Charakterisierung von C:

Grundsätzliches:

C ist eine kompakte, strukturierte, typisierte, hardwarenahe, portable, höhere Programmiersprache;
C verwendet wenige Symbole, hat wenige Befehle und reservierte Schlüsselwörter;
die Ein-/Ausgabe ist nicht im Befehlsumfang enthalten, sondern wird in den Funktionen der Standardbibliothek realisiert;

Quellcode-Übersetzen-Syntax-Semantik:

C-Quellcode muss zur Grammatik (Syntax) von C konform sein, der Quellcode muss darüber hinaus auch die fachliche Anforderung realisieren (Semantik);
jede C-Anweisung (Ausnahmen: if, while, for, switch) muss mit einem Strichpunkt (Semikolon) abgeschlossen werden;
vom Programmierer geschriebener Quellcode muss vom Compiler in Objektcode übersetzt und dieser muss vom Linker mit Bibliotheken zu einem ausführbaren Programm gebunden werden;
C-Quellcode setzt sich zusammen aus Quellcode-Dateien (Endung: .c), die Header-Dateien einbinden (Endung: .h);
der Compiler ruft vor der eigentlichen Übersetzung den Präprozessor auf, der die Präprozessordirektive (stehen in einer eigenen Zeile, beginnen mit #, vorher sind nur Leerzeichen oder Tabulator erlaubt) auswertet;

Kommentare:

Quellcode, der nicht übersetzt werden soll, kann als Kommentar markiert werden (// markiert bis Zeilenende, /* bis erstem folgendem */);

Variablen-/Funktionsnamen:

Namen von Funktionen und Variablen dürfen nur aus Buchstaben (ausgenommen Umlaute, ß), Ziffern und dem Unterstrich gebildet werden, wobei der Name mit einem Buchstaben beginnen soll, die Groß-/Kleinschreibung wird beachtet, eventuell sind nur 31 Zeichen maßgeblich, die ersten 8 Zeichen sollen sich wenn möglich unterscheiden.

Blöcke-Verschachtelung:

C-Programme bestehen aus ineinander geschachtelten Blöcken, die in der Regel durch Einrückung nach rechts optisch hervorgehoben werden. Funktionen enthalten ihren Code in einem Block. Außerhalb von Blöcken sind nur Präprozessordirektiven und Variablendeklarationen und -definitionen sowie Funktionsdeklarationen (Prototypen) erlaubt.

Reservierte Schlüsselwörter:

Folgende Namen sind in C reserviert und dürfen nicht für die Benennung von Variablen, Konstanten, Typen bzw. Funktionen verwendet werden:

Variablen- und Funktionsdeklaration:

char, short, int, long, float, double, void; signed, unsigned; volatile, auto, register, static, const, extern

Typdeklaration:

struct, union, enum, typedef

Ablaufsteuerung:

if, else, switch, case, default, break, do, while, for, continue, goto, return

Typ-/Variablengröße bestimmen:

sizeof()

Grundgerüst/Struktur eines C-Programms:

```
//Einbinden der Funktionsdeklarationen für z.B. Ein-/Ausgabe durch den Präprozessor:
#include <stdio.h>
//Hauptfunktion main, kann entweder int oder void sein, bei int muss letzte Anweisung return
(gefolgt von einem arithmetischen Ausdruck) sein
int main()
{
	// Deklaration von Variablen:
	//Typname Variablenname mit optionaler Initialisierung
	// Bsp.: int zahl = -123;
	// Eingabe der erforderlichen Werte (scanf)
	// Verarbeitung der eingegebenen Werte
	// und Berechnung der Ergebnisse
	// Ausgabe der Ergebnisse (printf)
	// return 0; // 0 bedeutet ok, andere Werte Fehler/...
}
```

Variablendeklaration:

Eine Variablendeklaration beginnt mit einer Angabe zur Vorzeichenbehandlung (unsigned, nur bei ganzzahligen Datentypen), gefolgt vom Datentyp und dem Namen der Variablen. Hinter der Variablen muss für Felder und Zeichenketten (sind Zeichenfelder) ein Paar eckige Klammern ([]) angegeben werden. Die Angabe einer Zahl innerhalb der eckigen Klammern für die Anzahl der Elemente/Zeichen kann bei Initialisierung (=Startwertbelegung) entfallen. In diesem Fall wird so viel Platz reserviert, wie für den Initialwert erforderlich ist. Bei einer Zeichenkette wird dabei ein zusätzliches Byte für das abschließende Zeichenketten-Endesymbol '\0' reserviert.
Mehrere Variablen können durch Beistrich getrennt deklariert werden.
Beispiele:

```
int a=7;
float radius=3.7, flaeche=7.903, volumen=34.4e+2;
```

Konstantendeklaration:

Durch Voranstellen des Schlüsselworts const kann eine Konstante deklariert werden. Diese kann nur bei der Deklaration initialisiert, also mit einem Wert belegt werden. Durch die Präprozessordirektive #define festgelegte Namen für Werte werden symbolische Konstante genannt.
Beispiele:

```
#define LEN_MAX 23
#define ZEILEN_MAX 22
```

const int monate = 12;

Literale – unmittelbare Werte:

Im Programm unmittelbar zu verarbeitende Werte (Zahlen, Buchstaben, Texte) nennt man Literale (manche bezeichnen diese Werte auch als Konstante, was aber nicht stimmt).
Beispiele:
3.14159, -3.14e+12, 'a', "Zeichenkette"

Zahlen:

Zahlen werden im Quellcode unmittelbar geschrieben und vom Compiler als entsprechendes Bitmuster in den Maschinencode des erzeugten Programms eingetragen. Bei negativen Zahlen muss ein – vorangestellt werden.
Gleitkommazahlen werden am Dezimalpunkt (. und nicht ,) oder an der Angabe eines Exponenten (e bzw. E) erkannt und können in normaler Darstellung oder in Exponentialschreibweise notiert werden. Dabei darf das e klein oder groß geschrieben werden. Neben dezimalen Zahlen können auch oktale bzw. hexadezimale Zahlen verwendet werden, wobei oktale Zahlen mit einer Null und hexadezimale Zahlen mit einer Null gefolgt von einem großen oder kleinen x beginnen müssen. Dezimale Zahlen haben den Typ int bzw. long, falls int zu klein ist. Oktale bzw. hexadezimale Zahlen bekommen den kleinsten Typ aus der Reihe int, unsigned int, long oder unsigned long. Durch Anhängen eines oder mehrerer Buchstaben kann der Typ der Zahl erzwungen werden. L bzw. l erzwingen long, UL bzw. ul erzwingen unsigned long, LL bzw. ll erzwingen long long, ULL bzw. ull erzwingen unsigned long long, U bzw. u erzwingen unsigned int und f bzw. F erzwingen float.
Beispiele:
12, -2345, 45.349, 34609UL, 0278, 0377, 0x378, 0xaa55, 0xffff, -78.19304e+12, 1.5035e-3, 123L, 12345ULL

Zeichen:

Zeichen können als Buchstaben eingeschlossen in einfaches Hochkomma oder als Zahl geschrieben werden. Bei der Zahl muss es sich um den Zeichencode des üblicherweise verwendeten ASCII-Code handeln. Das Zeichen ' kann nur als '\'' geschrieben werden, d.h. dem Zeichen ' muss mit dem vorangestellten Backslash die Hochkomma-Bedeutung genommen werden. Die Zahl 0 (ASCII-Code des ersten Zeichens) und das Zeichen '0' (ASCII-Code 48) sind verschieden. Einzelne Zeichen haben den Datentyp int. 'A' hat den ASCII-Code 65 und 'a' 97.
Beispiele:
'a', 'Z', '5', '\0', '\n', '\t', ' ', '!', ...

Strings / Zeichenketten:

Zeichenketten sind in C als Felder von Zeichen realisiert und müssen in doppeltes Hochkomma " eingeschlossen werden. Eine Zeichenkette entspricht dabei einer Folge von Buchstaben, die nach dem letzten Zeichen das Zeichen '\0' als Endemarke enthält. Längere einzelne Texte, die nur durch Leerzeichen, Tabulatoren oder Zeilenunbrüche getrennt sind, werden vom Compiler zu einer Zeichenkette zusammengezogen.
Beispiele:
"Hallo Welt!", "\nHallo C-Neuling!\n", "Einwohneranzahl:\t%lu\n", "Hallo" "neue" "Welt!"

Escape-Sequenzen:

Zur Realisierung von speziellen Zeichen werden in C besondere Zeichenfolgen verwendet, die mit einem Backslash (\) beginnen. Den Zeichen ', " bzw. \ kann ihre spezielle Bedeutung durch Voranstellen eines \ genommen werden. Sie können damit auch im resultierenden Text vorkommen.

