3.1 C-Datentypen/Datentyp char

3.1.1 Beispiel 1 – Passwortgenerator

Ein Programm `makenewpass` soll zufällige Passwörter ausgeben, die aus Gründen der Anwenderfreundlichkeit nur aus (genau acht) Kleinbuchstaben bestehen sollen. Bei jedem Start des Programms soll ein Passwort ausgegeben werden:

Terminal

```
schueler@debian964:~$ makenewpass
mtzrspyv
```

Zur Lösung muss man sich drei Fragen stellen:

- Wie kommt man in C an zufällige Daten?
- Wie werden im Computer Buchstaben verwaltet?
- Wie kann man in C diese Buchstaben benutzen?

3.1.2 Zufallszahlen

In diesem Beispiel wird eine Quelle für Zufallszahlen gebraucht. Dazu gibt es zwei Funktionen, deren Prototypen in `<stdlib.h>` stehen:

- `x=rand();`
  gibt an x eine zufällige Zahl vom Typ `long int` zurück. Der Zufallsgenerator ist ein rückgekoppeltes Schieberegister mit einer internen Variablen, deren Wert bei jedem Aufruf verwendet und neu gesetzt wird. Daher gibt jeder neue Aufruf eine neue Zufallszahl.
- `srand(12345);`
  Hiermit kann man die genannte interne Variable im Zufallsgenerator einmal zum Beginn des Programms auf einen bestimmten Wert setzen (hier eben 12345). Allerdings läuft anschließend bei jedem Programmaufruf die gleiche Zufallszahlenfolge ab. Besser ist es, wenn man diesen Wert selbst auch zufällig wählt. Oft nimmt man für diesen Wert die aktuelle Uhrzeit (in Sekunden seit dem 1.1.1970), die man sich aus dem Funktionsaufruf `time(0)` holen kann. Dazu muss vorher allerdings noch `<time.h>` eingebunden werden.

Ein komplettes Beispiel für einen Würfel zeigt `src/wuerfel.c`:

```c
#include <stdio.h>
#include <stdlib.h> /* fuer rand() und srand() */
#include <time.h>   /* fuer time(0) */

int main(void)
{
    long zahl;
    srand(time(0)); /* Initialisierung des Zufalls-Generators */
    printf("Computer-Wuerfel, Ende mit Strg-C\n");
    while(1)
    {
        zahl=rand(); /* Ergebnis zwischen 0 und 2 Mrd. */
        zahl=zahl%6;   /* Ergebnis zwischen 0 und 5 */
        ++zahl;        /* Ergebnis zwischen 1 und 6 */
        printf("%d\n", zahl);
        sleep(1);
    }
    return 0;
}
```

3.1.3 ASCII

Im obigen Beispiel geht es zum ersten Mal nicht um die Verarbeitung von Zahlen, sondern von Buchstaben.

Statt von Buchstaben spricht man in der Informatik von alphanumerischen Zeichen. Das umfasst dann:

- die Buchstaben `A` bis `Z` und `a` bis `z`
- die Ziffernzeichen; das sind die Zeichen 0 bis 9, aus denen unsere Zahlen bestehen
- Satz- und Sonderzeichen wie Punkt, Komma, Semikolon und andere

Nun braucht man eine Vereinbarung, welches Bitmuster im Computer oder auf einem Datenträger welches Zeichen darstellen soll. Bei Zahlen ist das einfach, dort gibt es Zahlensysteme, bei denen man zu jedem Bitmuster die zugehörige Zahl mittels einer Formel berechnen kann. Bei Buchstaben-, Ziffern- und Sonderzeichen ist das nicht möglich. Und so könnte man jede Zuordnung als eigene Hausnorm etablieren (und manche Firmen haben das, zumindest teilweise, wirklich gemacht).

