Karlsruher Institut für Technologie

Institut für Technik der Informationsverarbeitung

Institutsleitung
Prof. Dr.-Ing. K. D. Müller-Glaser
Prof. Dr.-Ing. J. Becker
Prof. Dr. rer. nat. W. Stork

Vorlesung: Informationstechnik (IT)

Prof. Dr.-Ing. K.D. Müller-Glaser

Programmiersprachen Teil1

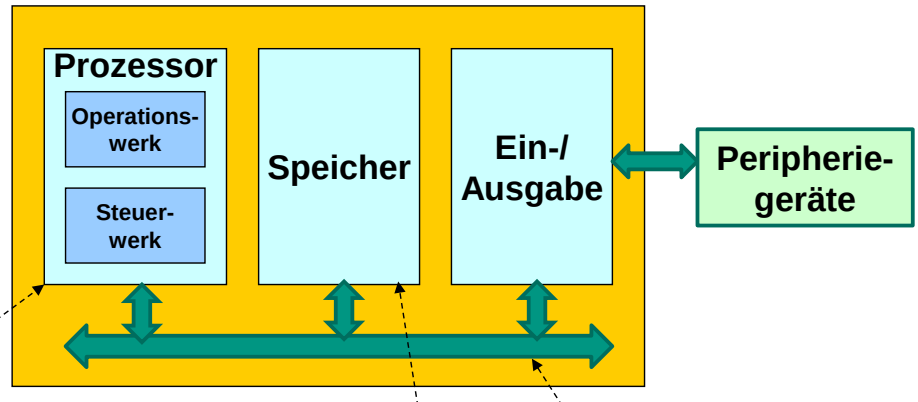
Übersicht

Karlsruher Institut für Technologie

- Kurzer Rückblick DT: Rechnerarchitektur
- Was ist eine Programmiersprache?
- Formale Sprachen
- Höhere Sprachen
- Programmierparadigmen
- Wichtige Programmiersprachen

Wiederholung aus DT: Rechnerarchitekturen

- Interne Architektur (John von Neumann Architektur 1903 – 1957)



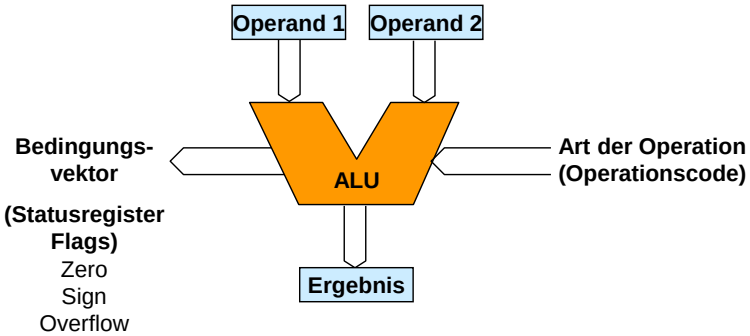
„Rechnerkern“
CPU (central processing unit)
führt Maschinenprogramm aus
Programm ist Folge von Maschinenbefehlen

Speicher
Arbeitsspeicher speichert Maschinenbefehle und zugehörige Daten

Systembus
(bidirektional)
verbindet vorhandene Komponenten,
überträgt Daten, Befehle, Steuersignale,
Busbelegung nach Vorrang der Geräte

Rechenwerk (Arithmetisch/Logische Einheit ALU)

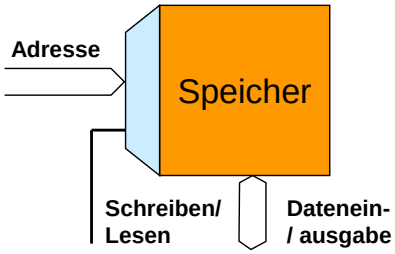
- Verarbeitung der Daten
- arithmetische und logische Basisfunktionen
- für ein einfaches Rechenwerk:
  - Arithmetische Befehle:
    Addition
    Komplementbildung
  - Logische Befehle:
    Negation
    Konjunktion
    Disjunktion
  - Liefert „Flags“ zurück an Steuerwerk



(Statusregister Flags)
Zero
Sign
Overflow

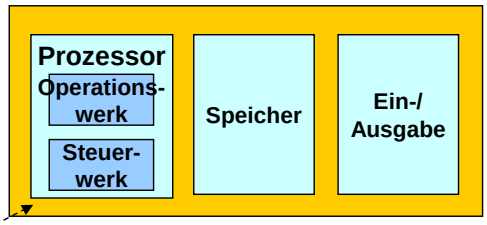
Speicherwerk

- Register im Rechnerkern zur Speicherung von Operanden, Bedingungsvektor und Ergebnis
- wahlfrei adressierbarer Schreib/Lese-Speicher (RAM, RWM), nicht für jede Operation muss Ein-/ Ausgabewerk benutzt werden
- Die Breite des Adressvektors richtet sich nach der Zahl der verfügbaren Speicherworte: $\lceil \log_2 (\text{Anzahl Speicherworte}) \rceil$



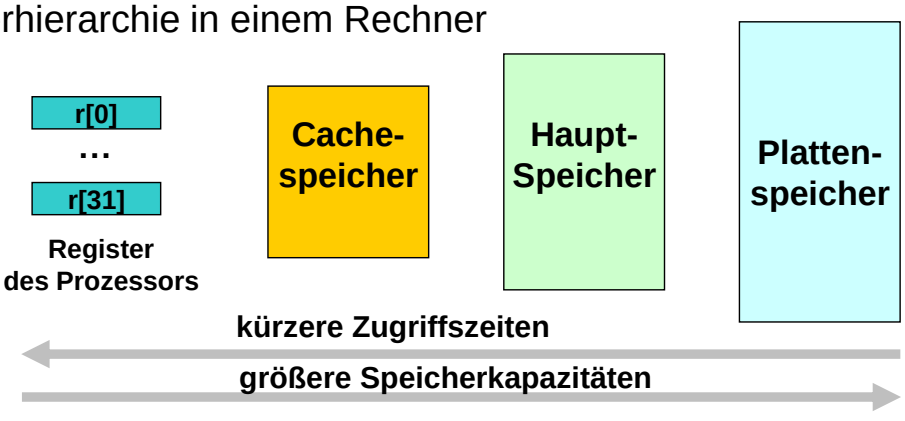
Rechnerarchitekturen

- Interne Architektur



„Rechnerkern“,
CPU (central processing unit)
Operationswerk enthält Registerspeicher

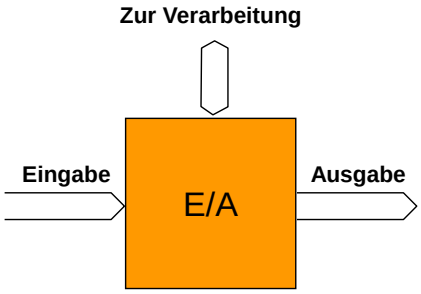
- Speicherhierarchie in einem Rechner



Ein-/Ausgabewerk

- Kommunikation mit dem Benutzer
  Ein-/Ausgabegeräte (Tastatur, Maus, Bildschirm)
  und mit den Peripheriegeräten (z.B. Drucker, Plattenspeicher)

- Eingabe:
  - Übernahme Daten von extern angeschlossenen Geräten
  - Umsetzung in eine geeignete, normierte Darstellung zur weiteren Verarbeitung
- Ausgabe:
  - Datenformatmäßige und elektrische Aufbereitung der Daten
  - Weiterleitung an externe Ausgabegeräte
  - Eventuell Auswahl des Ausgabegerätes



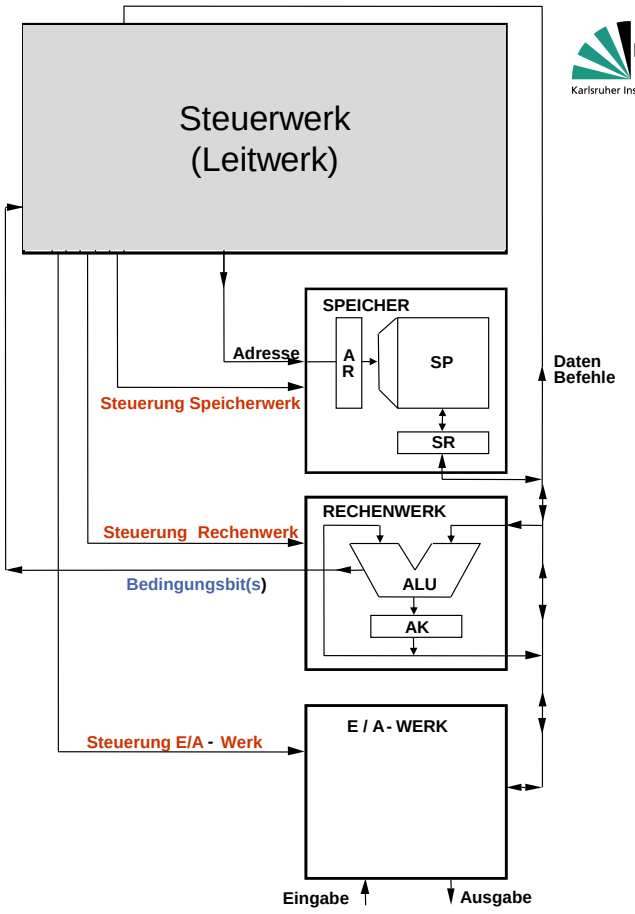
Schema eines einfachen Rechners

Befehle:

1. Transportbefehle
   - Externer Transport (Arbeitsspeicher, E/A-System)
   - Interner Transport (Register-Register)
2. Verarbeitungsbefehle
   - Arithmetische und logische Befehle
   - Schiebebefehle
   - Vergleichsbefehle
3. Sprungbefehle
4. Systembefehle (z.B. Unterbrechungsbearbeitung, Speicherverwaltung)

Abkürzungen:

SP Speicher
AR Adressregister
SR Speicherregister
ALU Arithmetik / Logik-Einheit
AK Akkumulator
E/A Ein-/Ausgabe-Werk



Von-Neumann-Architektur

Arbeitsweise:

1. Befehl aus dem Speicher in den Prozessor holen (lesender Zugriff)
2. Befehl dekodieren,
   gegebenenfalls Operanden aus dem Speicher oder der Peripherie
   in den Prozessor holen (lesender Zugriff)
3. Befehl ausführen,
   gegebenenfalls Ergebnis in den Speicher
   oder die Peripherie schreiben (schreibender Zugriff)

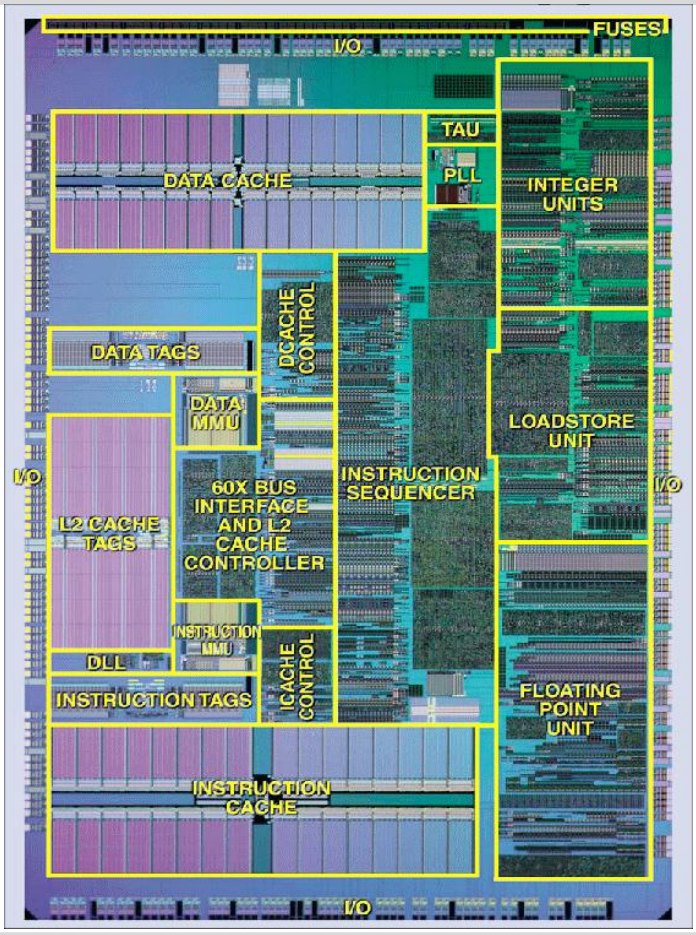
Weiter bei Schritt 1

Rechner führt im Prozessor ständig Befehle aus

Daten und Befehle sind binär codiert

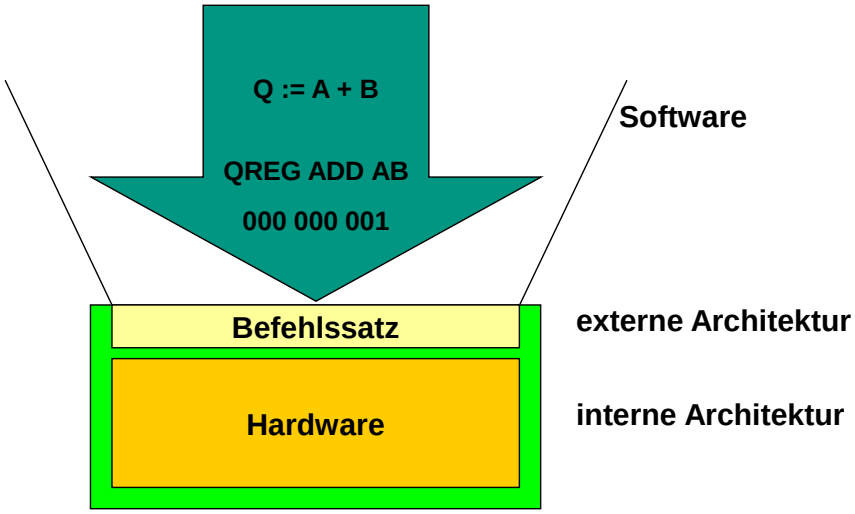
Auf Maschinenbefehlsebene nur binäre Werte 0 und 1

IBM Power PC 750 Chip-Photo



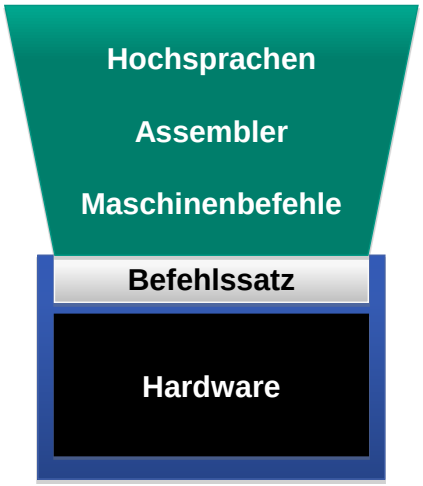
Rechnerarchitekturen

- Maschinenbefehlssatz definiert die dem Programmierer sichtbaren Befehle Instruction Set Architecture ISA (externe Architektur)
- Häufig wird ISA mit der Prozessor Architektur gleichgesetzt
- Interne Architektur (Prozessor Mikroarchitektur) definiert die interne Struktur des Prozessors
  - Verschiedene interne Architekturen können dieselbe externe Architektur aufweisen
  - Computerfamilie



Programmiersprachen

- Geschaffen um Entwicklung von Software effizienter zu machen
- Computer besitzen einen binären Maschinen-Befehlssatz
- Programmiersprachen bieten die Möglichkeit, den Computer auf höherer Ebene zu programmieren (abstrakter, verständlicher für den Menschen, weniger Fehleranfällig)



Ein Informationstechnisches System besteht aus der Einheit von Hardware und Software. Bei der Hardware handelt es sich üblicherweise um einen Prozessor mit einem speziellen Befehlssatz (bei applikationsspezifischen Standardprozessoren ASSP) oder einem standardisierten Befehlssatz (Standard Mikroprozessor wie z.B. von Intel).

Da man aber mit Computern üblicherweise nicht auf der Ebene von Nullen und Einsen kommunizieren möchte, gibt es Methoden zur Programmierung des Computers, die den Maschinenbefehlssatz durch mnemotechnische Abkürzungen (Assemblersprachen) oder durch die Hardware abstrahierende höhere Programmiersprachen für den Menschen besser handhabbar machen.

Was ist eine Programmiersprache?

- Programmiersprache: Notation für ein Computerprogramm
  - Darstellung während und nach der Entwicklung (Programmierung)
  - Übermittlung des resultierenden Programms zur Ausführung an Rechensystem
- Programmiersprache muss für maschinelle Analyse geeignet sein (zahlreiche Einschränkungen)
- Programmiersprachen basieren auf „formalen Sprachen“

Die Nutzung einer Programmiersprache ist eine formalisierte Methode, um Anweisungen an einen Computer zu übermitteln.

Programmiersprachen müssen auf einer eindeutig definierten Aussageform basieren, die es erlaubt, dem Computer in eindeutiger Weise zu sagen, was er tun soll.

Der Programmiersprache kommt dabei die Rolle einer „Lingua Franca“ zu, die sowohl der menschliche Programmierer als auch der Computer versteht.

Programmieren ist eine Tätigkeit, bei der versucht wird, durch systematischen Einsatz einer gegebenen Programmiersprache ein gestelltes Problem zu lösen.

• Ein Programm ist ein Algorithmus formuliert in einer Programmiersprache.