\0	\t	\n	\\	\'	\"
Zeichenkettenende	Horzontaler Tabulator	Zeilenschaltung	\ als normales Zeichen	' als normales Zeichen	" als normales Zeichen

Ein-/Ausgabe (E/A):

Es gibt keine C-Befehle für die Ein-/Ausgabe. Die Ein-/Ausgabe wird in C über Bibliotheksfunktionen zur Verfügung gestellt. Die wichtigsten E-/A-Funktionen sind printf und scanf.

Ausgeben mit printf:

printf ist eine sehr mächtige Ausgabefunktion, die als erstes Argument eine Zeichenkette mit dem auszugebenden Text erwartet. Dieser Text darf beliebig viele mit % beginnende Formatelemente enthalten. Dahinter kommen – der Reihe nach, durch jeweils ein Komma getrennt - die Werte/ Variablen zu den Formatelementen der Zeichenkette. Die passenden Formatelemente zu den einzelnen Datentypen können der Datentypentabelle entnommen werden.

Beispiel:

printf("Österreich hat %u Bundesländer und %u km² bei einer Einwohnerzahl von %u. Das ergibt eine Einwoherzahl/km² von %6.2f.", bundeslaenderanzahl, flaeche, einwohnerzahl,einwohnerzahl/ flaeche);

Interpretation der Variableninhalte durch printf:

Die Angabe des Formatelements sollte möglichst exakt zum Typ des auszugebenden Wertes passen, da ansonsten Fehlausgaben erfolgen. Negative Zahlen werden mit unsigned-Formatelementen positiv ausgegeben. Positive Zahlen außerhalb des Wertebereichs des verwendeten signed Formatelements (d) werden negativ ausgegeben.

%u, %o, %x und %X interpretieren das Bitmuster der Variablen als positive Ganzzahl (unsigned). Dies führt dazu, dass der Wert negativer Zahlen falsch ausgegeben werden (%d wäre hier das richtige Formatelement).

Zwischen dem %-Zeichen und der Typangabe kann eine Feldbreite gefolgt von einem Punkt und einer Zahl für die Genauigkeit angegeben werden. Die Feldbreite bestimmt die Anzahl der Zeichen, die für die Ausgabe des Wertes mindestens verwendet werden. Notfalls wird mit Leerzeichen aufgefüllt. Jedenfalls wird der auszugebende Wert nicht abgeschnitten. Die Genauigkeit bestimmt bei Gleitkommazahlen (%f, %lf, %g, %lg, %e, %le) die Anzahl der Nachkommastellen und bei Zeichenketten (%s) die Anzahl der maximal auszugebenden Zeichen. Die Genauigkeitsangabe bewirkt bei Gleitkommazahlen eine Rundung. Durch Voranstellen des Zeichens – kann die Ausrichtung der Ausgabe (linksbündig bei Zahlen, rechtsbündig bei Zeichenketten) umgedreht werden.

Einlesen mit scanf:

scanf erwartet wie printf als erstes Argument eine Zeichenkette mit den einzulesenden Zeichen inklusive Formatelementen für einzulesende Variablen. Die Reihenfolge und Datentypen der Variablen müssen den Formatelementen exakt entsprechen. Damit scanf die Variablen befüllen kann, müssen die Adressen der jeweiligen Variablen übergeben werden, was über den Adressoperator & erfolgen muss. Die Zeichen zwischen den Formatelementen müssen in der Eingabe exakt an der erwarteten Stelle vorkommen (siehe -, / und : im Beispiel). Die Zuordnung von Zeichen zu einer Variable endet beim ersten Zwischenraumzeichen (Leerzeichen, Tabulator bzw. Zeilenende) bzw. beim ersten Zeichen, das nicht zur Typangabe passt (jedes Zeichen außer

Ziffern und Vorzeichen bei Ganzzahlen, ungültiges Format einer Gleitkommazahl, ...). Das erste nicht zugeordnete Zeichen bleibt im Eingabepuffer für die weitere Eingabeverarbeitung. Das Entfernen von ungelesenen Zeichen kann durch fflush(stdin) erzwungen werden. %c liest als einziges Formatelement auch Zwischenraumzeichen. Wohingegen %s nur ein Wort bis zum ersten Zwischenraumzeichen (Leerzeichen, Tabulator, …) einliest und in der Variable speichert. %s kann beispielsweise nicht für das Einlesen von mehreren Vornamen in eine Zeichenkette genutzt werden. Bei scanf ist keine Angabe der Genauigkeit möglich. Eine Angabe der Feldbreite begrenzt die Anzahl der für eine Variable betrachteten Zeichen (siehe Beispiel für Datumseingabe).

Bsp.:

Einlesen von Datum und Uhrzeit im Format tt-mm-jjjj/hh:mm:ss (z.B. 30-11-2010/14:24:04).

scanf("%2u-%2u-%4u/%2u:%2u:%2u", &tag, &monat, &jahr, &stunde, &minute, &sekunde);

scanf erwartet sich die angeführten Trennzeichen -, : und / an der entsprechenden Position.

Zum Einlesen von Texten, die Leerzeichen/Tabulatoren/… enthalten kann eine an reguläre Ausdrücke angelehnte Syntax angewendet werden. Anstatt eines %s kann %[] mit den zulässigen Zeichen in den eckigen Klammern angegeben werden (z.B.: %[0123456789ABCDEFabcdef] für Hex-Ziffer). Dies bewirkt, dass scanf die entsprechende Zeichenkette mit den konformen Zeichen befüllt, bis ein ungültiges -also nicht in den Klammern enthaltenes – Zeichen auftritt. Durch ein ‚– , zwischen zwei Zeichen kann ein Bereich angegeben werden. Durch ein ‚^' am Beginn kann die Bedeutung der Zeichen auf unerlaubt umgestellt werden (z.B.: %[^.,;:!?] für ‚kein Satzzeichen').

Bsp.:

Einlesen einer Zeile einer csv-Datei mit Tabulator als Spaltentrenner (z.B. 10100 18 Burger Stefan M 20.08.1998 Steigenteschgasse 35 1220 Wien- Donaustadt 2AHCI).

Scanf("%u\t%hu\t%[^\t]\t%[^\t]\t%c\t%hu.%hu.%d\t%[^\t]\t%hu...", &id, &katalognummer, name, vorname, &geschlecht, &tag, ...);

Zeichenkette einlesen mit fgets:

Mit fgets(zielzeichenkette, zielzeichenkettenlaenge, stdin) kann eine Zeichenkette aus der Standardeingabe befüllt werden, wobei bis zum abschließenden Zeilenendezeichen '\n' gelesen wird. Es werden maximal Zielzeichenkettenlänge-1 Zeichen inklusive Zeilenendezeichen in die Zielzeichenkette übertragen und mit einem String-Endezeichen '\0' abgeschlossen. fgets übernimmt nur maximal zielzeichenkettenlaenge – 1 Zeichen, sodass ein Zeichenkettenüberlauf verhindert wird.

ANSI-C-Datentypen und Formatelemente (Wertebereich bezieht sich auf gnu C-Compiler von mingw):

Typ	Bytes	Wertebereich von	Wertebereich bis	Genauigkeit	Formatelement	Beispiel/Anmerkung

char	1	-128 bzw. CHAR_MIN oder 0 falls ohne Vorzeichen	127 bzw. CHAR_MAX bzw. UCHAR_MAX falls ohne Vorzeichen		%c	char zeichen='a'; // zeichen mit ASCII-Code initialisieren
						printf(“%c“, zeichen); // Buchstaben ausgeben
						scanf(“%c“, &zeichen); // Buchstaben einlesen

unsigned char	1	0	255 bzw. UCHAR_MAX		%c	char zeichen='a'; // zeichen mit ASCII-Code initialisieren
						printf(“%c“, zeichen); // Buchstaben ausgeben
						scanf(“%c“, &zeichen); // Buchstaben einlesen

signed char	1	-128 bzw. SCHAR_MIN	127 bzw. SCHAR_MAX		%c	char zeichen='a'; // zeichen mit ASCII-Code initialisieren
						printf(“%c“, zeichen); // Buchstaben ausgeben
						scanf(“%c“, &zeichen); // Buchstaben einlesen

short	2	-32768 bzw. SHRT_MIN	32767 bzw. SHRT_MAX		%hd, %hi	short minizahl=-23; // minizahl mit -23 initialisieren
						printf(“%hd“, minizahl); // minizahl dezimal ausgeben
						scanf(“%hd“, &minizahl); // minizahl dezimal einlesen

unsigned short	2	0	65535 bzw. USHRT_MAX		%hu, %ho, %hx, %hX	unsigned short minizahl2=12; // minizahl2 mit 12 initialisieren
						printf(“%hu“, minizahl2); // minizahl2 dezimal ausgeben
						scanf(“%hu“, &minizahl2); // minizahl2 dezimal einlesen
						// bei %ho erfolgt Ein-/Ausgabe oktal
						// bei %hx bzw. %hX erfolgt Ein-/Ausgabe hexadezimal
						// scanf liest bei %hu auch Eingaben mit vorangestelltem -

