Zeichen

Zeichen werden als ganze Zahlen im Bereich 0-127 repräsentiert (die ersten 32 Zeichen sind Steuerzeichen, die folgenden 96 sichtbare). Um gegenüber `int` Speicherplatz zu sparen, existiert ein Ganzzahltyp `char`, welcher je nach Plattform Zahlen von -128 bis +127 (entspricht `signed char`) oder 0 bis 255 (entspricht `unsigned char`) darstellen kann.

Die Umwandlung zwischen Zahlen und Zeichen geschieht nur während der Ein- und Ausgabe – `char` speichert Zahlen, mit denen ganz normal gerechnet werden kann.

Der Compiler unterstützt den Programmierer durch die Bereitstellung von Literalen, die in einfachen Hochkommata angegeben werden – diese generieren zur Übersetzungszeit die entsprechenden Zahlen (im ASCII-Code wird aus `'0'` z.B. 48, aus `'a'` wird 97 usw).

String-Literale

String-Literale werden durch Anführungszeichen begrenzt. Um ein Anführungszeichen innerhalb eines String-Literals zu generieren, muss man `\"` schreiben. Es gibt weitere Backslash-Escapes wie `\n` (neue Zeile) oder `\t` (Tabulator). Um einen Backslash selbst in einem String-Literal zu erhalten, muss man `\\` schreiben. Backslash-Escapes sehen im Quelltext wie zwei Zeichen aus, tatsächlich verbrauchen sie aber nur ein Zeichen.

Längere String-Literale kann man durch Hintereinanderschreiben von kürzeren zusammenbauen: "Dampfschiff" "fahrtskapitän" wird zu "Dampfschifffahrtskapitän".

String-Literale sind Konstanten, d.h. `"Hello World."[11] = '!';` ist unzulässig!

Zeichenweise Eingabe

Ein einzelnes Zeichen kann man per `int c = getchar();` einlesen. Eingaben über die Konsole werden vom Betriebssystem gepuffert, d.h. das Programm blockiert beim Aufruf der Funktion `getchar` so lange, bis der Benutzer die Eingabe per Enter bestätigt. Erst dann wird das erste eingegebene Zeichen an das Programm weitergeleitet, und nachfolgende Aufrufe von `getchar` liefern die nächsten Zeichen der eingegebenen Zeile.

(Interessiert man sich lediglich für das erste Zeichen einer eingegebenen Zeile, dann muss man alle nachfolgenden Zeichen bis zum Zeilenumbruch explizit überlesen.)

Beim Umleiten der Standardeingabe aus einer Datei gibt es diese Verzögerungen nicht. Das Ende einer Datei signalisiert `getchar` durch den Rückgabewert `EOF`. Auf der Konsole kann man `EOF` per Strg+Z (unter Windows) bzw. Strg+D (unter Linux) simulieren.

Der Zeilenumbruch `\n` ist etwas völlig anderes als `EOF` (ansonsten könnte eine Datei nicht aus mehreren Zeilen bestehen). Das Standardidiom zum Einlesen aus einer Datei lautet:

```
int c; while ((c = getchar()) != EOF) verarbeite(c);
```

Die fett gedruckten Klammern sind notwendig, weil der Vergleichsoperator `!=` eine höhere Präzedenz hat als der Zuweisungsoperator =. Ohne die Klammern würde zunächst geprüft werden, ob `getchar` den Wert `EOF` zurückgeliefert hat, und das Ergebnis dieses Vergleichs (entweder 0 oder 1) würde dann in `x` gespeichert werden.

Arrays

Ein Array ist eine Sequenz fester Länge von Zellen desselben Typs. Die Länge des Arrays muss zur Übersetzungszeit bekannt sein. Ein Array wird wie folgt definiert:

```
int zaehler[26];      // zaehler besteht aus 26 int-Zellen
```

Die einzelnen Zellen des Arrays können mit dem Operator `[]` angesprochen werden:

```
foo = zaehler[0];     // erste Zelle lesen
zaehler[25] = bar;    // letzte Zelle schreiben
++zaehler[i];         // Erhöhen der Zelle i um 1
```

Ein Array der Länge n besteht aus den Zellen 0 bis n-1, nicht jedoch aus der Zelle n selbst! Jeder Zugriff außerhalb der festgelegten Arraygrenzen führt zu undefiniertem Verhalten.