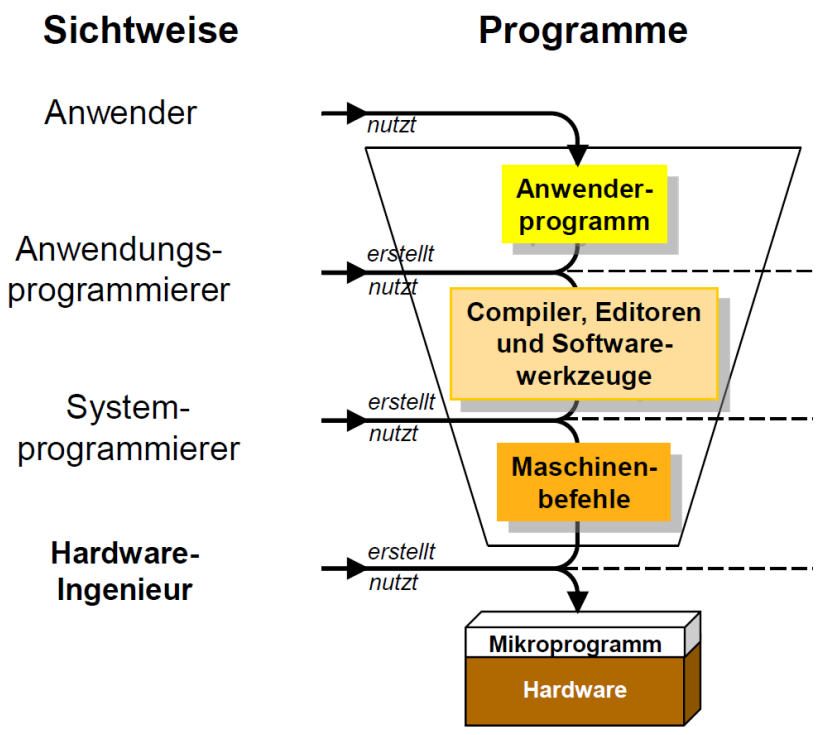
- Maschinensprachen, bzw. Assemblersprachen sind auf einen bestimmten Prozessor optimiert,

- Höhere Programmiersprachen erlauben eine problemnahe Formulierung

Programm

- Ein Programm ist eine Folge von elementaren Schritten
- Ein typischer elementarer Schritt ist die Wertzuweisung
- Komplexere Anweisungen werden aus den elementaren Anweisungen zusammengesetzt
- Die Reihenfolge der Schritte ergibt sich aus Kontrollstrukturen wie
  Konkatenation, Sequenzen (….; ….; ….;)
  Alternativen (if - then - else, switch)
  Iterationen (for, while Schleifen)
  Unterprogrammaufrufe (call / x())
- Die Speicherung der Daten erfolgt in Variablen
- Es gibt elementare Datentypen
  (Integer, Real, Boolean, Char, String, Array…),
  aus denen komplexere Datentypen zusammengesetzt werden können

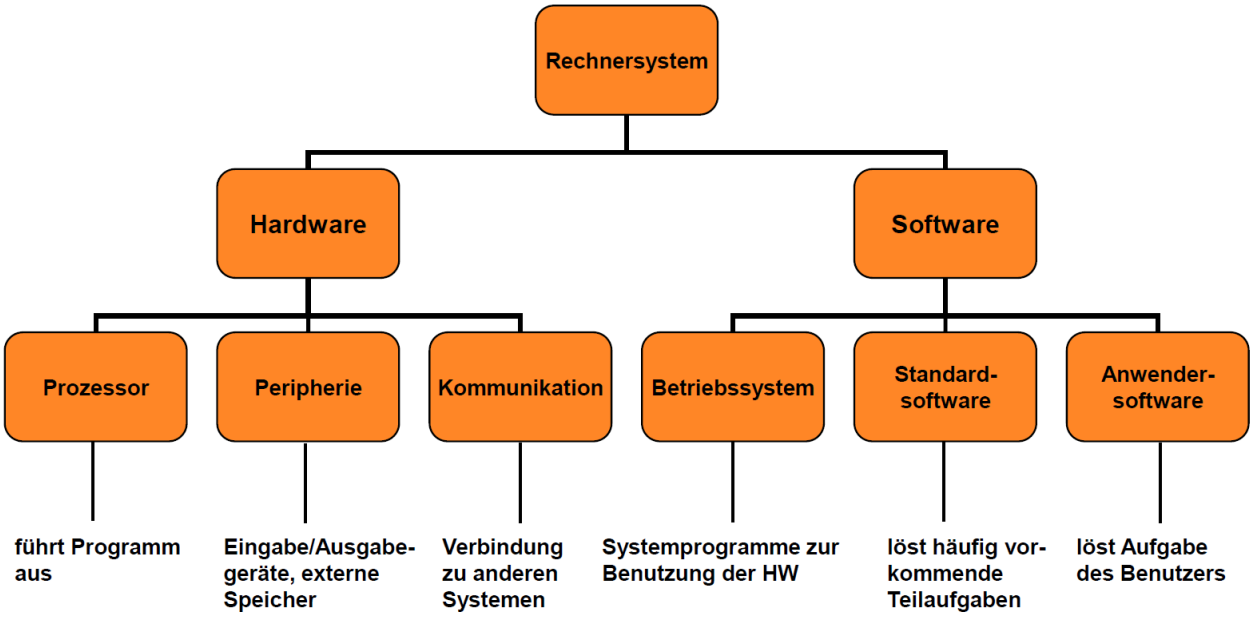
Sichtweise auf Programme



Quelle: KIT IMI

Betriebssystem

Bindeglied zwischen der Hardware eines Computers und dem Anwender bzw. seinen Programmen.



Betriebssystem: Klassifikation

nach Betriebsart

- Stapelverarbeitung (batch processing) — z.B. IBM OS/390, z/OS, BS2000
- Dialogbetrieb (interactive processing) — z.B. MS-DOS, MS-Windows, UNIX, LINUX
- Echtzeitverarbeitung (real time processing) — z.B. VxWorks, VRTX, QNX, OSEK/VDX
- Verteilte Verarbeitung (distributed processing) — z.B. Amoeba, MACH, Novell Netware

nach Anzahl der Benutzer

- Einzelnutzer-System (Single User System) — z.B. MS-DOS, MS Windows 95/98/ME
- Mehrnutzer-System (Multi User System) — z.B. UNIX, Linux, IBM z/OS, BS 2000, OpenVMS

nach Anzahl der Aufträge

- Einzelprozess-System (Single Tasking System) — z.B. CPM, MS-DOS
- Mehrprozess-System (Multi Tasking System) — z.B. UNIX, Linux, IBM z/OS, BS 2000, OpenVMS, VxWorks, VRTX, LynxOS

Betriebssystem typische Funktionen

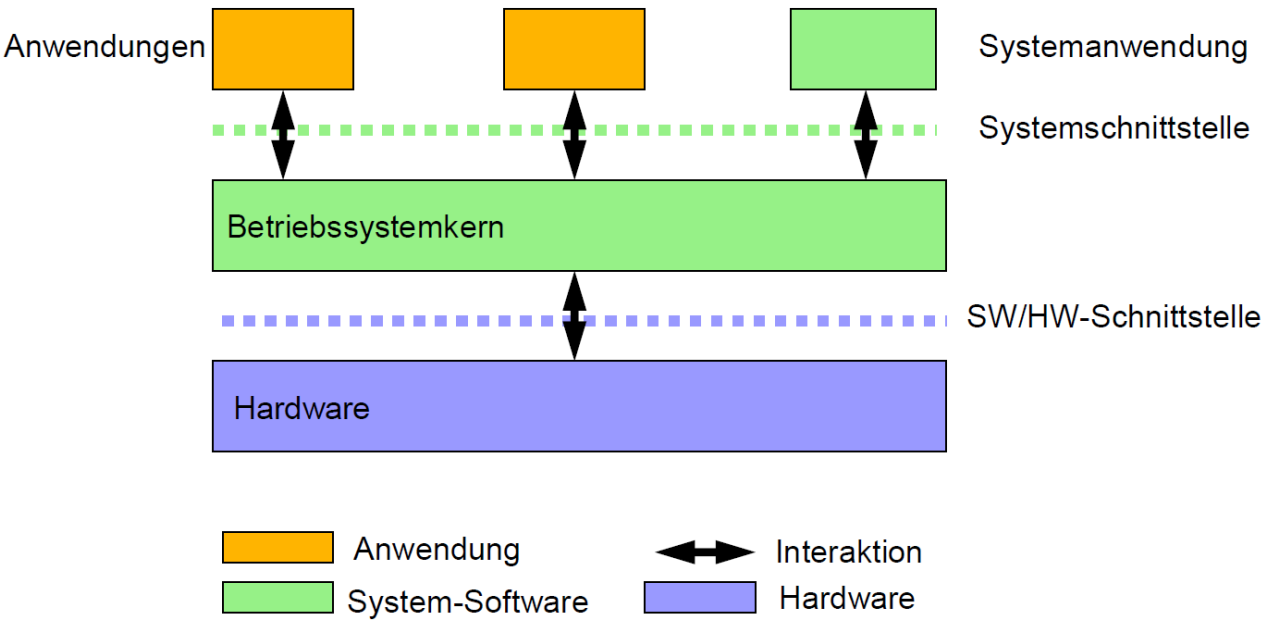
- Benutzerverwaltung
- Auftragsverwaltung
- Prozessverwaltung
- Betriebsmittelverwaltung
- Prozessorverwaltung (Scheduling)
- Hauptspeicherverwaltung
- Dateiverwaltung
- Ein-/Ausgabe-Steuerung
- Kommunikation mit der Umgebung

Systemsoftware: umfasst alle Programme, die eine effiziente und komfortable Benutzung eines Computers ermöglichen:

Programmierumgebungen: Editoren, Compiler, Linker, Lader, Testhilfen

Dienstprogramme: Suchen, Kopieren, Sortieren, Installieren, Konfigurieren, Administrieren

Betriebssystem zwischen Anwendung und Hardware



Zurück zu was ist eine Programmiersprache?