int	4	-2147483648 bzw. INT_MIN	2147483647 bzw. INT_MAX		%i, %d	int zahl=-12345; // zahl mit -12345 initialisieren
						printf(“%d“, zahl); // zahl dezimal ausgeben
						scanf(“%d“, &zahl); // zahl dezimal einlesen

unsigned int	4	0	4294967295 bzw. UINT_MAX		%u, %o, %x, %X	unsigned int zahl2=56789; // zahl2 mit 56789 initialisieren
						printf(“%u“, zahl2); // zahl2 dezimal ausgeben
						scanf(“%u“, &zahl2); // zahl2 dezimal einlesen
						// bei %o erfolgt Ein-/Ausgabe oktal
						// bei %x bzw. %X erfolgt Ein-/Ausgabe hexadezimal

long	4	-2147483648 bzw. LONG_MIN	2147483647 bzw. LONG_MAX		%ld	long maxizahl=-456; // maxizahl mit -456 initialisieren
						printf(“%ld“, maxizahl); // maxizahl dezimal ausgeben
						scanf(“%ld“, &maxizahl); // maxizahl dezimal einlesen

unsigned long	4	0	4294967295 bzw. ULONG_MAX		%lu, %lo, %lx, %lX	unsigned long maxizahl2=56789; // maxizahl2 mit 56789 initialisieren
						printf(“%lu“, maxizahl2); // maxizahl2 dezimal ausgeben
						scanf(“%lu“, &maxizahl2); // maxizahl2 dezimal einlesen
						// bei %lo erfolgt Ein-/Ausgabe oktal
						// bei %lx bzw. %lX erfolgt Ein-/Ausgabe hexadezimal

long long	8	-9223372036854775808 bzw. LLONG_MIN	9223372036854775807 bzw. LLONG_MAX		%lld	long long maxizahl=-456; // maxizahl mit -456 initialisieren
						printf(“%lld“, maxizahl); // maxizahl dezimal ausgeben
						scanf(“%lld“, &maxizahl); // maxizahl dezimal einlesen

unsigned long long	8	0	18446744073709551615
			
			bzw. ULLONG_MAX
					%llu, %llo, %llx, %llX	unsigned long long maxizahl2=56789; // maxizahl2 mit 56789 initialisieren
						printf(“%llu“, maxizahl2); // maxizahl2 dezimal ausgeben
						scanf(“%llu“, &maxizahl2); // maxizahl2 dezimal einlesen
						// bei %llo erfolgt Ein-/Ausgabe oktal
						// bei %llx bzw. %llX erfolgt Ein-/Ausgabe hexadezimal

float	4	1.17E-38	3.4E38	6 Stellen	%f, %e, %g	float gleitzahl=-12345.67; // gleitzahl mit -12345.67 initialisieren
						printf(“%f“, gleitzahl); // gleitzahl ausgeben
						scanf(“%f“, &gleitzahl); // gleitzahl einlesen
						// bei %e wird wissenschaftliche Notation -12.34567E+3,... verwendet
						// bei %g wird die jeweils kürzere (%f oder %e) verwendet

double	8	2.22E-308	1.8E308	15 Stellen	%lf, %le, %lg	double gleitzahl2=-1.234567E-4; // gleitzahl mit -0.00001234567 initialisieren
						printf(“%lf“, gleitzahl); // gleitzahl ausgeben
						scanf(“%lf“, &gleitzahl); // gleitzahl einlesen
						// bei %le wird wissenschaftliche Notation -1.234567E-4,... verwendet
						// bei %lg wird die jeweils kürzere (%lf oder %le) verwendet

long double	8	3.36e-4932	1.18973e+4932	18 Stellen	%Lf, %Le, %Lg	long double gleitzahl2=-1.234567E-4; // gleitzahl mit -0.00001234567 initialisieren
						printf(“%Lf“, gleitzahl); // gleitzahl ausgeben
						scanf(“%Lf“, &gleitzahl); // gleitzahl einlesen
						// bei %Le wird wissenschaftliche Notation -1.234567E-4,... verwendet
						// bei %Lg wird die jeweils kürzere (%Lf oder %Le) verwendet

char[]					%s	Char text[100] = “Hallo Welt“; // Zeichenkette mit Platz für 99 nutzbare Zeichen anlegen und mit Hallo Welt initialisieren
						printf(“%s“, text);
						fgets(text, maxZeichenAnzahl); // scanf(“%s“, text) liest nur bis 1. Trennzeichen, fgets ersetzt abschließendes '\n' durch '\0'

Das Formatelemente x bzw. X erzeugen kein Präfix 0x für die ausgegebene Hex-Zahl, sondern geben nur die Hex-Ziffern der Zahl aus. Zur Erzeugung dieses Präfix muss 0x bzw. 0X vor das

Formatelement gestellt werden.

Anmerkungen zur Thematik mit/ohne Vorzeichen:

Beim Datentyp char (ohne unsigned/signed Angabe) legt der ANSI-Standard nicht fest, ob er mit (signed) oder ohne (unsigned) Vorzeichen realisiert werden muss. Der Wertebereich entspricht entweder dem von unsigned char oder signed char.
short, int und long sind ohne dem Zusatz unsigned mit Vorzeichen, können durch den unsigned Zusatz aber ohne Vorzeichen behandelt werden.
Float, double und long double sind immer mit Vorzeichen, sie können nicht ohne Vorzeichen betrachtet werden.
Die Byteanzahl der einzelnen Datentypen bzw. Variablen kann mit dem Operator sizeof() ermittelt werden (Bsp. unsigned int laenge = sizeof(long); bzw. unsigned int laenge = sizeof(laenge);). Die Grenzen der Wertebereiche (INT_MAX, UINT_MIN,...) können aus denHeader-Dateien limits.h für Ganzzahlen und float.h für Gleitkommazahlen gewonnen werden.

Allgemeines zur Byteanzahl laut ANSI-Standard:

char < short <= int <= long <= long long (short kann also höchstens so lang wie int, long oder long long sein, int ist mindestens so lang wie short und höchstens so lang wie long oder long long); die oben angeführten Byteanzahlen gelten für die von der HTL Wels für SEW/AINF/AIIT/... zur Verfügung gestellten Systeme (Windows, gnu-C in mingw 32Bit).
Durch Verwendung eines falschen Formatelements im printf-Aufruf wird der Inhalt einer Variable falsch interpretiert. D.h. es wird zum Beispiel im Fall eines unsigned int die darin höchste mögliche Zahl (INT_MAX) bei %d als -1 ausgegeben.

Operatoren

Operatoren sind Symbole, die Ausdrücke zu neuen Ausdrücken verbinden. In C werden sämtliche Wertänderungen, Berechnungen, Transformationen,... durch Ausdrücke bewirkt. Ein Ausdruck in C entspricht einem Term in der Mathematik. Variablen, Konstante, Literale und Funktionsaufrufe sind elementare Ausdrücke, die keine Operatoren zu ihrer Erzeugung benötigen. Nach Anzahl der durch einen Operator verbundenen Operanden werden Operatoren in unär (ein Operand, z.B. Vorzeichen), binär (zwei Operanden, z.B. Multiplikationsoperator) und trinär (drei Operanden, z.B. Auswahloperator) eingeteilt. Ausdrücke haben einen Wert (Variableninhalt, Berechnungsergebnis, …) und den Typ des berechneten Wertes (hängt von den beteiligten Operanden ab, siehe Typerweiterung).

Arithmetische Operatoren:

Unär:

- (Vorzeichen) ändert das Vorzeichen der Variablen (vgl. 2-er Komplement), ++, – (Inkrement/ Dekrement, a++ entspricht a = a + 1, verändert den Wert der Variablen um 1 bzw. um die Größe eines Elements bei Pointerarithmetik), von rechts nach links zusammengefasst, steht ++ bzw. -- vor der Variablen (Präfix), dann wird als erstes der Variableninhalt verändert und als zweites die Variable ausgelesen, steht ++ bzw. -- nach der Variablen (Postfix), dann wird als erstes die Variable ausgelesen und als zweites der Variableninhalt verändert, Inkrement-/Dekrementoperatoren verursachen sogenannte Seiteneffekte und sollten deshalb bei Funktionsaufrufen und Makroaufrufen vermieden werden. Standardmäßig werden Argumente von rechts nach links ausgewertet und an die aufgerufene Funktion als Kopie übergeben. Argumente mit Inkrement/ Dekrement werden somit mit verschiedenen Werten übergeben (Bsp.: printf(“%d-%d-%d“, a++, a+ +, a++)).

Binär:

+, -, *, /, % (Rest der Division, nur für ganzzahlige Operanden), von links nach rechts zusammengefasst, Punktrechnung (*, /, %) vor Strichrechnung (+, -), sobald ein Operand eine Gleitkommazahl ist, wird in Gleitkomma gerechnet.