Die am weitesten benutzte Möglichkeit, Bitmuster zu alphanumerischen Zeichen zuzuordnen, ist ASCII[1]. In Tabelle 1 wird er dargestellt, wobei das jeweilige Bitmuster der Einfachheit halber durch die entsprechende Zahl (Code-Nummer) ersetzt wurde. Die ersten 32 Zeichen sind Steuerzei-

Nr.	Z.	Nr.	Z.	Nr.	Z.	Nr.	Z.	Nr.	Z.	Nr.	Z.	Nr.	Z.	Nr.	Z.
0		16		32	␣	48	0	64	@	80	P	96	`	112	p
1		17		33	!	49	1	65	A	81	Q	97	a	113	q
2		18		34	"	50	2	66	B	82	R	98	b	114	r
3		19		35	#	51	3	67	C	83	S	99	c	115	s
4		20		36	$	52	4	68	D	84	T	100	d	116	t
5		21		37	%	53	5	69	E	85	U	101	e	117	u
6		22		38	&	54	6	70	F	86	V	102	f	118	v
7		23		39	'	55	7	71	G	87	W	103	g	119	w
8		24		40	(	56	8	72	H	88	X	104	h	120	x
9		25		41	)	57	9	73	I	89	Y	105	i	121	y
10		26		42	*	58	:	74	J	90	Z	106	j	122	z
11		27		43	+	59	;	75	K	91	[	107	k	123	{
12		28		44	,	60	<	76	L	92	\	108	l	124	|
13		29		45	-	61	=	77	M	93	]	109	m	125	}
14		30		46	.	62	>	78	N	94	^	110	n	126	~
15		31		47	/	63	?	79	O	95	_	111	o	127	

Tabelle 1: ASCII

chen, danach folgen einige Satz- und Sonderzeichen. Ab Code-Nummer 48 folgen die Ziffernzeichen, ab 65 die Groß- und ab 97 die Kleinbuchstaben in alphabetischer Reihenfolge. Zeichen 127 ist nicht druckbar.

In diesem Code fehlen alle Umlaute und der Buchstabe ß, ebenso die Akzente und Sonderzeichen anderer europäischer Sprachen. Für solche Zeichen gibt es deshalb andere Codes, die zu ASCII kompatibel sind. Ihre Benutzung in der Programmierung ist aber nicht immer ganz einfach[2]. Es empfiehlt sich daher, diese Zeichen zunächst wegzulassen.

[1] **a**merican **s**tandard **c**ode for **i**nformation **i**nterchange, bitte nicht "aski-2" sagen!

[2] Die Gründe liegen einerseits darin, dass in PCs verschiedene Codes verwendet werden (IBMPC, Latin-1, UTF-8, UCS-2), andererseits darin, dass die Sprache C mehrere Möglichkeiten (Multibyte-Sequenzen, Wide Characters) bietet, mit diesen Codes und ihren Zeichen umzugehen

3.1.4 Ausgabe von Zeichen

In C wird *nicht* zwischen Zeichen- und Zahlendaten unterschieden. Das heißt, das gleiche Bitmuster kann eine Zahl oder ein Zeichen *bedeuten*. Deshalb braucht man in C auch keinen speziellen Datentyp nur für Zeichen. Es gibt aber (siehe später) einen Datentyp, in den ein ASCII-Zeichen genau hineinpasst.

In C ist also kein Unterschied zwischen Zahlen und Zeichen? Am einfachsten sieht man das an der Funktion `putchar`. Sie dient zur Ausgabe eines Zeichens. Man übergibt die ASCII-Codenummer einfach als `int`-Zahl:

ausgabe_a.c

```
1 #include <stdio.h>
2 int main(void)
3 {
4     putchar(65);
5     putchar(10);
6     return 0;
7 }
```

1 `putchar` ist wie `printf` in `stdio.h` deklariert.

