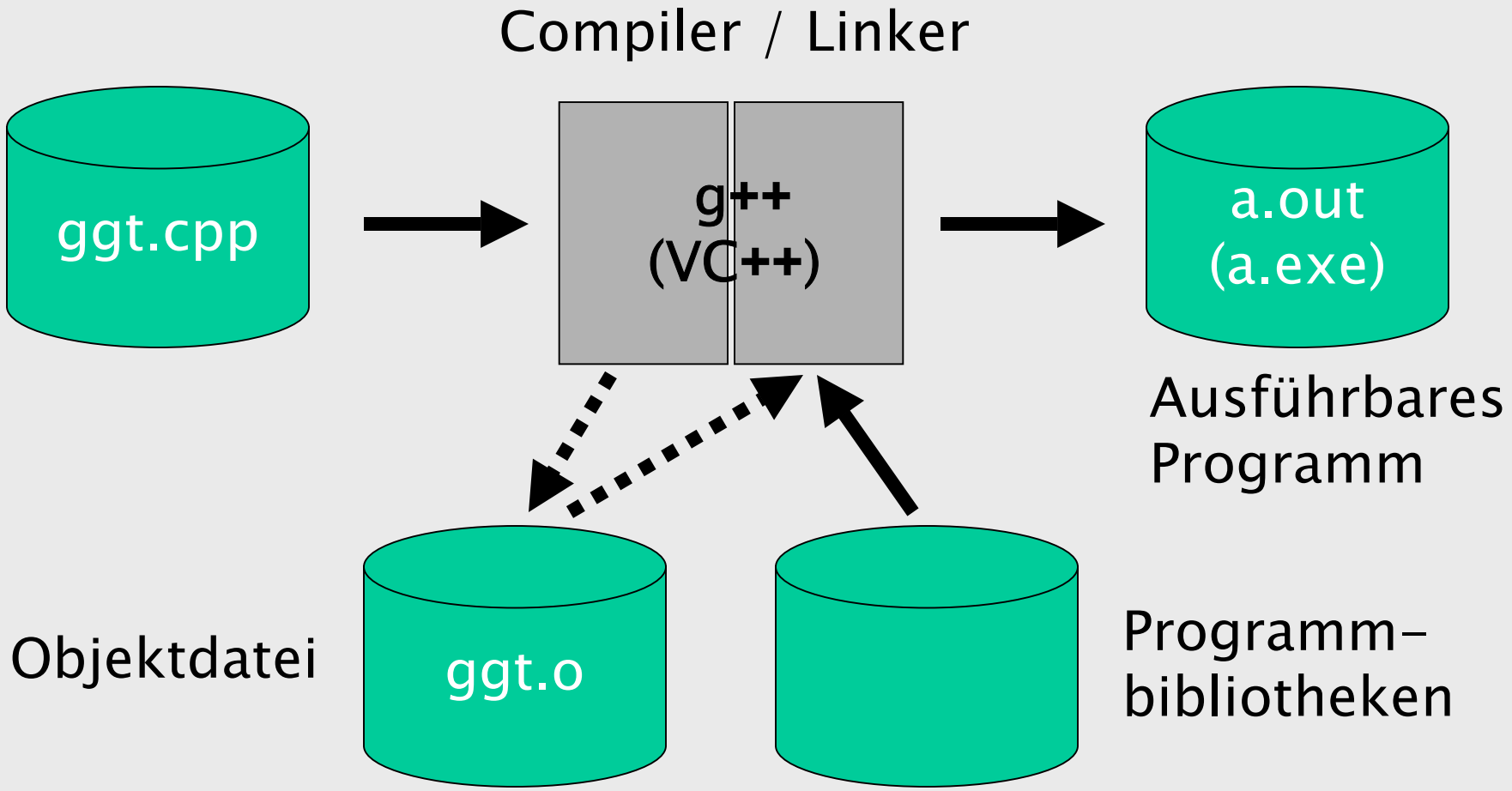
Schritt I – Erstellen eines Quellprogrammes