Zuweisungsoperatoren:

Für die Zuweisung eines neuen Wertes an eine Variable steht in C ein einfacher (=)und mehrere zusammengesetzte Zuweisungsoperatoren (+=, -=, *=, /= und %=) zur Verfügung. Auf der linken Seite einer Zuweisung steht die Variable, deren Wert überschrieben werden soll. Rechts steht ein beliebig komplexer Ausdruck, dessen Wert vor der Zuweisung ermittelt wird. Die Zuweisungsoperatoren haben den zweit niedrigsten Rang aller Operatoren. Zuweisungen werden von rechts nach links ausgeführt, d.h. es wird zuerst die ganz rechts stehende Zuweisung ausgeführt, der zugewiesene Wert wird der Reihe nach jeder Variablen links davon zugewiesen. Eine Zuweisung ist ein Ausdruck und hat damit einen Wert (der zugewiesene Wert) und einen Typ (Typ des zugewiesenen Wertes).

Die kombinierte Zuweisung wird so ausgeführt, dass zuerst der rechts vom Zuweisungsoperator stehende Ausdruck berechnet wird. Dieser Zwischenwert wird mit dem Operator der kombinierten Zuweisung verknüpft und zum Schluss der links stehenden Variablen zugewiesen. Die kombinierten Zuweisungsoperatoren haben wie Inkrement/Dekrement den Seiteneffekt der Wertänderung der

zugeordneten Variablen.

Bsp.: a *= b + c; // entspricht: a = a * (b + c)

Vergleichsoperatoren:

Vergleiche von Werten zweier Ausdrücke können mit Vergleichsoperatoren ermittelt werden. Als Ergebnis wird eine Zahl vom Typ int geliefert, wobei 0 für Vergleich ist nicht erfüllt und 1 für Vergleich ist erfüllt steht. Dies passt sehr gut zur Logik von C, dass der Wert 0 als falsch und ein Wert ungleich 0 als wahr interpretiert wird. Für Größenvergleiche stehen die vier Vergleichsoperatoren <, <=, > und >= zur Verfügung. Gleichheit kann mit == und Ungleichheit mit != überprüft werden. == und != haben einen geringeren Rang als <, <=, > und >=.

ACHTUNG:

Vergleichsoperator == und Zuweisungsoperator = werden leicht verwechselt. Der Compiler kann eine irrtümliche Verwendung von = nicht erkennen, da eine Zuweisung einen Wert hat und dieser Wert als wahr oder falsch interpretiert wird. Durch Platzierung der Variablen auf der rechten Seite des Vergleichs können fehlerhafte Verwendungen durch den Compiler besser erkannt werden.

Bsp.: if (a = getchar())// hier liegt ein Fehler vor, falls der Programmierer das Ergebnis von getchar auf Übereinstimmung mit a prüfen will

if (getchar() == a) // für Prüfung auf Gleichheit besser so formulieren, damit bei nur einem =-Zeichen ein Syntaxfehler vorliegt

Logische Operatoren:

Zur logischen Verknüpfung von Ausdrücken (Vergleiche, Wahrheitswerte,...) stehen die drei Operatoren && (UND), || (ODER) und ! (NICHT) zur Verfügung. || hat den geringsten Rang, && einen höheren Rang und ! den höchsten Rang unter den logischen Operatoren. Die Auswertung eines logischen Ausdrucks erfolgt von links nach rechts. Sie endet an der Stell, ab der keine Änderung des vorliegenden Zwischenergebnisses nicht mehr möglich ist. Die senkrechte Linie für den ODER-Operator wird durch Drücken und Festhalten der rechten AltGr-Taste und Drücken der Taste <> erzeugt.

Bsp: index >= 0 && index < anzahl // wenn index >= 0 falsch ist, dann endet die Auswertung vor dem Vergleich mit anzahl, ansonsten nicht

index < 0 || index >= anzahl // wenn index < 0 wahr ist, dann endet die Auswertung vor dem Vergleich mit anzahl, ansonsten nicht

Rang der Operatoren:

Für die korrekte Anwendung der einzelnen Operatoren ist das Wissen über den jeweiligen Rang unumgänglich. Zusätzlich muss die Auswertungsrichtung (li → re oder re → li) berücksichtigt werden. Die folgende Tabelle soll die bisher besprochenen Operatoren mit ihrem jeweiligen Rang darstellen. Innerhalb eines Rangs entscheidet die Spalte Zusammenfassung/Richtung über die Zuordnung und Priorität.

Rang	Operatoren	Zusammenfassung/Richtung (von links bei fehlender Angabe)

1	( )	
2	! ++-- -(Vorzeichen) &(Adressoperator) (Datentyp) (=cast-Operator)	von rechts
3	* / %	
4	+ -	
5		
6	< <= > >=	
7	== !=	
8		
9		
10		
11	&&	
12	||	
13		
14	= += -= *= /= %=	von rechts
15		

Kontrollstrukturen (Verzweigungen, Schleifen)

if-Verzweigung:

Damit Programme bei unterschiedlichen Werten von Variablen verschiedene Anweisungen ausführen können, muss eine Verzweigungsanweisung mit entsprechender Bedingung im Programmquelltext notiert werden.

In C heißt das Schlüsselwort hierfür if und die Bedingung wird hinter dem if in runden Klammern angegeben. In C wird ein Ausdruck mit dem Wert 0 als falsch, jeder von 0 verschiedene Wert als wahr interpretiert. Hinter der if-Anweisung steht eine Anweisung, die bei erfüllter Bedingung ausgeführt wird. Sollen mehrere Anweisungen bei wahrer Bedingung ausgeführt werden, dann müssen diese durch geschwungene Klammern zu einem Block zusammengefasst werden. Die vom if kontrollierte Anweisung(sfolge) wird nach rechts eingerückt. Dies hat jedoch auf die Programmlogik keinen Einfluss.

Über das Schlüsselwort else kann eine weitere Anweisung bzw. ein Block angegeben werden, der bei Nichterfüllung der Bedingung ausgeführt werden soll.

Bsp.:

```
if (geburtsjahr < 1993)
        printf(“Person ist erwachsen.\n“);
else {
        kartenpreis = 13;
        printf(“Person ist erwachsen.\n“);
}
```

Nach der vom if kontrollierten Anweisung(sfolge) kann eine beliebige Anzahl von else if (Bedingung) notiert werden. Jede dieser else if Anweisungen führt bei Erfüllung der Bedingung die danach formulierte Anweisung(sfolge) aus. Die kontrollierten Anweisungen werden alle gleich weit eingerückt.

Bsp.:

```
if (geburtsjahr < 1993)
        printf(“Person ist erwachsen.\n“);
else if (geburtsjahr ) {
        kartenpreis = 13;
        printf(“Person ist erwachsen.\n“);
}
```

switch-Verzweigung:

Mit dem Schlüsselwort switch kann der Wert eines ganzzahligen Ausdrucks der Reihe nach (von oben nach unten) mit mehreren konstanten Werten (case-Marke) auf Gleichheit überprüft werden. Bei Ungleichheit wird mit dem nächsten Vergleich fortgesetzt. Das geht so lange, bis eine Übereinstimmung vorliegt oder die switch-Anweisung zu Ende ist. Bei Übereinstimmung wird das

Programm ab dieser Stelle ohne weitere Vergleiche mit den noch folgenden case-Marken forgesetzt. Dieses Verhalten kann durch eine break-Anweisung durchbrochen werden, sodass der Rest der switch-Anweisung übersprungen wird. Das Schlüsselwort default steht für 'stimmt mit jedem Wert überein'. Bei Erreichen der case-Marke default werden die Anweisungen dahinter ohne weitere Vergleiche ausgeführt. In sehr vielen Fällen steht am Ende des Programmcodes einer case-Marke eine break-Anweisung, um die Verzweigung zu verlassen. Bei identem Code für verschiedene Werte können jedoch mehrere case-Marken unmittelbar aneinander gereiht werden. In seltenen Fällen kann es auch vorkommen, dass zwischen case-Marken Code steht, der eine zu diesen case-Marken passende Verarbeitung vornimmt. Eine break-Anweisung folgt unter Umständen erst bei einer der folgenden case-Marken.