- Programmiersprache: Notation für ein Computerprogramm
  - Darstellung während und nach der Entwicklung (Programmierung)
  - Übermittlung des resultierenden Programms zur Ausführung an Rechensystem
- Programmiersprache muss für maschinelle Analyse geeignet sein (zahlreiche Einschränkungen) und muss lesbar sein durch den Menschen
- Programmiersprachen basieren auf „formalen Sprachen“

Formale Sprachen

definieren eine bestimmte Menge von Zeichenketten, die aus einem Zeichenvorrat zusammengesetzt werden können

- Grundlage für höhere Programmiersprachen
- Voraussetzung für Compilerbau
- Produktionsregeln und Eindeutigkeit

Eine Sprache besteht aus einem Alphabet von Zeichen, wobei die Wörter einer Sprache durch (formale) Regeln - der Grammatik - gebildet werden. Die Grammatik für Programmiersprachen wird Syntax genannt, während die inhaltliche Bedeutung durch die Semantik ausgedrückt wird.

Die Nutzung einer Programmiersprache ist eine formalisierte Methode, um Anweisungen an einen Computer zu übermitteln.

Programmiersprachen müssen auf einer eindeutig definierten Aussageform basieren, die es erlaubt, dem Computer in eindeutiger Weise zu sagen, was er tun soll.

Der Programmiersprache kommt dabei die Rolle einer „Lingua Franca“ zu, die sowohl der menschliche Programmierer als auch der Computer versteht.

Programmieren ist eine Tätigkeit, bei der versucht wird, durch systematischen Einsatz einer gegebenen Programmiersprache ein gestelltes Problem zu lösen.

• Ein Programm ist ein Algorithmus formuliert in einer Programmiersprache.

- Maschinensprachen, bzw. Assemblersprachen sind auf einen bestimmten Prozessor optimiert,

- Höhere Programmiersprachen erlauben eine problemnahe Formulierung

Programmiersprachen basieren auf Formalen Sprachen

- Wörter werden aus gegebenem Alphabet gebildet
  - Bildungsregeln für Wörter

Die **Syntax** einer Programmiersprache beschreibt die Menge der erlaubten Zeichenketten für Programme

Die **Semantik** einer Programmiersprache definiert die Bedeutung der einzelnen Sprachkonstrukte (meist textuelle Beschreibung)

Die **Grammatik** einer Programmiersprache legt die Produktionsregeln, also den Zusammenhang zwischen den Zeichen und den Wörtern, fest.

(Formales Beschreibungsmittel: Backus-Naur-Form, Syntaxdiagramme)

eines entworfenen Algorithmus.

• Zur Entwicklung der Anwendungsprogramme verwenden die Programmierer nach Möglichkeit höhere Programmiersprachen.

Durch Systemprogramme (Interpreter oder Compiler) wird der Quellcode automatisch vom Rechner in Maschinenbefehle übersetzt,

so dass diese schließlich von der Hardware (dem Prozessor) abgearbeitet werden können.

Programmiersprachen basieren auf Formalen Sprachen

- Wörter werden aus gegebenem Alphabet gebildet
  - Bildungsregeln für Wörter

Alphabet, Grammatik, Syntax, Semantik

Alphabet { a, b, c, d, e, f, h, i, l, n, p, s, t, o }
Zulässige Wörter:
„affe“ „otto“ „stop“

~~„sTop“~~
„etit“ „ist“ „toll“

Alphabet { a, b, c, d, e, f, h, i, n, p, t, 0, 1, 2, … , 9, , (,), [,], <, >, =, +, -, *, / }

if a[pi3,17] > 3.14 * (b+5) then do

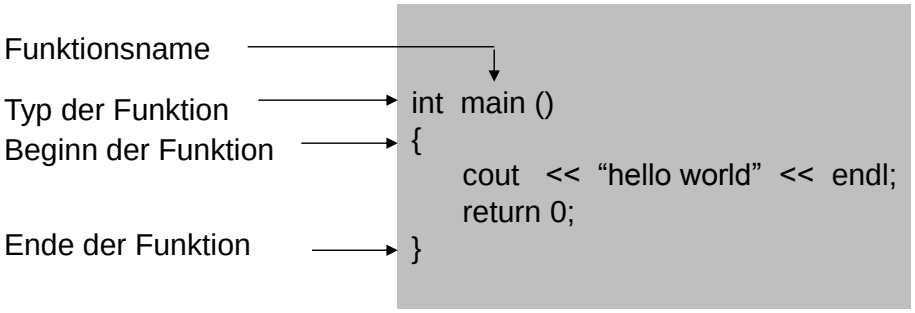
Wie erwähnt müssen Programmiersprachen einer strikten, eindeutigen Grammatik folgen. Eine solche künstliche Sprache wird als formale Sprache bezeichnet. Die formalen Sprachen bilden daher einen eigenen Zweig der Informatik. Als Voraussetzung an eine Programmiersprache muss die Eindeutigkeit gelten: Jede Anweisung muss für den Compiler unmissverständlich sein, um den Maschinencode daraus in eindeutiger Weise erstellen zu können!

Für den Programmierer muss verständlich sein, wie er gültige Befehle verfasst, und diese Bildungsregeln müssen für den Compiler entsprechend in verarbeitbare Formen umgewandelt werden können.

Ein erstes C++-Programm

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

int  main ()
{
     cout   <<  “hello world”  <<  endl;
     return 0;
}
```



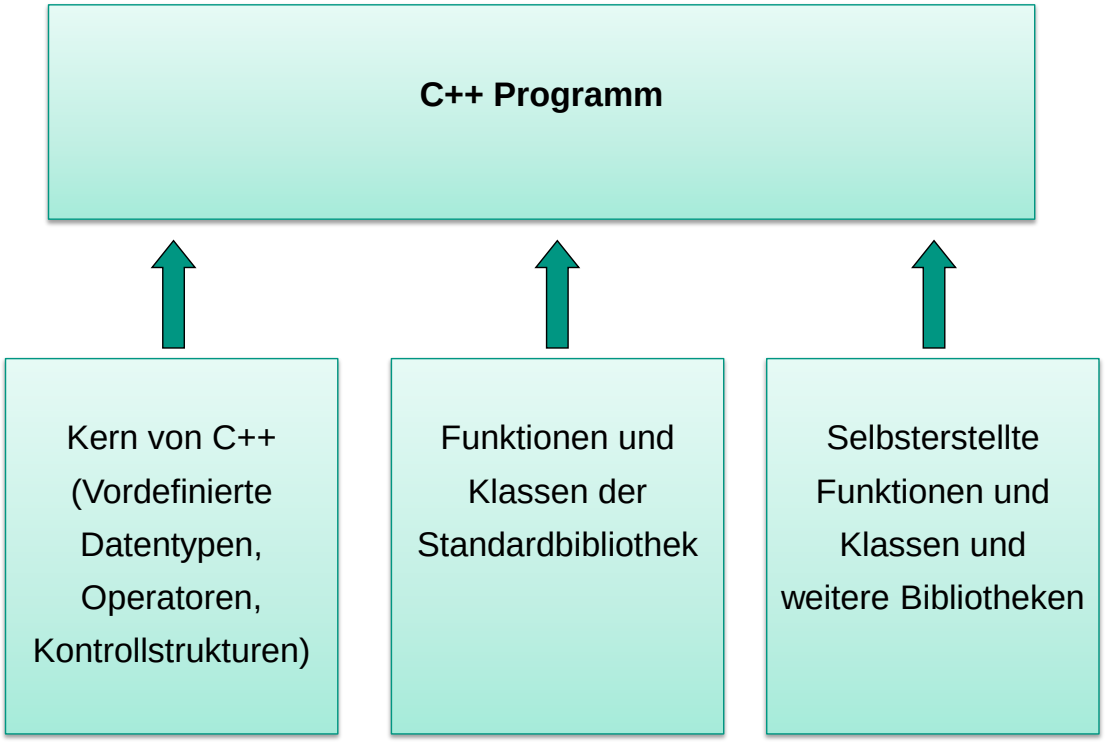
Funktion

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

Um Wörter einer Sprache bilden zu können, muss ein Alphabet definiert sein. Bildungsregeln legen fest, wie aus den Elementen des Alphabets (oder der Alphabete) ein Wort zusammengesetzt werden darf. Ein Alphabet ist zumeist eine Menge von Zeichen oder Wörtern.

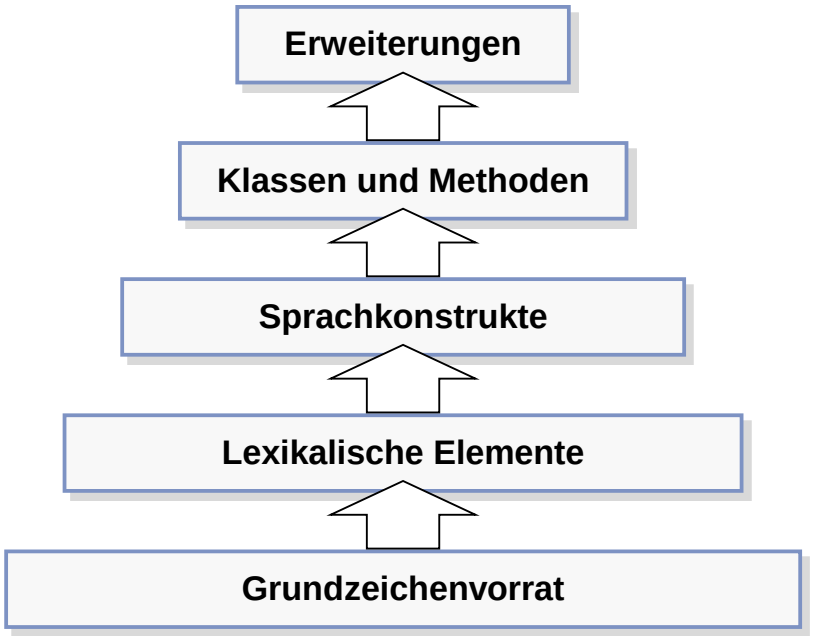
Formale Sprachen erlauben es, nicht nur Zeichen aus Alphabeten zu Wörtern zusammenzusetzen, sondern Wörter durch Verknüpfung miteinander zu Aussagen. Diese Verknüpfung wird durch Grammatik repräsentiert. Es gibt verschiedene Formen, die Wort- und Aussagebildung sowie gültige Alphabete zu definieren. Dazu gehört die Backus-Naur-Form genauso wie die Syntax-Diagramme.

Bestandteile eines C++ Programms



Kern von C++ / Sprachaufbau

Kern von C++ (Vordefinierte Datentypen, Operatoren, Kontrollstrukturen)



Alphabet: 7-bit ASCII-Tabelle

American Standard Code for Information Interchange (ASCII) ist eine 7-Bit-Zeichenkodierung; sie entspricht der US-Variante von ISO 646 und dient als Grundlage für spätere auf mehr Bits basierenden Kodierungen für Zeichensätze.

	-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	LF	VT	FF	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

Zwischenübung: ASCII Kodierung

- Kodieren Sie folgende Zeichenkette mit dem ASCII Code in hexadezimaler Darstellung:
  - Hello IT! ←→

	-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	LF	VT	FF	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

Lexikalische Elemente Übersicht

Formale Sprache

- Kommentare
- Begrenzungszeichen
- Reservierte Schlüsselworte
- Namen und Konventionen
- Konstanten und Variablen
- Escape-Sequenzen
- Elementare Datentypen

Lexikalische Elemente

- Kommentare
  - Syntax :