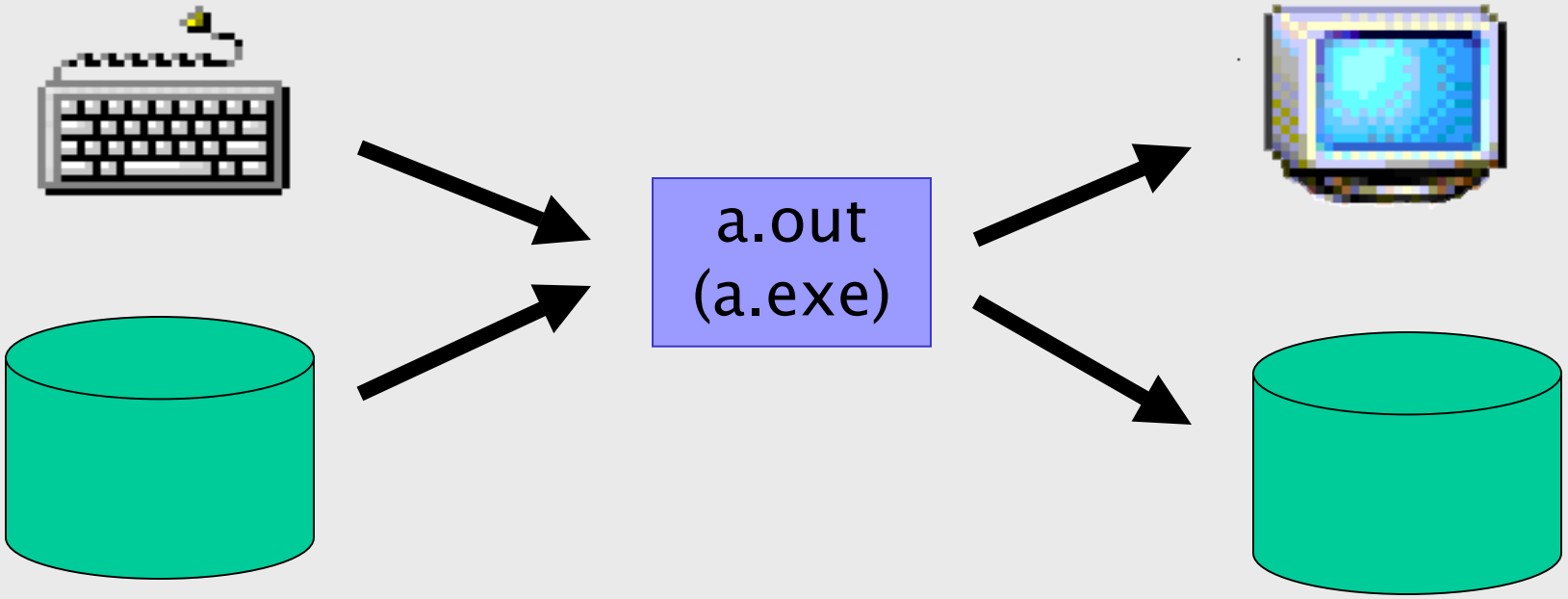
Texteditor

Quellprogramm
in der Sprache
C++

Schritt II – Übersetzen in Maschinensprache



Schritt III – Ausführung



Eingabe von Tastatur und/oder Datei

Ausgabe auf Bildschirm und/oder Datei

Erste C++ Programme

Das einfachste gültige C++ Programm lautet:

```
int main(){}
```

Es ist das “leere Programm”. Vom Compiler wird es akzeptiert und auch übersetzt.

Bei sinnvolleren Programmen befindet sich zwischen den Klammern `{}` eine Folge von *Deklarationen* und *Anweisungen*.

Eine Deklaration ist zum Beispiel

```
float x, y, summe;
```

Und eine Anweisung ist zum Beispiel

```
summe = x + y;
```

Das Programm

```
int main()
{
  float x, quadrat;
  x = 1.5;
  quadrat = x * x;
}
```

ist ebenfalls syntaktisch korrekt.

Fehlte die Deklaration `float x, quadrat;` so wäre dies ein *Syntax-Fehler*. Der Compiler würde diesen melden und das Programm nicht akzeptieren.

Fehlte die Zuweisung `x = 1.5;` so wäre dies ein *semantischer Fehler*. Manche Compiler geben eine *Warnung* aus.

Alle Programme brauchen *Dateneingabe* und *-ausgabe*. Eine Form davon ist Texteingabe und -ausgabe via Tastatur und Bildschirm.

Die Sprache C++ selber bietet keine E/A-Funktionen an, dazu müssen *Programmbibliotheken* verwendet werden. Funktionen für Text-E/A befinden sich in der *Standardbibliothek*.