Bsp.:

```
switch (buchstabe) {
case 'a':
case 'e':
case 'i':
case 'o':
case 'u':
        printf(“%c ist ein Vokal“, buchstabe);
        break;
case '.':
case ',':
case ':':
case ';':
case '!':
case '?':
        printf(“%c ist ein Satzzeichen“, buchstabe);
        break;
default:
        if (buchstabe >= 'a' && buchstabe <= 'z' || buchstabe >= 'A' && buchstabe <= 'Z')
                printf(“%c ist ein Mitlaut“, buchstabe);
        else
                printf(“%c ist ein sonstiges Zeichen“, buchstabe);
}
```

Schleifen allgemein

Als nächstes wollen wir uns mit der wiederholten Ausführung von Anweisungen, den sogenannten Schleifen, näher auseinandersetzen. Schleifen haben einen steuernden Teil (Schleifenkopf/-fuß) und

einem gesteuerten Teil (Schleifenrumpf). Die Bedingung, ob der Schleifenrumpf ausgeführt werden soll, kann entweder vor jedem Schleifendurchlauf geprüft werden (Kopf gesteuerte Schleife, 0 Durchläufe möglich, while, for). Die Schleifenbedingung kann jedoch auch nach jedem Schleifendurchlauf geprüft werden (Fuß gesteuerte Schleife, mindestens 1 Durchlauf, do while). In C besteht der Schleifenrumpf aus genau einer Anweisung. Um mehrere Anweisungen durch eine Schleifenbedingung kontrollieren zu können, müssen diese Anweisungen durch geschwungene Klammern zu einem Block zusammengefasst werden. Für C-Einsteiger empfiehlt sich die generelle Blockbildung, um Problemen mit nachträglich zum Schleifenrumpf hinzugefügten Anweisungen zu vermeiden.

Schleifen können in C durch die break-Anweisung verlassen werden, sodass mit der ersten Anweisung nach der Schleife fortgesetzt wird. Durch die Anweisung continue kann die aktuelle Ausführung des Schleifenrumpfes beendet werden und mit der Überprüfung der Schleifenbedingung fortgesetzt werden.

Ein Strichpunkt nach dem Schleifenkopf der while- bzw. for-Schleife führt zu einem leeren Schleifenrumpf, da der Strichpunkt eine Anweisung darstellt.

while-Schleife:

Die while-Schleife ist eine Kopf gesteuerte Schleife. Die Schleifenbedingung (in runde Klammern eingeschlossen) wird vor jedem Durchlauf geprüft. Der Schleifenrumpf wird nur dann durchlaufen, wenn die Bedingung erfüllt ist. Die while-Schleife wird verlassen, wenn die Schleifenbedingung falsch (Wert 0) ergibt.

Bsp.:

```
int index = 0;
while (index < anzahl) {
        summe += index;
        index++;
}
```

do-while Schleife:

Bei der do-while-Schleife wird der Schleifenrumpf zwischen die beiden Schlüsselwörter do und while eingeschlossen. Dennoch darf hier nur eine Anweisung formuliert werden, für mehrere Anweisung ist wiederum Blockbildung erforderlich. Die do-while-Schleife wird mindestens einmal durchlaufen und die Bedingung wird nach jedem Schleifendurchlauf geprüft. Bei Erfüllung der Bedingung wird der Schleifenrumpf erneut ausgeführt.

Bsp.:

```
do {
        printf(“Zahl eingeben (Ende bei 0): “);
        scanf(“%d“, &zahl);
} while (zahl != 0);
```

for-Schleife:

Viele Schleifen haben folgenden Ablauf: Zuerst wird die Ausgangssituation für die Schleifendurchläufe hergestellt (Initialisierung). Dann wird die Schleifenbedingung geprüft. Als nächstes wird der Schleifenrumpf ausgeführt. In diesem wird zumeist als letzte Anweisung die Ausgangssituation für den nächsten Durchlauf geschaffen (Reinitialisierung). In der for-Schleife besteht im Schleifenkopf die Möglichkeit, die Initialisierung, die Schleifenbedingung und die Reinitialisierung festzulegen. Jeder dieser drei Bereiche kann unabhängig von den anderen leer gelassen werden. Eine leere Schleifenbedingung wird als wahr angenommen. Der Schleifenrumpf wird jeweils zwischen Bedingung prüfen und reinitialisieren ausgeführt. Nach dem Reinitialisieren wird die Schleifenbedingung geprüft.

Schema: for (initialisieren; bedingung prüfen; reinitialisieren)

Bsp.:

```
int index;
for (index = 0; index < anzahl; index++) {
        summe += index;
}
```

break/continue:

break dient zum Verlassen einer switch-Anweisung bzw. einer Schleife. Es wird jeweils die innerste Kontrollstruktur verlassen und hinter dieser fortgesetzt.

continue dient zum Überspringen des Rests des Schleifenrumpfes. Bei while- und do-while-Schleife wird mit der Überprüfung der Schleifenbedingung fortgesetzt. Bei der for-Schleife wird mit der Reinitialisierung fortgesetzt.

Der Präprozessor

Vor der eigentlichen Übersetzung des C-Quellcodes in Maschinencode ruft der C-Compiler den Präprozessor auf. Dieser interpretiert die an ihn gerichteten Befehle (Präprozessor-Direktiven) und erzeugt eine neue Eingabedatei für den folgenden Compilerlauf.

Präprozessor-Direktiven müssen in einer Quellcode-Zeile alleine stehen, beginnen mit einem #-Symbol und werden nicht mit einem Strichpunkt abgeschlossen. Es gibt Präprozessor-Direktiven zur Einbindung einer Dateien in die aktuell verarbeitete Datei, zum Definieren einer symbolischen Konstanten (Name für ein Literal) bzw. eines Makros (Name für eine Anweisungsfolge). Sehr große Bedeutung haben die Präprozessor-Direktiven zur bedingten Kompilierung, da hierdurch viele Aspekte der Plattformunabhängigkeit bewältigt werden können.

#include <datei.h>

#include ersetzt die eigene Zeile durch den Inhalt der angegebenen Datei. Die in der eingefügten Datei enthaltenen #include-Direktiven werden ebenfalls durch die einzufügende Datei ersetzt.

#define NAME ersatztext

#define erzeugt eine symbolische Konstante als Paar aus name und ersatztext und ersetzt ab der eigenen Zeile jedes Vorkommen von name (ausgenommen in Zeichenkettenliterale) durch ersatztext. Bei mehrzeiligem Ersatztext muss jede Zeile mit Ausnahme der letzten Zeile mit einem Backslash abgeschlossen werden. In der C-Entwicklergemeinde herrscht die Konvention, dass symbolische Konstante keine Kleinbuchstaben enthalten. Symbolische Konstante verbessern die Verständlichkeit des Quellcodes und vereinfachen die Wartung durch selbsterklärende Namen. Bei Änderung des Wertes einer symbolischen Konstanten müssen alle davon abhängigen Quellcode-Dateien erneut übersetzt werden.

Die #define-Direktive ermöglicht die Definition von Makros. Ein Makro ist einer Funktion ziemlich ähnlich mit dem Unterschied, dass der Makroaufruf zur Übersetzungszeit durch die als ersatztext formulierte Anweisungsfolge ersetzt wird.

Bsp.:

```
#define QUADRAT(a)          ((a)*(a))
```

Der Ersatztext und die Makroparameter in ersatztext müssen zur Sicherstellung der Korrektheit immer geklammert werden, da Ausdrücke mit Operatoren geringeren Rangs (z.B. 3 * c) als Argument erlaubt sind. Zwischen dem Makronamen und den Makroparametern dürfen keine Trennzeichen gesetzt werden.

Vergleich Makro und Funktion:

Ein Makro muss für den Compiler zur Übersetzungszeit sichtbar sein. Der Präprozessor setzt für jeden Makroaufruf den kompletten Ersatztext ein, wofür der Compiler wiederholt den selben Maschinencode erzeugt. Das resultierende Programm wird größer, enthält jedoch keine Funktionsaufrufe für die einzelnen Makroaufrufe. Seiteneffekte in den Makroaufrufen sowie im Ersatztext führt zu Problemen (Bsp.: QUADRAT(a++)).

Bei einer Funktion muss zur Übersetzungszeit nur die Deklaration der Funktion (Prototyp) für den Compiler sichtbar sein. Die Definition der Funktion muss jedoch für den Linker sichtbar sein, es sei

denn, dass sich diese Funktion in einer dynamischen Bibliothek (.DLL bzw. .so) befindet. Der Compiler erzeugt für jede Funktionsdefinition den entsprechenden Maschinencode, sodass das Programm keinen doppelten Code hierfür enthält. Zur Laufzeit muss dieser Maschinencode jedoch über entsprechende Maschinencode-Anweisungen aufgerufen werden. Argumente mit Seiteneffekten sind weniger problematisch (pow(c++), da ausgewertetes Argument an die Funktion als Kopie übergeben wird.

Die vom Präprozessor erzeugte temporäre Zwischendatei <eingabedateiname>.i kann z.B. beim gcc durch die Option -save-temps vor dem Löschen bewahrt werden, sodass sie für eine Untesuchung des resultierenden Quellcodes zur Verfügung steht. Es kann somit untersucht werden, ob die korrekten Definitionen bei bedingter Kompilierung erzeugt wurden.