```
// beliebiger Text aus dem Zeichensatzvorrat von C++
```

  - Beispiele :

```
// Ein Kommentar beginnt mit // und reicht
// bis zum Zeilenende.
// Er kann alle moeglichen Zeichen (*'_<->) beinhalten
// und über mehrere Zeilen gehen, wenn er mit /* … */
// begrenzt wird
```

  - Zur Verdeutlichung des Programms (menschenlesbar)
  - Meist bedeutungslos für Compiler

Lexikalische Elemente

- Trenn- und Begrenzungszeichen
  - Dienen zur Trennung aufeinanderfolgender lexikalischer Elemente
  - Einzelzeichen (neben Leerzeichen) :

```
(  )  [  ]  {  }  |  &  !  #  .  ,  :  ;  /  %  *  -  <  =  >  +
```

  - Zusammengesetzte Zeichen :

```
>=   <=   ==   !=   &&  ||  ++  --  +=  -=  *=  /=
```

(Aufzählung ist unvollständig)

Lexikalische Elemente

- Befehlskette ohne Trenn- und Begrenzungszeichen:

```
inta=5b=7ifa>=ba-belseb-a
```

- → Funktion nicht interpretierbar
- Befehlskette mit Trenn- und Begrenzungszeichen:

```
int a=5, b=7; if(a>=b){a-b;} else{b-a;}
```

- → Betragsberechnung
- ABER: Kein guter Programmierstil!

Lexikalische Elemente

- Bezeichner ("identifier")
  - Namen von Variablen, Kostanten, Objekten, Typen usw.
  - Bezeichner bestehen aus Zeichen des 7-bit ASCII Zeichensatzes
    - Buchstaben
    - Ziffern
    - Einzelnen Unterstrichen
    - Keine Leer- und weitere Sonderzeichen erlaubt!
  - Bezeichner sind „case-sensitive“
  - Erstes Zeichen muss ein Buchstabe oder ein Unterstrich sein
  - Unterstrich (“_”) darf nicht
    - zweimal unmittelbar aufeinanderfolgen (.Net Schlüsselworterweiterungen __abstract, __event, …)
    - am Ende eines Bezeichners stehen
  - Bezeichner dürfen keine reservierten Worte sein

Schlüsselworte in C++

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

Beispiele für Namen

gültig:

a	DM	dm	VOID
_var	SetTextColor		
B12	top_of_window		
ein_sehr_langer_name1234567890			

ungültig:

goto	586_cpu	Hans-Otto	void
zähler	true	US$	

Lexikalische Elemente

- Elementare Datentypen (Literale, Größen)
  - numerische Größen
  - Zeichen (character), Zeichenketten (strings), „Bit-Strings“

Character

'A'
'a'
'@'
' '
'1'
'+'
'x'

Strings

"string \"IN\" a string"
"This is a string"
"xy"
""

Integerzahlen

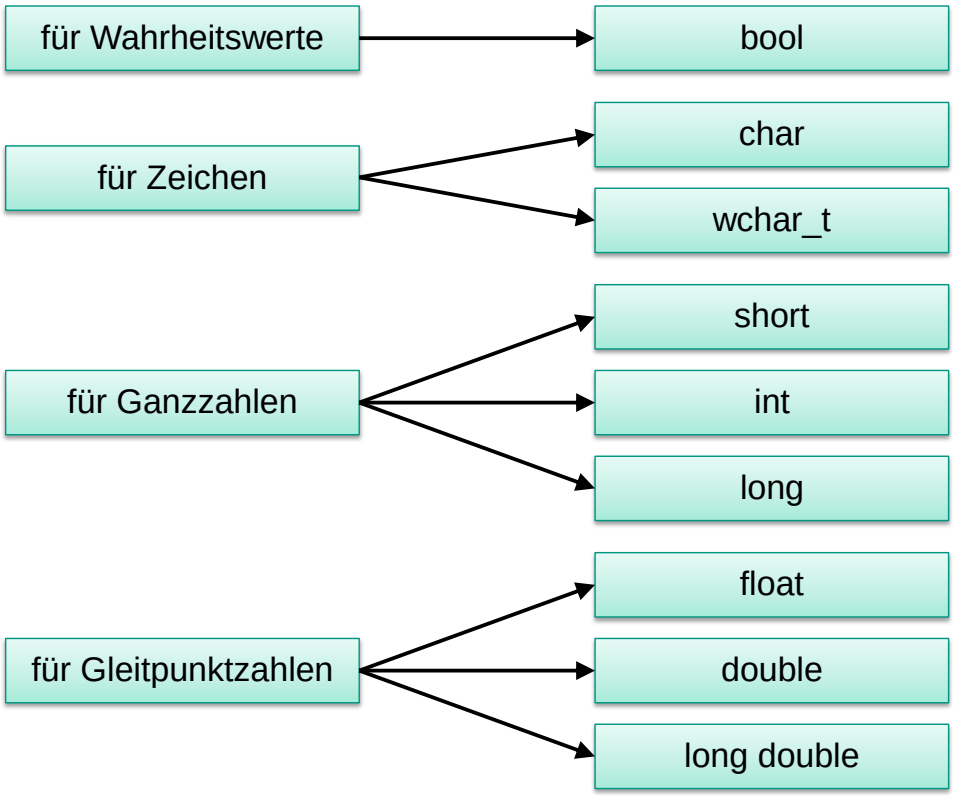
2
3E4
100000
2153E3

Fließkommazahlen

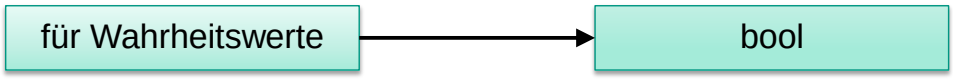
2.3
0.04
1.4E12
1.00345E-6

- Compiler muss erkennen (Syntax, Semantik) um welchen Datentyp es sich handelt

Elementare Datentypen



Elementare Datentypen



Für Boolesche Konstanten in C++ Schlüsselworte:

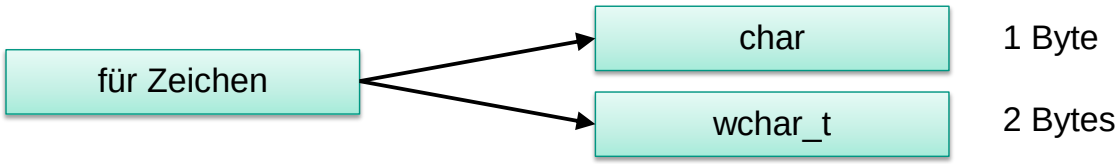
true (wahr)

false (falsch)

Relationale und logische Operatoren werden dazu benutzt, boolesche Werte zu produzieren und werden häufig zusammen angewendet (siehe weiter unten bei Operatoren)

in C (C++) jede Zahl ungleich null wird als true interpretiert,
die Null entspricht dem Wert false (in C gibt es keinen Typ bool)

Elementare Datentypen char, wchar_t



- dienen zur Speicherung von Zeichencodes (1 Byte, 2 Bytes)
- jedem Zeichen ist eine ganze Zahl zugeordnet
- Zuordnung wird festgelegt durch verwendeten Zeichensatz
  (C++ legt keinen Zeichensatz fest)
  üblich 7-bit ASCII Code (Codes 0 bis 31 Steuerzeichen
  32 bis 127 druckbare Zeichen)
- erweiterter Zeichensatz ANSI 8-bit Code (z.B Umlaute)
- erweitert Unicode: 16 Bit Code ca. 35000 Zeichen aus 24 Sprachen
  - 0x0000 – 0x007F entspricht ASCII

Beispiele für Zeichenkonstanten (character)

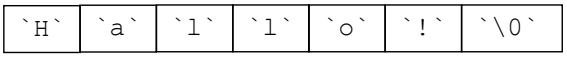
Konstante	Zeichen	Wert der Konstanten (ASCII-Code dezimal)
`'A'`	Großbuchstabe A	65
`'a'`	Kleinbuchstabe a	97
`' '`	Leerzeichen	32
`'.'`	Punkt	46
`'0'`	Ziffer 0	48
`'\0'`	String-Ende-Zeichen	0

Beispiele für Zeichenkette (string Konstante)

String-Konstante: im Speicher abgelegt
als Byte-Folge

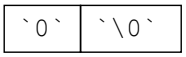
Stringendezeichen \0
Escape Sequenz

“Hallo!“

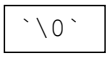


“Dies ist ein String“

“0“



““



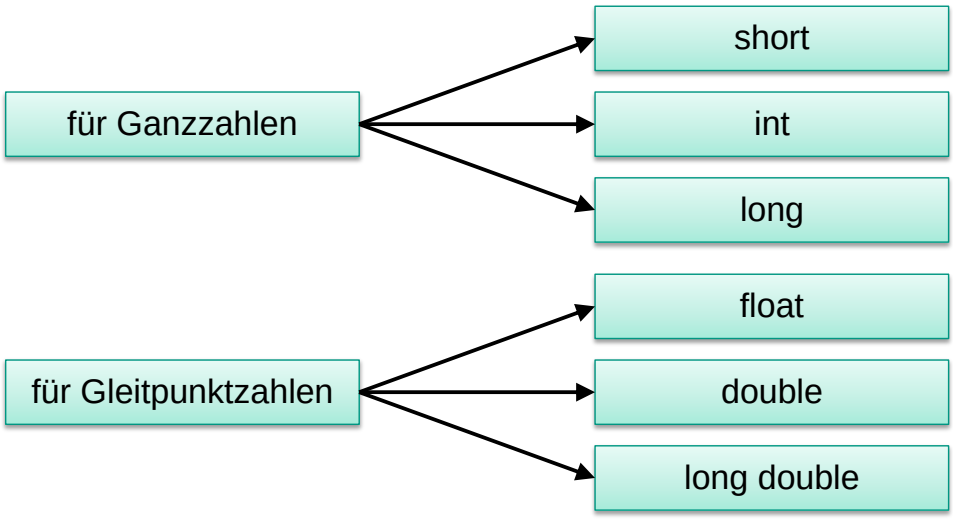
Escape Sequenzen finden in strings Verwendung für grafisch nicht darstellbare Zeichen:

```
cout << "\nDies ist\t ein String\n\t\t"
    "mit \"vielen\" Escape-Sequencen!\n"
```

„Escape-Sequenzen“ (Steuerzeichen und Sonderzeichen)

Einzelzeichen	Bedeutung	ASCII-Wert (dezimal)
\a	alert (BEL)	7
\b	backspace (BS)	8
\t	horizontal tab (HT)	9
\n	line feed (LF)	10
\v	vertikal tab (VT)	11
\f	form feed (FF)	12
\r	carriage return (CR)	13
\"	“	34
\'	‘	39
\?	?	63
****	\	92
\0	Stringende-Zeichen	0
\ooo (bis zu drei Oktalziffern)	numerischer Wert eines Zeichens	ooo (oktal!)
\xhh (Folge von Hex-Ziffern)	numerischer Wert eines Zeichens	hh (hexadezimal!)

Elementare Datentypen



Die ganzzahligen Datentypen

Typ	Speicherplatz	Wertebereich
`char`	1 Byte	-128 bis +127 bzw. 0 bis 255
`unsigned char`	1 Byte	0 bis 255
`signed char`	1 Byte	-128 bis +127
`int`	2 Byte bzw. 4 Byte	-32768 bis +32767 bzw. -2147483648 bis +2147483647
`unsigned int`	2 Byte bzw. 4 Byte	0 bis 65535 bzw. 0 bis 4294967295
`short`	2 Byte	-32768 bis +32767
`unsigned short`	2 Byte	0 bis 65535
`long`	4 Byte	-2147483648 bis +2147483647
`unsigned long`	4 Byte	0 bis 4294967295

Die Datentypen für Gleitpunktzahlen

Typ	Speicherplatz	Wertebereich	kleinste positive Zahl	Genauigkeit (dezimal)
float	4 Bytes	±3.4E+38	1.2E-38	6 Stellen
double	8 Bytes	±1.7E+308	2.3E-308	15 Stellen
long double	10 Bytes	±1.1E+4932	3.4E-4932	19 Stellen

Arrays

Array: Mehrere Werte eines Datentyps in einer Variable ablegen.
Array Variable belegt zusammenhängenden Speicherbereich

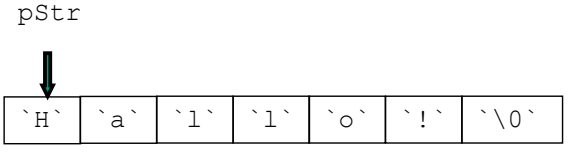
```
char str[7] = { `H`, `a`, `l`, `l`, `o`, `!`, `\0`};
```

oder äquivalent

```
char str[7] = "Hallo!";
```

Array Variable ist auch **Zeiger (Pointer)** auf das erste Element des Arrays im Speicher

```
char* pStr = str;
```



Arrays

Weitere Beispiele

- Array vollständig mit definierten Werten initialisieren

```
int iArr[5] = {1, 2, 3, 4, 5};
```

- Submenge des Array mit definierten Werten initialisieren, den Rest des Arrays mit default Wert (=0):

```
double dArr[4] = {1.1, 2.2};
```

- Alle Elemente mit default Wert initialisieren

```
long lArr[100] = {};
```

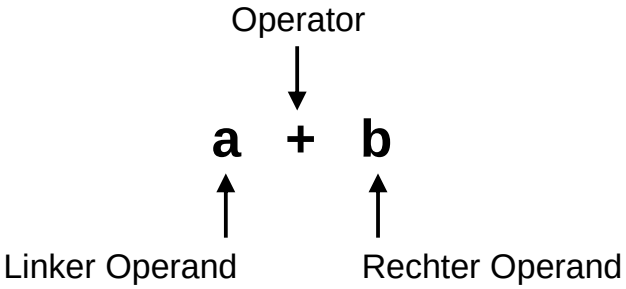
- Zugriff auf Array Elemente über Index-Operator