Die Standardbibliothek wird automatisch dazugeladen. Die Text-E/A Funktionen erfordern aber Deklarationen, die in der Datei “iostream” stehen (in einem Verzeichnis wie /usr/include/g++, je nach System).

Statt die Datei selber ins Programm einzufügen genügt die Zeile

```
#include <iostream>
```

an der betreffenden Stelle.

Damit können wir nun ein Programm mit Ein- und Ausgabe schreiben:

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

int main()
{
      int x, y, summe;
      cout << " x = ";  cin >> x;
      cout << " y = ";  cin >> y;
      summe = x + y;
      cout << " Die Summe ist ";
      cout << summe;  cout << endl;
}
```

summe.cpp

Die ersten zwei Zeilen sind nötig, weil wir E/A-Funktionen aus der Standardbibliothek brauchen wollen.

Nach `main()` folgt syntaktisch gesehen ein sog. *Block*. Blöcke treffen wir später auch an anderen Stellen an. Sie bestehen aus einem Klammernpaar `{}` und dazwischen einer Folge von Deklarationen und Anweisungen.

Sowohl Deklarationen wie auch Anweisungen enden mit einem Semikolon `;`

Im Beispiel sind es eine Deklaration und acht Anweisungen.

Das Beispiel verwendet aus der Standardbibliothek:

- den Eingabestrom **`cin`** und den Ausgabestrom **`cout`**
- die Eingabe- und Ausgabeoperatoren **`>>`** und **`<<`**
- das Zeilenende **`endl`**

Der Operator **`cout`** passt sich dem Operanden-Typ an:

- Zeichenketten *(strings)* werden unverändert ausgegeben.
- Variablen für (ganze oder Gleitkomma-) Zahlen werden ausgewertet und im Dezimalsystem ausgegeben.
- etc.

Nicht jedes Programm braucht Text-E/A. Stattdessen kann z.B. grafische Ein-/Ausgabe verwendet werden.

zeichne.cpp

Die Zuweisung

Die einfachste Form der Anweisung ist die Zuweisung *(assignment)*. Die Zuweisung hat die Syntax

`Variable = Ausdruck ;`

d.h. sie besteht aus dem Zuweisungsoperator **`=`** mit zwei Operanden (links und rechts davon) und dem Semikolon **`;`** mit dem jede Anweisung endet.

Die beiden Operanden müssen eine Variable resp. ein Ausdruck sein, was beides noch erklärt wird.

Die Schreibweise mit den Einrahmungen ist "metasprachlich". Gemeint ist, dass man je eine gültige Variable und einen gültigen Ausdruck einsetzt.

Ein Beispiel einer Zuweisung ist