Datentypumwandlung

Werden in einem Ausdruck Operanden mit verschiedenen Datentypen verknüpft, dann erzeugt der Compiler Maschinencode zur impliziten (nicht ausdrücklich) Angleichung der Datentypen. Dabei werden Operanden mit einem kürzeren/kleineren Datentyp auf einen längeren/größeren Datentyp erweitert. Der ermittelte gemeinsame Datentyp ergibt den Ergebnistyp.

Die Hierarchie der arithmetischen Datentypen ab dem int ist wie folgt:

int → unsigned int → long → unsigned long → long long → unsigned long long → float → double → long double (unsigned int und long entfallen, falls int == long, long long und long double wurden mit ANSI C99 eingeführt)

Arithmetische Typumwandlungen:

Die im folgenden beschriebenen Typumwandlungen gelten für alle Ausdrücke außer der Zuweisung und den logischen Operationen && bzw. ||. Die Typumwandlungen für die Zuweisung werden weiter unten beschrieben.

Ganzzahlerweiterung:

Bei ganzzahligen Datentypen erfolgen folgende Angleichungen/Erweiterungen:

char → unsigned char → short → int

unsigned short → int (falls int == long) bzw. unsigned short → unsigned int (falls int == short)

Ganzzahlige Operanden werden immer mindestens als int verarbeitet. Wird eine negative Zahl als unsigned interpretiert, dann ändert sich der Wert der Zahl (und zwar auf Maximalwert des unsigned Typs abzüglich Betrag der negativen Zahl).

Weitere implizite Typanpassungen:

Treten nach der Ganzzahlerweiterung noch Operanden verschiedenen Typs auf, dann wird entsprechend der oben angeführten Hierarchie der arithmetischen Datentypen so lange erweitert, bis Typgleichheit erreicht ist.

Die üblichen arithmetischen Typumwandlungen erhalten den Wert der Zahl, soweit dieser mit dem neuen Typ darstellbar ist.

1. Unsigned Typ in einen größeren ganzzahligen Typ: Durch Null-Erweiterung (an den zusätzlichen höheren Bits) wird das Bitmuster links durch Auffüllen mit Nullen auf die Länge des neuen Typs gebracht, der Wert der Zahl bleibt dabei erhalten.
2. Signed Typ in einen größeren ganzzahligen Typ:
   - Neuer Typ signed: Vorzeichenerweiterung zur Erhaltung des Wertes indem links mit dem Vorzeichen-Bit (höchstes Bit des Ausgangswertes) aufgefüllt wird.
   - Neuer Typ unsigned: Der negative Wert bleibt nicht erhalten. Bei gleicher Länge des alten und neuen Typs bleibt das Bitmuster erhalten (Wert ändert sich auf Maximalwert des neuen Typs abzüglich Betrag der negativen Zahl), es ändert sich die Interpretation. Bei größerem neuen Typ wird zuerst die Vorzeichenerweiterung durchgeführt und das entstandene Bitmuster unsigned interpretiert.

3. Ganzahliger Datentyp in Gleitpunkttyp: Hier erfolgt eine Umrechung in die exponentielle Form der Gleitpunktdarstellung, wobei der Wert erhalten bleibt, soweit dies von der Genauigkeit möglich ist (float hat weniger Stellen Genauigkeit als z.B. eine 32-Bit-Zahl). Reicht die Genauigkeit des neuen Typs nicht, dann erfolgt eine Rundung gegen 0.

Implizite Typumwandlungen bei Zuweisung:

Bei unterschiedlichen Datentypen links und rechts der Zuweisung erzeugt der Compiler Maschinencode zur Typanpassung. Bei einer zusammengesetzten Zuweisung wird zuerst der Ausdruck rechts vom Zuweisungsoperator auf einen gemeinsamen Typ gebracht und dann erfolgt die Typanpassung für die Zuweisung.

Steht der Typ links vom Zuweisungsoperator in der Hierarchie höher als der Ergebnistyp des rechts stehenden Ausdrucks, dann erfolgt die oben beschriebene Typerweiterung. Im umgekehrten Fall muss eine Typumwandlung erfolgen.

1. Umwandlung eines ganzzahligen Typs in einen kleineren Typ: Die Umwandlung in einen Datentyp mit weniger Bytes erfolgt durch Abschneiden der/des höheren Bytes. Der Wert bleibt nur dann erhalten, wenn er nicht zu groß ist für den neuen Typ. Ein unsigned Typ wird in einen signed Typ gleicher Größe konvertiert, indem das unveränderte Bitmuster mit Vorzeichen interpretiert wird (Werte über dem Maximalwert des neuen Typs werden negativ interpretiert).
2. Umwandlung eines Gleitpunkttyps in einen ganzzahligen Typ: Der gebrochene Teil der Gleitpunktzahl wird abgeschnitten. Ist die so erhaltene Zahl zu klein/groß für den neuen Datentyp, so ist das Ergebnis undefiniert.
3. Umwandlung eines Gleitpunkttyps in einen kleineren Gleitpunkttyp: Ein Wert innerhalb des Wertebereichs des neuen Datentyps bleibt bei eventuell geringerer Genauigkeit erhalten. Bei einem zu großen Wert ist das Ergebnis undefiniert.

Typumwandlung bei einem Funktionsaufruf

Die Argumente eines Funktionsaufrufs werden wie bei der Zuweisung implizit arithmetisch angepasst, soweit für die aufgerufene Funktion ein Prototyp angegeben ist. Fehlt der Prototyp, dann wird die Ganzzahlerweiterung durchgeführt bzw. float zu double erweitert. Wegen dieser Gleitkommaerweiterung sind %f bzw. %lf bei printf gleichwertig.

Explizite Typanpassung - cast-Operator

Durch Voranstellen des gewünschten Typs in runden Klammern kann die Anpassung des Typs eines Ausdrucks erzwungen werden. Der cast-Operator hat wie die unären Operatoren (Vorzeichen, Inkrement/Dekrement, Address of,...) den zweithöchsten Rang.

Vektoren/Felder und Strings (Zeichenketten)

Ein Vektor/Feld (engl. Array) dient zur Speicherung mehrerer Elemente des gleichen Datentyps. In diesem Dokument werden Feld und Vektor synonym verwendet. Mit der Definition eines Vektors/ Feldes wird der Variablenname, der Typ der Elemente und deren Anzahl festgelegt (Syntax: typ name[anzahl]). Die Anzahl der Elemente muss ein konstanter Ausdruck sein (nur Literale, Konstante), d.h. der Wert muss zur Übersetzungszeit bekannt sein. Ein Vektor/Feld wird als zusammenhängender Speicherbereich abgelegt. Jeder bekannte Datentyp darf als Typ der Feldelemente verwendet werden.

Bsp.:

```
#define ZK_LEN 101

char zeichenkette[ZK_LEN] = “HTL-Wels“; // 100 Nutzzeichen + '\0'
```

Der Zugriff auf ein Vektorelement erfolgt über den Index des gewünschten Elements. Der Index beginnt bei 0 und endet bei Vektorlänge – 1. Jeder ganzzahlige Ausdruck darf als Index verwendet werden. Das C-Laufzeitsystem prüft den Indexwert nicht auf Einhaltung des zulässigen Bereichs. Hierfür ist ganz allein der Programmierer verantwortlich (Schleifenbedingung, if-Bedingung,...).

Bsp.:

```
for (i = 0; i < ZK_LEN; i++)
        printf(“%c“, zeichenkette[i]);
```

Vektoren können bei der Definition initialisiert werden, indem die Werte der einzelnen Vektorelemente in geschwungenen Klammern durch Komma getrennt angegeben werden. Wird ein Vektor initialisiert, dann darf die Länge des Vektors weggelassen werden. Die Länge des Vektors ergibt sich aus der Anzahl der Initialisierungswerte. Ist die Länge eines Vektors trotz Initialisierung angegeben, so werden die verbleibenden Vektorelemente mit 0 initialisiert. Zu viele Initialisierungswerte werden hingegen ignoriert.

Lokal (d.h. Innerhalb einer Funktion) definierte Vektoren werden zur Laufzeit am Stack angelegt und durch erzeugte Maschinenanweisungen initialisiert. Größere Vektoren (über 1 KB) sollten global bzw. static definiert werden, da sie dann den Stack nicht belasten und außerdem zur Übersetzungszeit initialisiert werden können.

Bsp.:

```
double polynomkoeffizient[10] = {1.1, 22.22, 33.33, 444.444, 5555.5555};
```

Strings / Zeichenketten

Zeichenketten in C sind Zeichenfelder mit einem abschließenden String-Endezeichen '\0'. Der Zeichenvektor muss deshalb um 1 länger sein als die zu speichernde Buchstabenanzahl. Eine Zeichenkette kann durch ein String-Literal initialisiert werden. Die Initialisierung mit den einzelnen Buchstaben ist hierzu äquivalent, wobei dann das abschließende String-Endezeichen mit angegeben werden muss (Bsp.: char zeichenkette[ZK_LEN] = {'H', 'T', 'L', '-', 'W', 'e', 'l', 's', '\0'};).