```
int temp = iArr[2];   // temp = 3
iArr[2] = iArr[3];    // iArr[2] = 4
iArr[3] = temp;       // iArr[3] = 3
```

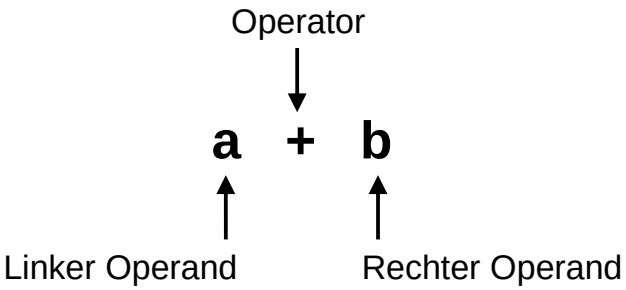

C++ Operatoren

Kern von C++ (Vordefinierte Datentypen, Operatoren, Kontrollstrukturen)

Operatoren und Operanden



Binärer Operator und Operanden



Die binären arithmetischen Operatoren

Operator	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulodivision

Operatoren

Die unären arithmetischen Operatoren

Operator	Bedeutung
`+ -`	Vorzeichenoperator
`++`	Inkrement-Operator
`--`	Dekrement-Operator

Vorrang der arithmetischen Operatoren

Priorität	Operator	Zusammenfassung
hoch ↕ niedrig	`++ --` (Postfix)	von links
	`++ --` (Präfix) `+ -` (Vorzeichen)	von rechts
	`* / %`	von links
	`+` (Addition) `-` (Subtraktion)	von rechts

Vergleichsoperatoren

Operator	Bedeutung
`<`	kleiner
`<=`	kleiner gleich
`>`	größer
`>=`	größer gleich
`==`	gleich
`!=`	ungleich

Beispiel:

Vergleich	Ergebnis
`5>=6`	False
`1.7<1.8`	True
`4+2==5`	False
`2*4!=7`	True

Vorrang der Vergleichsoperatoren

Priorität	Operator
hoch ↕ niedrig	arithmetische Operatoren
	`< <= > >=`
	`== !=`
	Zuweisungsoperatoren

Logische Operatoren

Operator	Bedeutung
`&&`	UND
`||`	ODER
!	NICHT

Boolesche Operatoren für zusammengesetzte Bedingungen

x	y	logischer Ausdruck	Ergebnis
1	-1	`x <= y || y >= 0`	false
0	0	`x > -2 && y == 0`	true
-1	0	`x && !y`	true
0	1	`!(x+1) || y-1 > 0`	false

Wahrheitstafel

A	B	A!	A && B	A || B
true	true	false	true	true
true	false	false	false	true
false	true	true	false	true
false	false	true	false	false

Die Operanden der booleschen Operatoren sind vom Typ bool. Als Operanden sind aber auch beliebige Ausdrücke zulässig, deren Typ in bool konvertiert werden kann (alle arithmetischenTypen). Operand wird in false konvertiert, wenn sein Wert 0 ist, jeder von 0 verschiedene Wert wird zu true konvertiert!

logische Bitoperatoren

Operator	Bedeutung
&	UND
I	ODER
~	NICHT
^	Exklusiv ODER
<<	Links-Shift
>>	Rechts-Shift

- Für bitcodierte Daten einzelne Bits lesen oder ändern oder wortweise verarbeiten, daher hohe Speicherplatzeffizienz und Rechenzeiteffizienz
- Operanden müssen ganzzahligen Datentyp haben

- Nicht verwechseln mit logischen Operatoren, diese arbeiten nicht bitweise
  1 && 2 hat Wert true
  1 & 2 hat Wert 0

Unsigned integer a, b, c;	Bitmuster
a = 5;	0 0 . . . . . 0 0 0 0 0 1 0 1
b = 12;	0 0 . . . . . 0 0 0 0 1 1 0 0
c = ~a;	1 1 . . . . . 1 1 1 1 1 0 1 0
c = a & b;	0 0 . . . . . 0 0 0 0 0 1 0 0
c = a | b;	0 0 . . . . . 0 0 0 0 1 1 0 1
c = a ^ b;	0 0 . . . . . 0 0 0 0 1 0 0 1
c = b << 3;	0 0 . . . . . 0 1 1 0 0 0 0 0
c = b >> 2;	0 0 . . . . . 0 0 0 0 0 0 1 1

Bei Linksschieben nachschieben immer 0-Bits

Bei Rechtsschieben von links 0-Bit nachschieben wenn der linke Operand vom Typ unsigned ist oder nicht negativen Wert hat. Andernfalls Compiler-abhängig ob von links 0-Bits (logischer Shift) oder mit Vorzeichen (arithmetischer Shift) aufgefüllt wird (normal arithmetischer Shift).

C++ Kontrollstrukturen

Kern von C++
(Vordefinierte Datentypen, Operatoren, Kontrollstrukturen)

Anweisungen zur Kontrolle des Programmflusses

Schleifen mit while, do-while und for
Verzweigung mit if-else und switch
Bedingungsfreie Sprünge mit continue, break

Programm Demo mit Visual Studio 2010

Karlsruher Institut für Technologie

Institut für Technik der Informationsverarbeitung

Institutsleitung
Prof. Dr.-Ing. K. D. Müller-Glaser
Prof. Dr.-Ing. J. Becker
Prof. Dr. rer. nat. W. Stork

Vorlesung: Informationstechnik (IT)

Prof. Dr.-Ing. K.D. Müller-Glaser

Entwicklungsumgebung

Übersicht

Karlsruher Institut für Technologie

- Die Integrierte Entwicklungsumgebung
- Teile der Entwicklungsumgebung
- Vom Code zum Programm
- Ablauf Erstellung eines Programms

Entwicklungsumgebung

- IDE - Integrated Development Environment
  - Sammlung aller Werkzeuge, die dem Entwickler helfen, seine Programme zu erstellen

- Entwickeln ist mehr als Codeschreiben

- Entwickler brauchen mehr als Codeeditoren

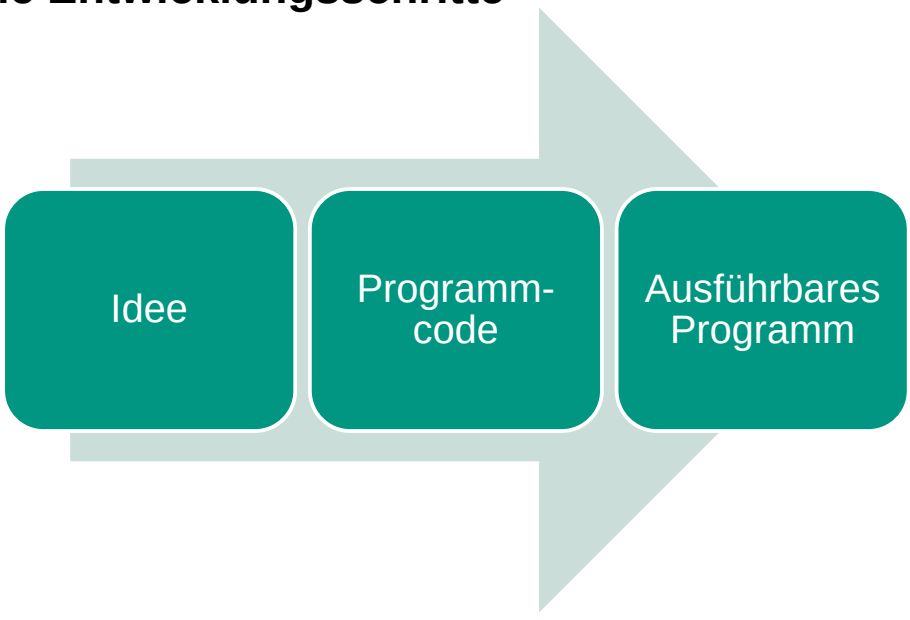
Funktionen der IDE

Unterstützung des Entwicklers

- Von früher Planungsphase
- Über das Schnittstellendesign
- Über das Programmieren
- Über das Unit-Testen
- Über das Integrieren
- Über das Übersetzen
- Bis hin zur Langzeitwartung

- Beim Dokumentieren
- Beim Versionieren

Minimale Entwicklungsschritte

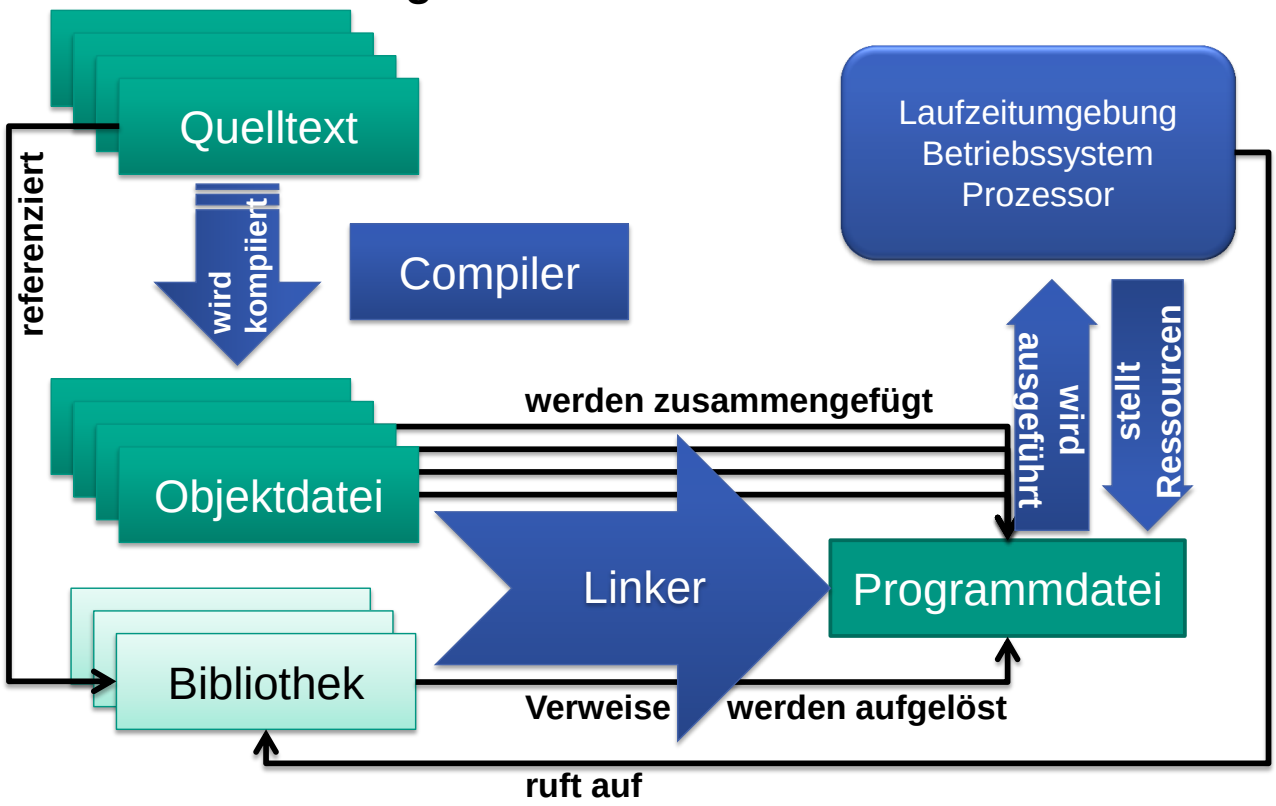


- Programmierer setzt Idee, Anforderung um in Code
- Compiler übersetzt Code
- Linker linkt Objektdateien

Minimale Entwicklungsumgebung

- Programmcodeeditor
- Compiler
- Bibliotheken
- Linker
- Laufzeitumgebung

Vom Code zur Programmdatei



Zwischenübung01: Elemente einer Programmierumgebung

Ordnen Sie die folgenden Aufgaben den richtigen Komponenten zu:

Compiler

Linker

Übersetzt Programmcode in Maschinencode

Verknüpft Objektdateien zu Hauptdatei

Bindet Bibliotheken ein

Erzeugt Objektdateien

Erstellt ausführbare Datei

Entwicklungsumgebungen Umfang

- Planung
- Programmierung
  - Intelligente Editoren
  - Versionierung
- Übersetzung
- Debugging
- Testen
- Dokumentation

Programmeditor

- Ermöglicht sauberes Schreiben von Code
- Auto-Formatierer
- Syntax Highlighting
- Autocompletion
- Integration
  - Versionierung
  - Debugger

Compiler

- Plattformspezifisch
- Übersetzt Programmiersprache in Maschinencode
- Erzeugt aus einzelnen Programmquellen Objektdateien
- In diesen Objektdateien sind noch nicht alle Sprungbefehle und Funktionsaufrufe aufgelöst
  - das erledigt der Linker

Planung:

Graphische Schnittstellenplanung (Einsatz UML – Unified Modeling Language)

Testplanung

Programmierung:

„Intelligente“ Editoren

Versionierung

Darstellen von Versionsunterschieden (diff, patch)

Sicherung und Vereinigung von im Team entwickelter Software (SVN, git)

Übersetzung:

Erstellen von Compilerskripten (Apache Ant, Makefiles, Autohell)

Rückmeldung und Hilfestellung bei Compilerfehlern

Konfiguration des Linkers

Debugging:

Unterstützung von Breakpoints

Speicherüberwachung

Auflösen von Variablennamen

Testen:

Unit-Testing

Module-Testing

Dokumentation:

Automatisierte Schnittstellendokumentation (z.B. Javadoc, Oxigen)

Ticketsysteme für Bugreports

Linker

- Verknüpft Objektdateien
- Bettet aus Bibliotheken verwendete Funktionen ein