4 Der Aufruf `putchar(65)` gibt ein `A` aus.

5 Der Aufruf `putchar(10)` gibt ein `\n` aus. An der Code-Nr. 10 liegt das Steuerzeichen für den Zeilenvorschub.

Terminal

```
schueler@debian964:~$ gcc ausgabe_a.c
schueler@debian964:~$ a.out
A
```

Statt `putchar` kann man auch `printf` für die Ausgabe eines Zeichens benutzen. Der passende Platzhalter ist `%c` :

```
1     putchar(65);       // gibt A aus
2     printf("%d", 65); // gibt erst 6, dann 5 aus
```

Ein Beispiel für die Ausgabe mehrerer Zeichen ist das folgende Programm:

src/ascii_int.c

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int zeichen;
5      for(zeichen=32; zeichen<127; ++zeichen)
6      {
7          printf("Zeichen␣Nr.␣%d:␣", zeichen);
8          putchar(zeichen);
9          putchar(10);
10     }
11     return 0;
12 }
```

Das Programm gibt alle ASCII-Zeichen von 32 bis 126 aus (0 bis 31 und 127 sind Steuerzeichen, die gibt man besser nicht aus):

Terminal
```
schueler@debian964:~$ a.out
Zeichen Nr. 32:
Zeichen Nr. 33: !
Zeichen Nr. 34: "
...
Zeichen Nr. 125: }
Zeichen Nr. 126: ~
```

3.1.5 Pufferung der Ausgabe

Wenn man etwas herumspielt, kommt man auf die Idee, die Zeichen langsam, eins nach dem anderen auszugeben:

src/ascii_langsam0.c
```
1  #include <stdio.h>
2  #include <unistd.h> // fuer sleep()
3  int main(void)
4  {
5      int zeichen;
6      for(zeichen=32; zeichen<128; ++zeichen)
7      {
8          putchar(zeichen);
9          sleep(1);
10     }
11     putchar(10);
12     return 0;
13 }
```

Nun die Praxis:

Terminal
```
schueler@debian964:~$ a.out
(.... 26 Sekunden passiert nichts ....)
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Das war nicht wie gewünscht: Anstatt nach jedem Zeichen eine Sekunde zu warten, wartet das Programm 26 Sekunden und gibt dann alle Zeichen auf einmal aus. Woran liegt das?

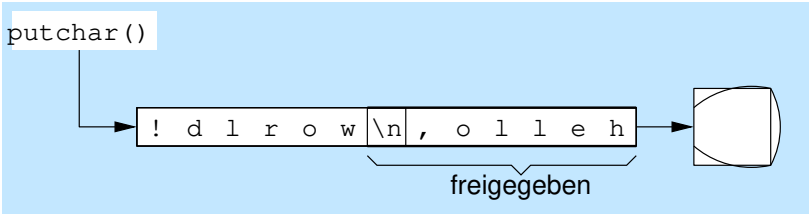

Abbildung 1: Ausgabe-Pufferung

Die C-Standard-Bibliothek benutzt für die Ausgabe auf den Bildschirm eine sogenannte *Pufferung*. Das heißt, es wird nicht jedes Zeichen einzeln über einen Betriebssystemaufruf zum Bildschirm geschickt. Stattdessen wird gesammelt, bis es sich lohnt, alle Zeichen auf einmal abzuschicken. Den Speicherbereich dazu nennt man *Puffer* (Abbildung 1). Standardmäßig wird der Puffer dann geleert, wenn eins der folgenden Ereignisse eintritt:

a) Der Puffer ist voll

b) Es wird ein Zeilenumbruch in den Puffer geschrieben

c) Es kommt ein Befehl, den Puffer jetzt zu leeren

Bis der Puffer voll ist, kann es ziemlich lange dauern. Deshalb ist standardmäßig *Zeilenpufferung* eingestellt. Das bedeutet, dass der Puffer bei einem Zeilenumbruch geleert wird. So kann man jede Ausgabe, die mit `\n` endet, sofort lesen.