Vektorname und Adresse des Vektors

Eine Vektorvariable steht eigentlich für die Adresse des ersten Elements des Vektors. D.h. es gilt die Bedingung polynomkoeffizient == &polynomkoeffizient[0]. Vektorvariablen darf kein Wert

zugewiesen werden, da die Adresse eines Vektors konstant ist und nicht überschrieben werden darf. Eine Anweisung der Form vektorvariable = … ist nicht zulässig. Es dürfen nur die einzelnen Vektorelemente verändert/beschrieben werden.

Die Tatsache, dass eine Vektorvariable für die Adresse des ersten Elements steht erklärt auch, warum beim Formatelement %s in scanf kein Adressoperator erforderlich ist.

Adressarithmetik

In C kann mit Adressen gerechnet werden, d.h. es kann eine Ganzzahl zu einer Adresse addiert oder von ihr subtrahiert werden. Im Fall eines Vektors ergibt sich daraus der Zusammenhang, dass vektorvariable + 4 die Adresse des Elements mit dem Index 4 (&vektorvariable[4]) darstellt. Über den Zugriffsoperator * kann hierdurch auf jedes einzelne Vektorelement zugegriffen werden (*(vektorvariable + 4) = ...). Diese Art des Vektorzugriffs verschlechtert jedoch die Lesbarkeit eines Programms, sodass wenn möglich der Indexzugriff eingesetzt werden soll.

Mehrdimensionale Vektoren

In C können Vektoren von Vektoren von Vektoren von … definiert und verarbeitet werden. Bei der Definition wird für jede Dimension ein Paar eckiger Klammern mit der entsprechenden Länge angegeben. Eine zweidimensionale Tabelle aus Zeilen und Spalten kann in C wie folgt definiert werden:

#define ZEILEN_ANZAHL 12

#define SPALTEN_ANZAHL 4

double tabelle[ZEILEN_ANZAHL][SPALTEN_ANZAHL]

Jedes Element tabelle[i] ist ein double-Vektor der Länge SPALTEN_ANZAHL.

Bei der Initialisierung eines mehrdimensionalen Vektors müssen die Werte wie bei einem eindimensionalen Vektor in geschwungene Klammern geschrieben werden. Die Längenanzahl für die erste Dimension (und nur diese) darf bei Initialisierung weggelassen werden, da sie sich aus der Anzahl der Initialisierungswerte ergibt. Die teilweise Initialisierung von Vektorelementen führt (zumindest beim GNU C-Compiler) dazu, dass die übrigen Vektorelement mit 0 initialisiert werden.

Bsp.: double matrix[ZEILEN_ANZAHL][SPALTEN_ANZAHL] = {{1, 2, 3}, {9, 8}, {2}}

Calling Convention cdecl (Aussprache: see-DECK-‘ll) bei x86

Für den Aufruf von C-Funktionen existieren mehrere Konventionen. Aufrufkonventionen regeln die Übergabe von Argumenten an Parameter, die Rückgabe von Ergebnissen an den Aufrufer, die Organisation des Aufruf-Stacks und das Verwenden/Sichern/Wiederherstellen der CPU-Register sowie den genauen Ablauf des Aufrufs und der Rückkehr. Die ursprüngliche und bei weitem am meisten verbreitete Aufrufkonvention (cdecl) erlaubt als Spezialität die Übergabe von variabel langen Parameterlisten.

Die Übergabe der Argumente erfolgt der Reihe nach von rechts nach links. Dies ermöglicht die Übergabe verschieden vieler Argumente an ein und dieselbe Funktion (printf, scanf). Das als letztes übergebene Argument (z.B.: der Format-String bei printf/scanf) muss dazu die Information über die Anzahl und Typen (ermöglicht Rückschluss auf Byteanzahl) der Argumente enthalten. Hiermit kann die aufgerufene Funktion in weiterer Folge die Parameter konform zum Aufruf verwenden.

Die Abarbeitung der Argumente (= Berechnung der Werte der einzelnen Argumente) ist unabhängig von der Übergabereihenfolge. Seiteneffekte (++/--/+=/…) sollten in Funktionsaufrufen ohnehin tunlichst vermieden werden, da es über den Rang der Operatoren hinaus keinen Standard für die Evaluation innerhalb einer Argumentliste gilt.

Die Rückgabe des Ergebnisses erfolgt an einer vordefinierten Stelle, Register AX (EAX für 32 Bit und RAX für 64 Bit) - tw. in Kombination mit DX (EDX bzw. RDX). Die aufgerufene Funktion legt hier das Ergebnis vor dem Rücksprung ab und der Aufrufer kann es nach der Rückkehr von dort auslesen.

In cdecl werden Argumente in der Reihenfolge ‚von rechts nach links‘ am Stack abgelegt. Alleinig der Aufrufer ist bei variabler Anzahl von Argumenten in der Lage, den Stack nach dem Rücksprung zu bereinigen. Die Größe eines einzelnen Stack-Elements entspricht der Bitanzahl der Architektur (4 Byte bei 32 Bit, 8 Byte bei 64 Bit, …). Dies erklärt auch die automatische Übergabe von char und short als int (entspricht der Registerbreite) in C.

Der Compiler generiert für den Eintritt in eine Funktion und das Verlassen einer Funktion kurze Befehlssequenzen. Diese werden für das Betreten der Funktion Funktionsprolog und für den Austritt aus der Funktion Funktionsepilog genannt. Dazwischen werden die aus dem C-Code der Funktion generierten Maschinenanweisungen eingebettet.

Der Code zur Übergabe von Argumenten, dem Aufruf der Funktion, der Übernahme des Ergebnisses und dem Bereinigen des Stacks wird vom Compiler an der Aufrufstelle eingefügt und ‚Call Sequence‘ genannt.

Folgende Register werden in der Intel-x86-Architektur durch Funktionsaufrufe berührt:

- Stack Pointer SP (ESP bei 32 Bit, RSP bei 64 Bit)
  Der Stack Pointer zeigt auf das jüngste, belegte Element (top of stack) des Stacks. Dieser wächst im Speicher des Programms von oben nach unten (gegen den Heap, der von unten nach oben wächst). Der Stack Pointer wird von Maschinenanweisungen implizit verändert (PUSH, POP, CALL, RET, …). Er kann aber auch wie andere Register mit einem – hoffentlich sinnvollen/korrekten - Wert überschrieben werden.
  PUSH entspricht in C-Notation: *--SP = value
  POP entspricht in C-Notation: value = *SP++
- Base Pointer BP (EBP bei 32 Bit, RBP bei 64 Bit)

Der Base Pointer stellt die Basis für die Adressierung der Parameter und lokalen Variablen dar. Jede Funktion hat zu ihrer Laufzeit den entsprechenden Wert im Base Pointer abgelegt und adressiert die Parameter und lokalen Variablen als – negativen/positiven – Abstand dazu. Der Base Pointer muss beim Aufruf einer Funktion für die aufgerufene Funktion explizit gesetzt bzw. bei der Rückkehr aus einer Funktion für die aufrufende Funktion explizit wieder hergestellt werden.

- Instruction Pointer IP (EIP bei 32 Bit, RIP bei 64 Bit)
  Der Instruction Pointer (oftmals auch Program Counter PC genannt) weist auf die der aktuellen Anweisung unmittelbar folgende Anweisung. Er wird von Maschinenanweisungen (CALL, RET, JUMP, …) implizit gesetzt. CALL speichert den aktuellen Wert am Stack und RET restauriert diesen gespeicherten Wert für die Rückkehr aus einem Funktionsaufruf.