- Dynamic Binding: Verweis auf ladbare Bibliotheken
- Bildet ausführbare Datei

Der Programmeditor ist ein Programm, um die textlichen Programme zu verfassen. Seine Aufgabe besteht darin, dem Programmierer dabei zu helfen, *sauberen, schönen* und *effektiven* Code zu entwickeln und frühzeitig syntaktische Fehler zu eliminieren.

Zu diesem Zweck beinhaltet er zumeist einen Auto-Formatierer, der etwa Codeblöcke automatisch einrückt, Kommentare fortsetzt usw. und so besser lesbaren und damit wartbaren Code ermöglicht.

Syntax Highlighting dient dazu, den Quellcode programmiererfreundlich anzuzeigen: Wichtige Syntaxelemente wie z.B. die Schleifenanweisung werden farblich hervorgehoben. Autocompletion ist das Feature, zu erraten, was der Programmierer schreiben möchte: Kennt die IDE die von ihm eingesetzten Objekte, kann es ihm eine Liste anbieten, aus der er die zu verwendende Funktion mit wenigen Tastatureingaben auswählen kann. Dies erspart viel Nachschlagen und vor allem syntaktische Fehler.

Der Programmeditor ist üblicherweise der zentrale Teil einer IDE, er integriert daher die anderen Teile unter seiner Benutzeroberfläche.

Offensichtlich ist es sehr vorteilhaft, wenn die Code-Versionierung hier integriert wird. Mit einigen Aufrufen kann der Programmierer seine Datei in das Versionierungssystem einpflegen (checkin/commit) oder den aktuellen Entwicklungsstand von dort beziehen (checkout/update).

Der Debugger ermöglicht es, zur Laufzeit Rückschlüsse vom Verhalten des Programms auf die verursachenden Codezeilen zu schließen und kann bei der Suche nach logischen Fehlern, die vom Compiler nicht erkannt werden können, sehr hilfreich sein.

Debugger

- Arbeitet mit fertigem Objektcode
- Instrumentalisiert Programmcode
- Fängt Fehler im Programmablauf ab und liefert Informationen darüber, wo diese auftraten
- Kann mit eingebetteter Debugging-Information betroffene Codezeile bestimmen

Der Compiler ist das Programm, das die verwendete Programmiersprache „versteht“: Er erstellt aus dem Quelltext Maschinencode.

Dabei übersetzt er die abstrakten Anweisungen der Programmiersprache in eine Vielzahl von Maschinenanweisungen, fügt u.U. automatisch Speicherverwaltung hinzu, und führt eventuell (zum Teil sehr leistungsfähige) Optimierungen durch.

Von Klassen (siehe Objektorientierung) stellt der Compiler entsprechende interne Darstellungen her, um in der Lage zu sein, Vererbung oder speziell das sogenannte „late binding“ zu ermöglichen.

Funktionsaufrufe aus anderen Objektdateien oder Bibliotheken kann der Compiler aber nicht direkt auflösen, da er nur die einzelne Quelltextdatei (und alle darin eingebundenen Header) sieht. Die Verknüpfung mehrerer Objektdateien übernimmt der Linker.

Entwicklung – typischer Ablauf I

- Erstellen eines Projekts
- Einbinden der benötigten Bibliotheken
- Festlegung der Ordnerstruktur, der Zielplattform, der Compilerversion
- Definition und Dokumentation von Programmschnittstellen
- Programmieren

Der Debugger hilft dem Programmierer beim Entfernen von Fehlern (Bugs). Er nimmt dabei das in Maschinencode übersetzte Programm (das häufig mit speziell für den Debugger relevanten Informationen angereichert ist), und führt es Schritt für Schritt aus, so dass der Programmierer praktisch eins zu eins überprüfen kann, wie eine beliebige einzelne Zeile seines Codes den Zustand des Programms verändert. Dazu hat der Compiler die Möglichkeit, Werte von Variablen zu überwachen. So ist es zum Beispiel möglich, herauszufinden, an welcher Stelle eine Variable auf einen ungültigen Wert gesetzt wird.

Entwicklung – typischer Ablauf II

- Compilieren & Linken
- Debugging
- Testen
  - Testautomatisierung
  - Integration von anderen Qualitätssicherungstools

Coding Guidelines

- Richtlinie: Kein technischer Zwang
- Lesbarkeit führt zu Wartbarkeit
- Vorhersehbarkeit bei Namen führt zu Wiederverwendbarkeit
- Struktur führt zu Ordnung führt zu Fehlervermeidung

Erstellen eines Projekts:

Zunächst legt der Entwickler fest, welche Art von Programm er schreiben will, gibt dem Programm einen Namen und legt Speicherpfade etc. fest.

Einbinden der benötigten Bibliotheken:

Der Programmierer legt dann fest, welche externen Bibliotheken er einbinden möchte. Die IDE sorgt dafür, dass dem Compiler und Linker entsprechende Anweisungen bereitgestellt werden, und betrachtet gegebenenfalls die eingebundenen Bibliotheken für die „Autocompletion“.

Festlegung der Ordnerstruktur, der Zielplattform und der Compilerversion:

Entsprechend seiner geplanten Softwarearchitektur legt der Programmierer eine Struktur an und teilt dem Compiler mit, welche Versionen der Compiler er einsetzen möchte, etc.

Definition und Dokumentation von Programmschnittstellen:

Entsprechend der Architektur werden häufig zunächst Schnittstellen (header etc.) festgelegt, bevor diese im nächsten Schritt implementiert werden.

Programmieren:

Beim Programmieren legt der Programmierer seinen Fortschritt regelmäßig in einem Versionisierungssystem ab, damit er später nachvollziehen kann, wann er oder andere Teammitglieder bestimmte Programmteile geändert haben, und um anderen Teammitgliedern zu ermöglichen, in ihrem Programmteil seinen Fortschritt einzusetzen.

Da die IDE dies komfortabel anbietet, kann der Programmierer relativ häufig sein Programm compilieren. Dabei wird er auf Fehler aufmerksam, z.B. wenn ein Compiler einen Typfehler oder Ähnliches anzeigt.

Zusammenfassung

Tool	Aufgabe
IDE	Integrierte Entwicklungsumgebung
Compiler	Übersetzt Quelltext in Maschinencode oder zunächst in Assembler
Linker	Verknüpft vom Compiler erstellte Objektdateien miteinander und mit Bibliotheken
Debugger	Unterstützt den Entwickler bei der Fehlersuche, in dem er Einblick in den Programmzustand zur Laufzeit gewährt
Bibliotheken	Stellen Funktionalität zur Verfügung, die dem Programmierer das Erstellen von Software ermöglicht, ohne jedesmal das Rad neu zu erfinden
Unit Tests	Funktionselementweises Prüfen von Software

Fazit

- IDEs sind heute hochentwickelte Unterstützer bei der Software-Entwicklung
- Der Compiler übersetzt Code in Maschinensprache
- Der Linker setzt Programme zusammen
- Die Entwicklungsumgebung bietet einen durchgängigen Workflow, der zur Qualität beiträgt