Manchmal möchte man den Puffer-Inhalt aber sofort ausgeben (wie in unserem Beispiel). Dazu gibt es einen Befehl:

```
1    fflush(stdout);
```

Der Parameter (hier `stdout`) bezeichnet den Ausgabekanal, der geleert werden soll. `stdout` ist dabei (wie) eine symbolische Konstante für die normale Bildschirmausgabe. Das korrigierte Programm sieht dann so aus:

src/ascii_langsam0.c

```
1  #include <stdio.h>
2  #include <unistd.h> // fuer sleep()
3  int main(void)
4  {
5      int zeichen;
6      for(zeichen=32; zeichen<128; ++zeichen)
7      {
8          putchar(zeichen);
9          sleep(1);
10     }
11     putchar(10);
12     return 0;
13 }
```

Nun funktioniert die langsame Ausgabe auch in der Praxis.

3.1.6 Zeichenkonstanten

Anstelle der ASCII-Codenummer kann man auch das zugehörige Zeichen in Hochkomma schreiben. So ersetzt `'A'` im Quelltext die Zahl `65`. Den Ausdruck `'A'` nennt man eine *Zeichenkonstante*. Sie hat den Datentyp `int`. Ihr Wert ist der die Codenummer des entsprechenden Zeichens[3]. Bei einem ASCII-Zeichensatz ist die Zahlenkonstante `65` also gleichbedeutend mit der Zeichenkonstante `'A'`. Auch die Ersatzdarstellungen für Steuerzeichen können als Zeichenkonstanten verwendet werden (wenn man will). Im folgenden Beispiel wird auf sechs verschiedene Arten die Zahl `65` in der Variablen `zahl` gespeichert:

```
1      int zahl;
2      zahl=65;    // dezimal
3      zahl=0101; // oktal
4      zahl=0x41; // hexadezimal
5      zahl='A';      // Zeichenkonstante
6      zahl='\0101'; // Zeichenkonstante, oktale Ersatzdarstellung
7      zahl='\0x41'; // Zeichenkonstante, hexad. Ersatzdarstellung
```

Genauso geht das auch bei einem Steuerzeichen, z. B. dem Zeilenumbruch `\n`:

[3] auf dem Computer, *für* den das Programm compiliert wird. Solange man keinen Cross-Compiler benutzt, ist das aber unwichtig.

```
1     int zahl;
2     zahl=10;  // dezimal
3     zahl=012; // oktal
4     zahl=0xc; // hexadezimal
5     zahl='\n';    // Zeichenkonstante, Ersatzdarstellung
6     zahl='\012'; // Zeichenkonstante, oktale Ersatzdarstellung
7     zahl='\0xc'; // Zeichenkonstante, hexad. Ersatzdarstellung
```

3.1.7 Lösung des Beispiels 1

Jetzt kann man das anfangs genannte Beispiel lösen. Eine mögliche Lösung ist diese:

```
1  #include <stdio.h>
2  #include <stdlib.h> /* fuer rand() und srand() */
3  #include <time.h>   /* fuer time()    */
4
5  int main(void)
6  {
7     int zeichen, lauf;
8
9     srand(time(0));
10    for(lauf=0; lauf<8; ++lauf)
11    {
12       zeichen = rand()%26+'a';
13       putchar(zeichen);
14    }
15    putchar('\n');
16    return 0;
17 }
```

3.1.8 Beispiel 2 – Menüauswahl

Ein Programm soll ein Menü ausgeben. Der Benutzer soll seinen Wunsch durch Eingabe eines Buchstabens (gefolgt von der Return-Taste) äußern dürfen:

Terminal

```
schueler@debian964:~$ simpelmenue
---------------
<H> Hallo sagen
<Q> Quit
---------------
Ihre Auswahl: H
Hallo, Welt!
---------------
<H> Hallo sagen
<Q> Quit
---------------
Ihre Auswahl: Q
Auf Wiedersehen!
schueler@debian964:~$
```

Bisher waren unsere Menü-Programme so angelegt, dass der Benutzer eine Zahl eingeben musste. Ab jetzt soll es auch möglich sein, dass man ein anderes Zeichen zur Auswahl benutzen kann.