Ein Funktionsaufruf läuft bei cdecl in der x86-Architektur in folgender Sequenz ab (optionale Schritte in []):

1. [Vor der Parameterübergabe muss der Aufrufer die von ihm zu sichernden Register AX, CX und DX sichern. Außer die darin befindlichen Werte werden nach der Rückkehr aus dem Funktionsaufruf nicht mehr benötigt.]
2. Die aufrufende Funktion legt die Argumente von rechts beginnend nach links fortschreitend am Stack ab. Das ganz links stehende erste Argument landet im ersten Parameter und bildet das ‚top of stack' Element des call stacks (Format-String bei printf/scanf). Der Aufrufer muss über die Anzahl der PUSH-Anweisungen Buch führen, um den Stack nach der Rückkehr aus dem Funktionsaufruf bereinigen zu können. Diese Buchführung übernimmt der Compiler und erzeugt dementsprechenden Code.
3. Der Wechsel in die aufgerufene Funktion erfolgt durch die Maschinenanweisung CALL. Diese speichert den aktuellen Wert des Instruction Pointers (Adresse der ersten Anweisung nach CALL) am Stack und setzt die Zieladresse (Beginn der aufgerufenen Funktion) in den Instruction Pointer. Dadurch wird in die aufgerufene Funktionen verzweigt. Der Name einer Funktion in C entspricht der Adresse dieser Funktion zur Laufzeit (vgl. Felder).
4. In der aufgerufenen Funktion wird ein neuer, eigener Stack Frame benötigt. Dazu wird der Wert des Base Pointers, der noch zum Aufrufer gehört, am Stack abgelegt. Durch das Zuweisen des Stack Pointers (zeigt aktuell auf den gesicherten Base Pointer des Aufrufers) auf den Base Pointer wird die Basisadresse der Parameter und lokalen Variablen der aufgerufenen Funktion festgelegt.
   Der gesicherte Base Pointer des Aufrufers ist nun an der Adresse BP. Die gesicherte Rücksprungadresse (IP) steht auf BP + 4 (32-Bit-System). Der erste Parameter auf BP + 8 (32-Bit-System), usw.
5. Der Platz für die lokalen Variablen wird durch Dekrementieren des Stack Pointers um die Anzahl der von den lokalen Variablen benötigten Bytes reserviert. Dies stellt sicher, dass eventuell folgende PUSH-/POP-Anweisungen die lokalen Variablen unbehelligt lassen. Aufgrund dieser Systematik können die lokalen Variablen über den Base Pointer und dem entsprechenden Abstand (-4, -8. -12, … bei 32-Bit-Systemen) systematisch adressiert werden.
6. [Sichern der vom Aufgerufenen zu sichernden Register BX, DI und SI. Mit Ausnahme derjenigen Register, die vom Aufgerufenen nicht verändert werden.]

7. Abarbeitung des Funktionsrumpfes inklusive Berechnung des Ergebnisses – falls Rückgabetyp der Funktion != void – bis unmittelbar vor den Funktionsepilog.
8. Vor dem Aufräumen des Stacks und dem Rücksprung muss ein eventuell berechnetes Ergebnis - für den Aufrufer zugänglich - abgelegt werden. Dies ist in aller Regel das AX-Register bzw. das AX- in Kombination mit dem DX-Register für ‚übergroße' Datentypen.
9. [Restauration der vom Aufgerufenen gesicherten Register BX, DI und SI.]
10. Am Ende der Funktionsausführung müssen die lokalen Variablen wieder frei gegeben werden indem der Stack-Pointer um die Byteanzahl der lokalen Variable erhöht wird. Der Stack Pointer zeigt nun wieder - ebenso wie der Base Pointer – auf den gesicherten Base Pointer des Aufrufers.
11. Als letzte Handlung in der aufgerufenen Funktion vor der Rückkehr wird der Base Pointer des Aufrufers – ist nun top of stack - aus dem Stack restauriert.
12. Durch Aufruf der RET-Maschinenanweisung wird die Rücksprungadresse aus dem Stack in den Instruction Pointer übertragen und dorthin verzweigt. Die aufgerufene Funktion ist verlassen und der Stack auf dem Stand unmittelbar vor der CALL-Anweisung.
13. Der Aufrufer muss nun nur noch die Parameter vom Stack nehmen indem er den Stack Pointer um die Anzahl der übergebenen Bytes erhöht. Der Stack ist nun wieder soweit hergestellt, dass der Aufrufer in seiner Anweisungsfolge fortsetzen kann.
    Durch diesen Ansatz ‚Aufrufer bereinigt den Stack' ermöglicht cdecl die Übergabe variabel langer Parameterlisten.
14. Die Übernahme des in der aufgerufenen Funktion berechneten Ergebnisses ist aus dem AX-Register bzw. AX- und DX-Register zu entnehmen, bevor der darin abgelegte Wert - z.B. durch Wiederherstellen der gesicherten Registerinhalte - überschrieben wird.
15. [Restauration der vom Aufrufer gesicherten Register AX, CX und DX.]

Bezüglich CPU-Register gibt es hinsichtlich Funktionsaufrufe folgende Unterteilung im x86-Umfeld:

- Vom Aufrufer gesichert und wieder restauriert: AX, CX und DX; AX erhält Rückgabewert (evtl. in Kombination mit DX); Register müssen vor dem Bereitstellen der Argumente in den Parametern am Stack gesichert werden; Wiederherstellung dieser Register erfolgt nach dem Entfernen der Parameter vom Stack.
- Vom Aufgerufenen gesichert und restauriert: BX, DI und SI; die aufgerufene Funktion muss diese Register für den Fall sichern, dass sie deren Inhalte überschreibt (der Aufrufer rechnet nicht mit deren Änderung); dies muss nach dem Anlegen der lokalen Variablen geschehen und vor dem Freigeben derselben unmittelbar vor dem Rücksprung.

Typen/Funktionen/Konstante/... der Standardbibliothek:

Die Sprache C stellt nur Schlüsselwörter für die wesentlichen Bereiche der Programmierung zur Verfügung. Ein-/Ausgabe, Zeichenkettenverarbeitung, Mathematik, Speicherverwaltung und viele andere Bereiche werden laut ANSI-Standard über Bibliotheken bereitgestellt. Das Einbinden der entsprechenden Deklarationen muss über #include erfolgen. In der folgenden Auflistung werden die wichtigsten Header-Dateien und deren wichtigsten Deklarationen erläutert.

ctype.h

In ctype.h sind Makros und Funktionen zur Behandlung von Zeichentypen deklariert.

islower(int zeichen) beantwortet die Frage, ob übergebenes Zeichen ein Kleinbuchstabe ist.

isupper(int zeichen) beantwortet die Frage, ob übergebenes Zeichen ein Großbuchstabe ist.

isdigit(int zeichen) beantwortet die Frage, ob übergebenes Zeichen eine Ziffer ist.

isalnum(int zeichen) beantwortet die Frage, ob übergebenes Zeichen ein Buchstabe oder eine Ziffer ist.

isspace(int zeichen) beantwortet die Frage, ob übergebenes Zeichen ein Trennzeichen (Leerzeichen, Tabulator, Zeilenumbruch) ist.

isprint(int zeichen) beantwortet die Frage, ob übergebenes Zeichen ein druckbares Zeichen ist.

float.h

In float.h sind die Grenzen der Wertebereiche von float (FLT_MIN, FLT_MAX) und double (DBL_MIN, DBL_MAX) deklariert, wobei diese Werte nicht in jeder Umgebung verfügbar sind.

limits.h

Limits.h enhält die Festlegung der Wertebereiche (INT_MIN, … LONG_MAX) der einzelnen Datentypen, sodass die Gegebenheiten der jeweiligen C-Umgebung berücksichtigt werden können.

math.h

Math.h enthält ein paar nützliche Konstanten (M_PI, M_E) sowie mathematische Funktionen (sqrt, pow, exp, sin, cos, tan,...).

stdio.h

In stdio.h sind die Konstanten (stdin, stdout, stderr, EOF) und Funktionen für die Ein- und Ausgabe (scanf, fgets, printf, fflush,...) deklariert.

Für die formatierte Eingabe steht scanf zur Verfügung, für die formatierte Ausgabe wird printf angeboten. Scanf beendet den Einlesevorgang je Formatelement bei einem Zeichen, das diesem Formatelement nicht entspricht (Buchstabe bei Dezimalzahl, Dezimalpunkt bei Ganzzahl,...). Leerzeichen, Tabulator und Zeilenendezeichen beenden das Einlesen einer Zeichenkette, sodass mit scanf(“%s“, zielzeichenkette) nur ein Wort und nicht mehrere Worte eingelesen werden können.

Für das Einlesen von Text bietet sich fgets an, da diese Funktion einen Zeichenkettenüberlauf verhindert und außerdem mehrere Worte bis zum Zeilenendezeichen einliest.

Das Makro int getchar() liest gepuffert von der Eingabe (also erst nach Betätigen der Enter-Taste)

und liefert die Konstante EOF (-1) bei Fehler bzw. Dateiende. Das Dateiende wird in Windows mit Strg+Z und in Linux mit Strg+D erzeugt.

Putchar(int zeichen) schreibt das übergebene Zeichen in die Ausgabe.

In der Eingabeaufforderung bzw. shell können die Eingabe und die Ausgabe umgeleitet werden. Mit > wird eine neue Ausgabedatei erzeugt, mit >> wird an eine bestehende Ausgabedatei angehängt oder diese neu erzeugt. Mit < wird diese Datei als Eingabe anstatt der Tastatur verwendet.

stdlib.h

In stdlib.h sind nützliche allgemeine Bibliotheksfunktionen wie exit, srand, rand, deklariert.

string.h

string.h enthält die Deklaration der Funktionen (strlen, strcpy, strcat, strcmp, zur Zeichenkettenverarbeitung in C.

time.h

In time.h sind der Datentyp für eine Zeitangabe time_t und z.B. die Funktion time deklariert. In time.h befinden sich auch die Funktionen zur Formatierung von Zeitangaben.