```
a = a + 1;
```

Die Zuweisung ist keine mathematische Gleichung sondern eine Operation.

Es wird zuerst der Ausdruck ausgewertet und der erhaltene Wert dann der Variablen zugewiesen.

In anderen Sprachen wie Pascal oder Maple wird der Zuweisungsoperator nicht `=` sondern `:=` geschrieben.

Variablen

Eine Variable beschreibt eine Grösse die während der Ausführung des Programmes ihren Wert ändern kann.

Die Variable wird im Computer durch eine Zelle im Arbeitsspeicher (*memory*, RAM) realisiert. Glücklicherweise erspart uns C++ aber, von „Speicherzelle Nr. 7654321“ etc. zu reden. Stattdessen werden symbolische Namen, sog. Bezeichner *(identifiers)* verwendet. Bezeichner werden später auch für weitere Dinge wie Funktionen gebraucht.

Ein gültiger Bezeichner ist eine Folge von Buchstaben, Ziffern oder dem Unterstrich `_` deren erstes Zeichen ein Buchstabe oder der Unterstrich ist.

Beispiele: **`kaese_menge, kaeseMenge, __1a__`**

Bei der Wahl eines Bezeichners muss darauf geachtet werden, dass er nicht mit einem der 63 reservierten Schlüsselwörter *(keywords)* zusammenfällt.

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

Jede Speicherzelle von z.B. 32 Bits kann unterschiedliche Daten speichern wie:

- Ganzzahl von -2^{31} bis $2^{31}-1$
- Ganzzahl von 0 bis $2^{32}-1$
- Gleitkomma-Zahl
- 4 Zeichen (Buchstaben, Ziffern, Spezialzeichen) nach einer Codierungstabelle wie ASCII oder EBCDIC
- etc.

Die Speicherzelle selber enthält keine Angabe über den *Datentyp*. Damit deren Inhalt trotzdem korrekt interpretiert werden kann, muss der Variable ein Datentyp fest zugeordnet werden, was durch die Deklaration geschieht.

Einfache Datentypen

Die *einfachen Datentypen* dienen zur Beschreibung von

- ganzen Zahlen mit oder ohne Vorzeichen
- reellen Zahlen mit oder ohne Vorzeichen
- Buchstaben, Ziffern und anderen Textsymbolen
- Wahrheitswerten
- Aufzählungen

Auf den einfachen bauen die *abgeleiteten Datentypen* wie Feld, Verbund, Zeiger und Klasse auf, die wir später kennenlernen werden.

Ganzzahlen

short, **int** und **long** sind die Datentypen der ganzen Zahlen *(integers).*

Auf vielen Rechnern ist **short** 16, **int** 32 und **long** 32 oder 64 Bit lang.

Es gilt: **int** ist mindestens so lang wie **short** und höchstens so lang wie **long**.

Der Sinn der drei Datentypen ist die Optimierung des Speicherplatzes (**short**), der Rechenzeit (**int**), resp. des Wertebereiches (**long**).

Es gibt dazu auch die vorzeichenlosen Varianten **unsigned short**, **unsigned long** und **unsigned int** . Diese haben einen Wertebereich von 0 bis 2^N-1 (bei N Bit langen Zahlen).

Für vorzeichenbehaftete Zahlen (**`short`**, **`long`** und **`int`**) gibt es zwei Codierungen, das *Einer-* und das *Zweierkomplement:*

- Das erste Bit von links gibt das *Vorzeichen* an, 0 bedeutet positiv, 1 bedeutet negativ.
- Den *Betrag* einer negativen Zahl erhält man durch Wechseln jedes Bits. Im Falle des Zweierkomplements wird zum Ergebnis noch 1 addiert.

Es wird praktisch nur das Zweierkomplement verwendet. Seine Vorteile sind:

- Die Null hat nicht zwei verschiedene Darstellungen.
- Das Vorzeichen muss nicht speziell behandelt werden, man kann modulo 2^N rechnen!

Beispiel: Addition (–1) + 1 auf einem 8-Bit Rechner im Zweierkomplement.

	Zahl	Codierung
Erster Operand	–1	**11111111**
Zweiter Operand	1	**00000001**
Summe		**00000000**
Übertrag (ignoriert)		**1**
Ergebnis	0	**00000000**

Der Wertebereich beim Zweierkomplement ist **10...0** bis **01...1**, also -2^{N-1} bis $2^{N-1}-1$.

Die *Grenzen* der Wertebereiche werden manchmal im Programm gebraucht, z.B. zur Vermeidung von Fehlern durch Überlauf. Dann ist es besser statt der Zahlen die *symbolischen Konstanten*

```
SHRT_MIN,  SHRT_MAX,  USHRT_MAX,
INT_MIN,   INT_MAX,   UINT_MAX,
LONG_MIN,  LONG_MAX,  ULONG_MAX
```

zu verwenden. Man macht sie verfügbar durch

```
#include <climits>
```

Symbolische Konstanten machen Programme

- lesbarer, weil sie die Bedeutung einer Zahl angeben,
- universeller, weil man ihre Definition bei Bedarf ändern kann.

Gleitkommazahlen

`float`, **`double`** und **`long double`** sind die Datentypen der Gleitkomma-Zahlen *(floating point numbers)*.

Im Dezimalsystem bedeutet Gleitkommazahl:
Vorzeichen mal Mantisse mal Zehnerpotenz.

Die Mantisse hat genau eine Stelle vor dem Komma.
Diese ist ungleich Null, ausser für Null selbst.

Beispiel: $0.00000000000000000000000000000911$

- in Gleitkomma-Darstellung: $9.11 \cdot 10^{-31}$
- in C++ Schreibweise:

```
float electronMass;
electronMass = 9.11e-31;
```

Im Binärsystem bedeutet Gleitkommazahl entsprechend: Vorzeichen mal Mantisse mal Zweierpotenz.

Die Mantisse hat wiederum genau eine Stelle vor dem Komma. Diese Vorkommastelle wird aber bei der Speicherung weggespart, weil sie ja (ausser bei der Zahl Null) immer eine Eins ist.

Heutige Rechner arbeiten nach dem IEEE-Standard:

Anzahl Bits	`float`	`double`
Vorzeichen	1	1
Exponent	8	11
Mantisse	23	52

Der 128 Bit Typ `long double` ist nicht standardisiert.

Der Exponent ist auf spezielle Weise codiert *(biased)* so dass er nie lauter 0-Bits und nie lauter 1-Bits enthält. Beim Datentyp **`float`** sind z.B. Exponenten von -126 bis +127 möglich.

Die Zahl Null wird durch lauter 0-Bits dargestellt.

Nicht jedes Bitmuster entspricht einer gültigen Gleitkommazahl. Ungültige Zahlen werden als "NaN" *(not a number)* ausgegeben.

Die Datentypen **`float`** und **`double`** enthalten auch noch die Unendlich-Werte **`-inf`** und **`+inf`** (zwei Bitmuster mit lauter 1-Bits im Exponenten).

Zeichen

`char` ist der Datentyp der Textzeichen *(characters)* wie Buchstaben, Ziffern, Interpunktionszeichen, Leerzeichen, etc.

Die Zeichen werden nach einer Tabelle wie ASCII *(American Standard Code for Information Interchange)* oder EBCDIC in eine 8 Bit lange Zahl codiert.

`char` ist deshalb ein weiterer Ganzzahl-Datentyp. Es existieren sogar die Varianten **`unsigned char`** (0 bis 255) und **`signed char`** (-128 bis 127), die man verwendet, wenn wirklich Zahlen und nicht Zeichen gemeint sind.

ASCII Zeichensatz (erste 128 von 256 Zeichen):

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0_	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	NL	VT	NP	CR	SO	SI
1_	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2_	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3_	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4_	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5_	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6_	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7_	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Nur die erste Hälfte der ASCII-Tabelle ist standardisiert (7-Bit-Zeichensatz). Die zweite Hälfte enthält unter anderem Buchstaben mit Umlauten und Akzenten.

Umlaute und Akzente sind in Bezeichnern nicht erlaubt, wohl aber bei **`char`**-Konstanten sowie innerhalb von Strings (Beispiele folgen).

Nebenbei: Für andere Alphabete bietet C++ einen Typ **`wchar_t`** *(wide character type)* . Der verwendete Code, „Unicode UTF-16“ hat 16 Bits und bis jetzt wurden darin rund 38000 Zeichen definiert. Die **`iostream`**-Operationen wie **`cout <<`** etc. funktionieren aber (noch?) nicht mit **`wchar_t`**.

Einige Beispiele:

```
char c;
c = 'a';
c = 97;       // gleichwertig in ASCII
c = 0x61;     // dito (0x heisst hexadezimal)
c = 'ä';      // erlaubt, aber nicht Standard
cout << ”Käse”;   // dito
c = 'i' + 1; // wie c = 'j', nicht für EBCDIC
c = 9;        // Tabulator in ASCII
c = '\t';     // Tabulator (besser!)
c = 0;        // wie c = '\0', Nullzeichen
c = '0';      // Ziffer 0, wie c = 48 in ASCII
```

Einige wichtige Spezialzeichen mit Ersatzdarstellung *(escape sequence)* in C++:

Beschreibung	Zeichen	ASCII	escape seq.
Zeilenende	NL	**0x0a**	**\n**
Wagenrücklauf	CR	**0x0d**	**\r**
Horizontaler Tabulator	HT	**0x09**	**\t**
Vertikaler Tabulator	VT	**0x0b**	**\v**
Alarm	BEL	**0x07**	**\a**
Backspace	BS	**0x08**	**\b**
Seitenvorschub	NP	**0x0c**	**\f**
Nullzeichen	NUL	**0x00**	**\0**
einfache Anführungszeichen	'	**0x27**	**\'**
doppelte Anführungszeichen	"	**0x22**	**\"**
Backslash	\	**0x5c**	****
Fragezeichen	?	**0x3f**	**\?**

Wahrheitswerte

`bool` ist der Datentyp der Wahrheitswerte (*Boolean,* benannt nach George Boole, 1815–1864).
Er besteht aus den zwei Werten **`false`** und **`true`**.

AND	F	T
F	F	F
T	F	T

OR	F	T
F	F	T
T	T	T

NOT	
F	T
T	F

Typkonversionen

Alle bisher behandelten Datentypen (und einige weitere, aber nicht alle) können gegenseitig zugewiesen werden. Dabei wird automatisch konvertiert:

```
double d;  d = 'a';
int pi;  pi = 3.14159;
unsigned short s;  s = -1;
```

Einige Compiler warnen in Fällen wo Werte durch die Konversion verändert werden. Um Warnungen zu vermeiden, kann man auch explizit konvertieren *(cast)* :

```
int pi;  pi = (int)3.14159;
unsigned short s; s = (unsigned short)-1;
```

Beim Zuweisen eines Wahrheitswertes an eine Zahl gilt die Konversion:

$$\texttt{false} \rightarrow 0$$
$$\texttt{true} \rightarrow 1$$

Beispiel: `int i; i = true;` weist die Zahl 1 zu.

Bei Zuweisung in die andere Richtung gilt:

$$0 \rightarrow \texttt{false}$$
$$\neq 0 \rightarrow \texttt{true}$$

Beispiel: `bool b; b = -2;` weist den Wert `true` zu.

Nicht nur bei der Zuweisung, auch bei Operationen werden Datentypen konvertiert *(propagation)*.
Für *arithmetische* Operationen gibt es 10 (!) Regeln, die man in etwa so zusammenfassen kann:

- Wenn keiner der Operanden einem der unten-stehenden Datentypen angehört, werden beide Operanden in **`int`** konvertiert.

- Andernfalls werden beide Operanden in den Typ konvertiert, der in der Sequenz

  `unsigned int` → **`long`** → **`unsigned long`** → **`float`** → **`double`** → **`long double`**

  später folgt.

Aufzähltypen

Mittels **enum** können Datentypen durch explizites Aufzählen *(enumeration)* ihrer Wertebereiche erzeugt werden. Ein Beispiel ist:

```
enum wochentag { sonntag, montag, dienstag,
  mittwoch, donnerstag, freitag, samstag };

wochentag heute, morgen; // Deklaration
heute = dienstag;        // Zuweisung
morgen = heute + 1;      // leider verboten
```

enum Datentypen können, automatisch oder mittels *cast*, in Ganzzahltypen konvertiert werden, aber nicht umgekehrt.

Ausdrücke

Bei Zuweisungen haben wir auf der rechten Seite bis jetzt angetroffen:

- Konstanten wie **`1.23e-45`**, **`0xffff`**, **`'#'`**,
- Variablen wie **`i`** oder **`a1`**
- arithmetische Ausdrücke wie **`a+1`** oder **`x*x`**

Alles was auf der rechten Seite einer Zuweisung stehen darf, wird in C++ als Ausdruck *(expression)* bezeichnet.

Ein Ausdruck ist entweder

- eine Konstante
- eine Variable
- aufgebaut aus einem Operator und Operanden, oder
- eine Folge **(`Ausdruck`)**.

Punkt 4 bedeutet einen Ausdruck zwischen runden Klammern **( )**. Wir haben es also mit einer *rekursiven* Definition zu tun.

Die Operatoren von C++ sind in der folgenden Tabelle vollständig aufgeführt.

Die Operanden sind selber wiederum Ausdrücke. (Hier ist nochmals eine Rekursion).

Operatoren und Operanden

Operatoren der Prioritätsstufen 1 bis 13:

Priorität	Assoz.	Operator	Operanden	Bedeutung
13	L	* / %	2	multiplikative Operatoren
12	L	+ -	2	additive Operatoren
11	L	>> <<	2	Shift-Operatoren
10	L	< <= > >=	2	relationale Operatoren
9	L	== !=	2	Gleichheitsoperatoren
8	L	&	2	bitweises UND
7	L	|	2	bitweises ODER
6	L	^	2	bitweises Exklusiv-ODER
5	L	&&	2	logisches UND
4	L	||	2	logisches ODER
3	L	?:	3	Konditional-Operator
2	R	= *= /= += -= >>= <<= &= ^= |=	2	Zuweisungs-Operatoren
1	L	,	2	Kommaoperator

Operatoren der Prioritätsstufen 14 bis 17:

Priorität	Assoz.	Operator	Operanden	Bedeutung
17	R	`::`	1	Globaler Geltungsbereich
	L	`::`	2	Klassen-Geltungsbereich
16	L	`->  .`	2	Elementauswahl
	L	`[]`	2	Indexoperator
	L	`()`	2	Funktionsaufruf
	L	`sizeof`	1	Grösse
	R	`dynamic_cast const_cast reinterpret_cast static_cast typeid`	2	Typkonversion
15	R	`++  --`	1	Inkrement, Dekrement
	R	`~`	1	bitweise Negation
	R	`!`	1	logische Negation
	R	`+  -`	1	Vorzeichen
	R	`*  &`	1	Inhalts-, Adressoperator
	R	`()`	2	Typkonversion (cast)
	R	`new  delete`	1	dynamische Speicherverwaltung
14	L	`->*  .*`	2	Dereferenzierung

Anzahl und Anordnung der Operanden müssen der vom jeweiligen Operator verlangten Syntax entsprechen.

Die 1-stelligen Operatoren verlangen Präfix-Notation.

Eine Ausnahme bilden die Operatoren `++` und `--` welche sowohl als Präfix- und Postfixoperatoren existieren.

Beispiele mit Ausdrücken `i++` und `++i`:

```
j = i++;  // gleichwertig mit j=i; i=i+1;
j = ++i;  // gleichwertig mit i=i+1; j=i;
i++;   // oder ++i; gleichwertig mit i=i+1;
```

Die meisten 2-stelligen Operatoren verlangen Infix-Notation.

Ausnahmen bilden beispielsweise:

- der Index-Operator (um in C++ Ausdrücke wie a_i zu bilden), z.B. `a[i]`
- der Funktionsaufruf, z.B. `f()`oder `f(j)` oder (kombiniert mit dem Komma-Operator) `f(x,0,i+1)`
- die Typkonversion, z.B. `(float)i`

Priorität

Jedem Operator ist eine Priorität *(precedence)* zugeordnet, die dafür sorgt dass beispielsweise `a + b * c` als `a + (b * c)` interpretiert wird.

Diese "Bindungsstärke" nimmt (wie in der Mathematik üblich) ab in der Reihenfolge:

- Multiplikative Operatoren `*`, `/` und `%` (Divisionsrest)
- Additive Operatoren `+` und `-`
- Vergleichsoperatoren `<`, `>`, `<=`, `>=`, `==` (Gleichheit) und `!=` (Ungleichheit)
- logisches UND `&&`
- logisches ODER `||`

Korrektes Beispiel:

```
i*i + j*j >= r2 || i==0 && j==0
```

“Beliebter” Fehler (1):

```
bool i_ist_ok;
i_ist_ok = (0 < i < 10);
```

“Beliebter” Fehler (2):

```
bool i_ist_hundert;
i_ist_hundert = (i = 100);
```

Frage: Wieso resultiert beide Male immer **true**, unabhängig von **i**?

Beispiel mit dem 3-stelligen **?:** Operator

```
cout << "100 modulo 17 = ";
int m;  cin >> m;
cout << (100%17 == m ? "richtig" : "falsch");
```

Wie später erläutert wird ist **cout** vom Typ **ostream**, einer Klasse mit **<<** als eigenem Operator. Dieser bindet stärker als **?:** was Klammern nötig macht.

Dagegen ist die Priorität der drei Operatoren **%** , **==** und **?:** absteigend (13, 9, 3) wie im Ausdruck beabsichtigt. Weitere Klammern sind also nicht nötig.

Assoziativität

Neben der Priorität besitzen die Operatoren noch eine Assoziativität. Diese regelt die Reihenfolge wenn der gleiche Operator zwei mal nacheinander auftritt wie beispielsweise in `x - y - z`. Der Minus-Operator ist linksassoziativ, deshalb wird `(x - y) - z` ausgewertet.

Dagegen ist der Zuweisungsoperator rechtsassoziativ, was beispielsweise `x = xStart = 0.5;` erlaubt.

Die Assoziativität spielt sogar bei den Operationen `+` und `*` eine Rolle, denn für die Gleitkommaarithmetik gilt (streng genommen) weder das Assoziativ- noch das Kommutativgesetz!

Beispiel (numerische Auslöschung):

assoc.cpp

```
#include <iostream>
int main()
{
  float x;  x = -10000.;
  float y;  y =  10001.;
  float z;  z =      0.0001;
  cout << (x +  y)+ z  << endl;
  cout <<  x + (y + z) << endl;
}
```