3.1.9 Eingabe und Eingabe-Pufferung

Was bis jetzt fehlt, ist ein Funktionsbaustein, mit dem man ein einzelnes Zeichen von der Tastatur lesen und in eine Variable schreiben kann. Dazu gibt es den Funktionsbaustein `getchar`. Hier ein Beispiel:

eingabea.c

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int z;
5     printf("Bitte ein Zeichen eingeben: ");
6     z=getchar();
7     printf("Eingelesen: Nr. %d\n", z);
8     return 0;
9 }
```

6 Mit dieser Anweisung liest man ein Zeichen von der Tastatur und schreibet es in die `int`-Variable `z`.

Terminal

```
schueler@debian964:~$ a.out
Bitte ein Zeichen eingeben:  A  ↵
Eingelesen: Nr. 65
```

Man sieht: Der Funktionsbaustein `getchar` hält das Programm an. Es läuft erst dann weiter, wenn eine Eingabe erfolgt ist. Man spricht von *blocking input* (=blockierender Eingabe). [4]. Dasselbe Verhalten ist von `scanf` bekannt.

Das obige Programm hat eine weitere interessante Eigenschaft: Es läuft nämlich erst dann weiter, wenn auf der Tastatur die ↵ -Taste gedrückt wurde. Das liegt daran, dass auch die Eingabe gepuffert wird. Bild 2 zeigt das Problem. Solange im Eingabepuffer kein Zeilenumbruch

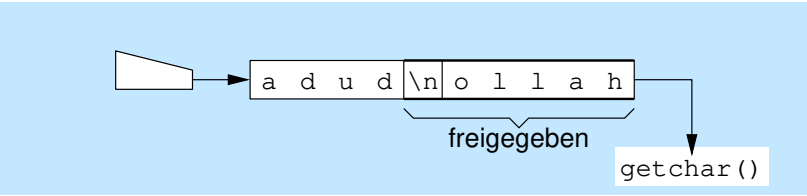

Abbildung 2: Eingabe-Pufferung

(`\n`) vorliegt, wird weiter gesammelt. Erst danach kann sich `getchar` ein Zeichen aus dem Puffer holen.

3.1.10 Einlesen mehrerer Zeichen

Der Funktionsbaustein `getchar` kann immer nur ein einziges Zeichen aus dem Puffer holen – egal, wie viele Zeichen eingegeben wurden. Für das Einlesen mehrerer Zeichen braucht man mehrere Aufrufe. Im obigen Beispiel wurde A und ↵ gedrückt. Also liegen `'A'` und `'\n'` im Eingabepuffer. Mit dem ersten Aufruf von `getchar` kann man nur das `'A'` holen; man braucht einen zweiten Aufruf, um auch den Zeilenumbruch `'\n'` herauszuholen – erst dann ist der Puffer wieder leer und kann für neue Aufgaben benutzt werden:

[4] Es gibt auch Funktionsbausteine mit *non blocking input*.

eingabe_b.c

```c
#include <stdio.h>
int main(void)
{
    int z;
    printf("Bitte ein Zeichen eingeben: ");
    z=getchar();
    printf("Eingelesen: Nr. %d\n", z);
    z=getchar();
    printf("Eingelesen: Nr. %d\n", z);
    return 0;
}
```

6 Das `'A'` wird gelesen

8 Das `'\n'` wird gelesen

3.1.11 Lösung des Beispiels 2

Jetzt kann man auch das zweite Beispiel lösen. Hier eine mögliche Lösung:

```c
#include <stdio.h>
int main(void)
{
    int dummy, auswahl;
    do{
        printf("-----------------\n"
               "<H> Hallo sagen\n"
               "<Q> Quit\n"
               "-----------------\n"
               "Ihre Auswahl: \n");
        auswahl=getchar();
        dummy  =getchar();
        switch(auswahl)
        {
        case 'h':
        case 'H':
            printf("Hallo, Welt!\n");
            break;
        case 'q':
        case 'Q':
            break;
        default:
            printf("Fehleingabe!\n");
        }
    }while(auswahl!='q' && auswahl!='Q');
    printf("Auf Wiedersehen!\n");
    return 0;
}
```

3.1.12 Einlesen vieler Zeichen und das Ende der Eingabe

Im folgenden Beispiel sollen ganz viele Zeichen von der Tastatur gelesen und sofort wieder auf den Bildschirm ausgegeben werden. Sobald das Zeichen `'A'` gelesen wurde, soll das Programm kein `A`

ausgeben, sondern sich beenden:

eingabe_c.c

```c
#include <stdio.h>
int main(void)
{
    int z;
    printf("Bitte viele Zeichen eingeben (Ende mit 'A'):\n");
    z=getchar();
    while(z!='A')
    {
        putchar(z);
        z=getchar();
    }
    putchar('\n');
    return 0;
}
```

6 Erstes Zeichen einlesen

7 Prüfen, ob schon Ende

9 Nutzen des Programms: Ausgabe

10 Weiteres Zeichen einlesen und zurück nach 7

12 Nur zur Dekoration

Terminal

```
schueler@debian964:~$ a.out
Bitte viele Zeichen eingeben (Ende mit 'A'):
Viel Text ist
Viel Text ist
das hier.
das hier.
Keine Ahnung.
Keine
schueler@debian964:~$
```

Dieses Programm hat eine typische Struktur für das Einlesen von Daten. Abbildung 3 zeigt links die gewählte Struktur und rechts eine andere Möglichkeit. Wichtig ist es, das das Ende-Zeichen (hier `A`) *nicht* ausgegeben wird, deshalb *muss* in der rechten Struktur die Verzweigung stehen. Eine der beiden Strukturen sollte man auswendig lernen, weil sie immer wieder benötigt wird.

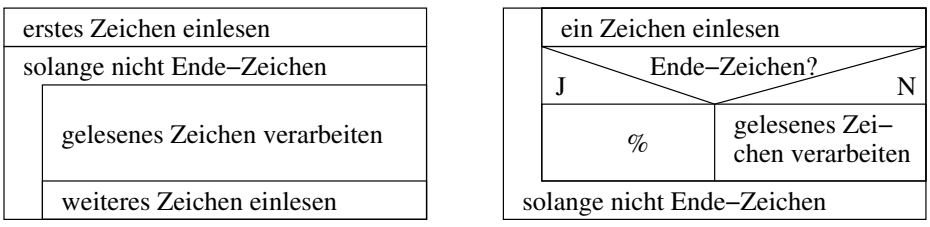

Abbildung 3: Typische Programmstrukturen beim Einlesen von Daten

Was ist aber dann, wenn man nicht mit einem `'A'` das Ende der Eingabe bezeichnen will? Dafür hat der Funktionsbaustein `getchar` eine Besonderheit: `getchar()` gibt das Ende der Eingabe bekannt durch den Rückgabewert `EOF` (Zahlenwert: -1, Bitmuster: lauter Einsen).

Wenn man die Eingabe mittels Eingabeumleitung aus einer Datei holt, wird mit `EOF` das das Dateiende erkannt (EOF=*end of file*, siehe Abbildung 4). Bei der Eingabe von der Tastatur

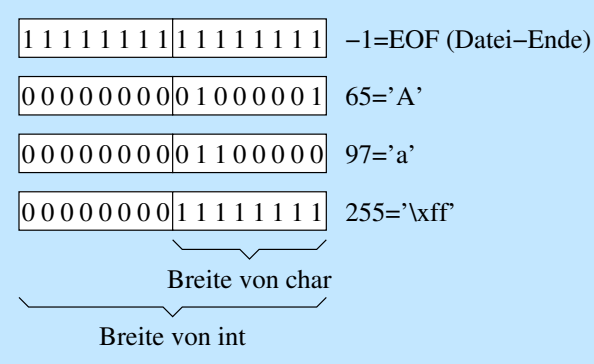

Abbildung 4: Rückgabewert von `getchar`

simuliert man das Dateiende, indem man Strg - D drückt.

Ein Beispiel zeigt das Programm `src/caesar.c`:

```c
#include <stdio.h>

int main(void)
{
    int zeichen;
    printf("Bitte Text eingeben (Ende mit Strg-D oder Strg-Z):\n");
    zeichen = getchar(); /* erstes Zeichen holen */
    while(zeichen != EOF)
    {
        putchar(zeichen+1);
        zeichen = getchar(); /* neues Zeichen holen */
    }
    printf("Ende.\n");
    return 0;
}
```

Wenn man das Programm ausprobiert, kann man beliebig viele Zeichen eingeben. Nach dem Drücken der ↩ -Taste arbeitet die Schleife jedes Zeichen aus dem Eingabepuffer ab. Mit dem Drücken von Strg - D [5] oder Strg - Z [6] gibt die Funktion `getchar()` den Wert `EOF` zurück, und die Schleife (und damit das Programm) wird beendet.

Man kann das Programm auch mit einer Eingabeumleitung aufrufen, z.B. erstellt man mit einem Texteditor die Textdatei `beispiel.txt` und ruft dann auf:

Terminal

```
schueler@debian964:~$ cat beispiel.txt
Das ist das Haus vom Nikolaus.
schueler@debian964:~$ gcc -o caesar caesar.c
schueler@debian964:~$ ./caesar < beispiel.txt
Bitte Text eingeben (Ende mit Strg-D oder Strg-Z):
Ebt!jtu!ebt!Ibvt!wpn!Ojlpmbvt/
                                Ende.
```

[5] muss zweimal gedrückt werden, falls man nicht am Zeilenanfang ist

[6] bei Windows

3.1.13 Der Datentyp `char`

Eigentlich braucht man für die Speicherung eines ASCII-Zeichens nicht unbedingt eine `int`-Variable. Je nach System braucht eine `int`-Variable 16, 32 oder 64 Bits, von denen aber nur acht benutzt werden. Deshalb gibt es spezielle Datentypen für alphanumerische Daten, die immer dann benutzt werden sollten, falls ganze Texte bearbeitet werden sollten:

a) Datentyp `char`: 8 Bit, Bereich -128 bis 127 oder 0 bis 255, daher nur für ASCII geeignet (sonst weiß man nicht, ob Codes größer als 127 negativ oder positiv sind) – dieser Typ ist für Texte üblich

b) Datentyp `unsigned char`: 8 Bit, Bereich 0 bis 255 – dieser Typ wird eher für die Hardware-nahe Programmierung eingesetzt

c) Datentyp `signed char`: 8 Bit, Bereich -128 bis 127

Trotzdem muss der Rückgabewert von `getchar` *immer* in einer `int`-Variablen gespeichert werden!

3.1.14 Zeichenklassen

Manchmal soll ein Programm unterschiedlich reagieren, je nachdem, ob ein Buchstabe oder eine Ziffer eingetippt wurden. Beispielsweise sind an einer Stelle nur Ziffern erlaubt. Das kann man nun so programmieren:

```
1    if(x=='0'||x=='1'||x=='2'||x=='3'||x=='4'
2    || x=='5'||x=='6'||x=='7'||x=='8'||x=='9')
3    { ... }
```

Das ist zwar richtig und portabel, aber sehr umständlich. Einfacher geht es, wenn man die Tatsache benutzt, dass bei den meisten alphanumerischen Codes alle Ziffernzeichen in aufsteigender Reihenfolge hintereinander liegen:

```
1    if(x>='0' && x<='9')
2    { ... }
```

Diese Lösung ist leider nicht portabel, sondern vom jeweiligen Code abhängig. Noch einfacher und dabei portabel geht es mit einer speziellen Funktion aus der C-Standard-Bibliothek:

```
1 #include <ctype.h>
2    if(isdigit(x))
3    { ... }
```

Die Funktion `isdigit(x)` gibt zu einer Zahl x an, ob das betreffende Zeichen eine Ziffer ist oder nicht. Bei einer Ziffer wird ein Wert ungleich null ausgegeben (entspricht TRUE), andernfalls eine Null (entspricht FALSE). Dazu muss die Header-Datei `<ctype.h>` eingebunden worden sein.

Die verschiedenen Arten von Zeichen (Ziffer, Buchstabe, Satzzeichen, ...) nennt man in C *Zeichenklassen*. Tabelle 2 zeigt die in C festgelegten Zeichenklassen und die dazu passenden Funktionen. Abbildung 5 zeigt, wie die wichtigsten Zeichenklassen zusammenhängen.

Wenn man ein Zeichen von Groß- in Kleinschreibung umwandeln will oder umgekehrt, dann gibt es dafür die Funktionen `tolower` und `toupper`:

```
1 #include <ctype.h>
2    int x='A';
3    x=tolower(x);
4    putchar(x); // gibt 'a' aus.
```

	Zeichenklasse	Funktion	Beschreibung
1	alnum	isalnum	Ziffer oder Buchstabe
2	alpha	isalpha	Buchstabe
3	cntrl	iscntrl	Steuerzeichen (0. . . 32 und 127)
4	digit	isdigit	Dezimalziffer
5	graph	isgraph	sichtbares Zeichen ohne Leerzeichen
6	lower	islower	Kleinbuchstabe
7	print	isprint	sichtbares Zeichen mit Leerzeichen
8	punct	ispunct	Satzzeichen (sichtbares Zeichen, aber kein Leerzeichen, kein Buchstabe und keine Ziffer)
9	space	isspace	Leerzeichen, `\f`, `\n`, `\t`, `\t` oder `\v`
10	upper	isupper	Großbuchstabe
11	xdigit	isxdigit	Hexadezimalziffer (Ziffer oder Buchstabe A. . . F bzw. a. . . f

Tabelle 2: Zeichenklassen

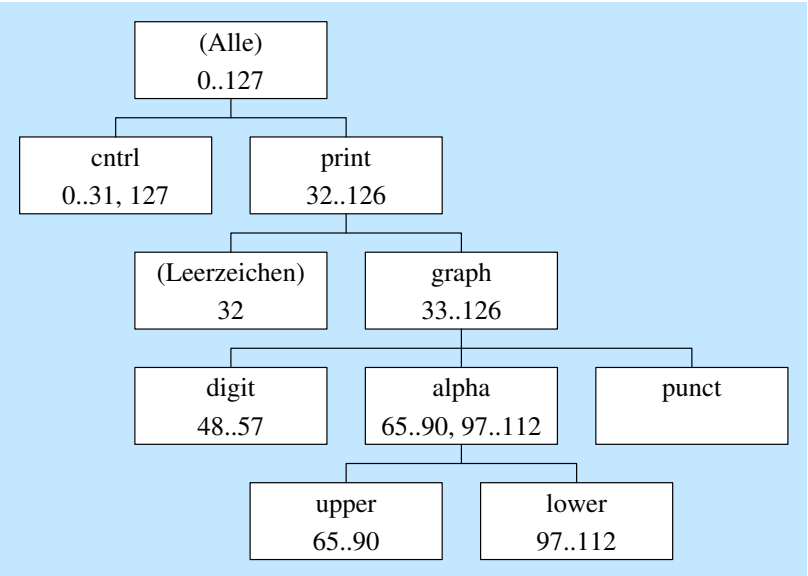

Abbildung 5: Zusammenhang zwischen den Zeichenklassen