Die Initialisierung eines Arrays mit bekannten Werten geschieht über folgende Syntax:

```
int tage[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Die Länge des Arrays wird dann automatisch aus der Anzahl der Werte berechnet.

Falls man trotzdem eine Länge angibt, werden die restlichen Zellen mit 0 initialisiert:

```
int a[5] = {8, 15};   // entspricht int a[] = {8, 15, 0, 0, 0};
```

Die Länge eines Arrays kann nur innerhalb des Kontexts ermittelt werden, in welchem das Array definiert wurde. Standardidiom zum Berechnen der Länge zur Übersetzungszeit:

```
int laenge = sizeof array / sizeof array[0];
```

Arrays können einander nicht zugewiesen werden. Stattdessen verwendet man eine Funktion aus der Standardbibliothek, welche zum Kopieren von Bytes im Speicher dient:

```
memcpy(ziel, quelle, anzahl_bytes);
```

Zeichenketten (C-Strings)

Da eine Zeichenkette nichts weiter als eine Sequenz von Zeichen ist, verwendet man in C Arrays mit dem Zellentyp `char`, um Zeichenketten zu realisieren. Viele Funktionen aus der Standardbibliothek (`strlen`, `strcpy`...) verlassen sich darauf, dass das letzte Zeichen einer Zeichenkette die Zahl 0 ist. Man spricht daher von „0-terminierten Zeichenketten“.

```
char genie[] = {'A', 'l', 'b', 'e', 'r', 't', ' ', 'E', 'i',
'n', 's', 't', 'e', 'i', 'n', 0};
```

Da diese Form der Initialisierung langatmig und fehleranfällig ist (man muss selbst an die 0 denken), erlaubt der Compiler auch String-Literale zur Initialisierung von Zeichenketten:

```
char genie[] = "Albert Einstein";
```

Man beachte, dass die terminierende 0 bereits Teil des String-Literals ist und im Zuge der Initialisierung mitkopiert wird. Man muss sich hier nicht mehr selbst um die 0 kümmern!

Zeichenketten können (wie andere Arrays auch) einander nicht zugewiesen werden. Die Standardbibliothek stellt zum Kopieren von Zeichenketten eine eigene Funktion bereit:

```
strcpy(ziel, quelle);
```

Zeiger

Ein Zeiger ist eine getypte Speicheradresse. Zu jedem Typ `T` gibt es einen Zeigertyp `T *`.

Zeiger erhält man, indem man den Operator `&` auf ein Objekt anwendet (`&v` ist ein Zeiger auf `v`) oder per Zeigerarithmetik auf bereits bestehenden Zeigern rechnet. String-Literale liefern in den meisten Kontexten ebenfalls Zeiger (auf ihr erstes Zeichen).

```
int i, * p, * * q;
char c, * x, * y;
p = &i;            // p zeigt auf i
q = &p;            // q zeigt auf p
x = "Test";        // x zeigt auf 'T' von "Test"
++x;               // x zeigt auf 'e' von "Test"
y = x + 3;         // y zeigt auf die terminierende 0 von "Test"
```

Zeigerarithmetik arbeitet auf dem Zeiger selbst. Um sich auf das Objekt zu beziehen, das der Zeiger referenziert, verwendet man den Dereferenzierungsoperator `*`

```
 *p = 5;           // i ist jetzt 5
**q = 7;           // i ist jetzt 7
  c = *x;          // c ist jetzt 'e'
```

Der `NULL`-Zeiger `((void *)0)` zeigt auf kein Objekt. In Bedingungen entspricht er *false*.

Der generische Zeigertyp `void *` ist der einzige ungetypte Zeiger und kann weder per Zeigerarithmetik manipuliert noch dereferenziert werden. `void *` kann implizit in jeden anderen `T *` konvertiert werden und umgekehrt (siehe zum Beispiel `memcpy`).

Arrays und Zeiger

`a` ist eine veränderbare Zeichenkette und verbraucht 13 Byte im Speicher:

```
char a[] = "Hello World.";   // a[11] = '!' ist zulässig
```

`p` ist ein Zeiger auf eine statische Zeichenkette und verbraucht 4 oder 8 Byte im Speicher:

```
char * p = "Hello World.";   // p[11] = '!' ist unzulässig
```

Das Vorkommen eines Arrays innerhalb eines Ausdrucks bezieht sich nur dann auf das komplette Array, wenn man die Operatoren `sizeof` oder `&` darauf anwendet oder ein `char[]` mit einem String-Literal initialisiert. Ansonsten wird implizit ein Zeiger auf die erste Zelle des Arrays generiert, wobei die Längeninformation des Arrays verloren geht, und anschließend wird mit diesem Zeiger weitergearbeitet (d.h. `T[n]` → `T *`).

Selbst bei der Indizierung eines Arrays `a[i]` wird aus `a` zunächst ein Zeiger generiert und dann mit Zeigerarithmetik und Dereferenzierung gearbeitet – **`a[i]`** ist bloß syntaktischer Zucker für **`*(a+i)`**. Streng genommen ist `[]` für Arrays also gar nicht definiert!
`&a[i]` entspricht somit `&*(a+i)`, was sich zu **`a+i`** vereinfachen lässt.

Ein `T *` kann sowohl auf ein einzelnes `T` als auch auf eine Zelle in einem `T`-Array zeigen. Zeigerarithmetik ist nur im zweiten Fall sinnvoll und wohldefiniert. Weiterhin ist die Differenzbildung nur für Zeiger erlaubt, die auf Zellen desselben Arrays zeigen.

Arrays und Funktionen

Bei der Übergabe von Argumenten an Funktionen wird aus Effizienzgründen statt eines Arrays ein Zeiger auf seine erste Zelle übergeben. Das bedeutet, dass C keine Funktionen vorsieht, die Arrays als Kopie entgegennehmen.

Scheinbar kann man solche Funktionen zwar schreiben:

```
void f(int a[]);
```

Die Array-Notation ist **in Parameterlisten** jedoch lediglich syntaktischer Zucker für:

```
void f(int * a);
```

Diese Ersetzung von `a[]` in `* a` geschieht ausschließlich innerhalb von Parameterlisten, nicht jedoch an anderen Stellen im Quelltext. **Arrays und Zeiger sind nicht dasselbe!**

Im Kontext des Funktionsrumpfs würde `sizeof a` also die Größe eines Zeigers auf `int` ermitteln (meist 4 oder 8) und nicht etwa die Größe des referenzierten Arrays. Letztere muss entweder explizit mitübergeben werden (als weiteres Argument), oder sie muss per Konvention aus der Struktur des Arrays selbst erkennbar sein; das prominenteste Beispiel einer solchen Konvention ist die terminierende 0 für das Ende von Zeichenketten.

Ein gutes Beispiel für die hier angesprochenen Punkte ist die Funktion `main`:

```
int main(int argc, char * argv[]);
```

ist lediglich syntaktischer Zucker für:

```
int main(int argc, char * * argv);
```

`argv` zeigt dabei auf den ersten von vielen `char *`, wobei jeder einzelne davon wiederum auf das erste Zeichen eines Kommandozeilenarguments zeigt.

Da zur Übersetzungszeit noch nicht feststehen kann, wie viele Argumente ein Benutzer später tatsächlich einmal übergeben wird, stellt C diese Information in einer eigenen Variable namens `argc` zur Verfügung. Trotzdem gibt es noch die Konvention, dass die Sequenz der `char *` mit einem `NULL`-Zeiger abgeschlossen wird. Doppelt hält besser ;)

Konstanten

Mit `const` kann man Objekte zur Übersetzungszeit vor Veränderung schützen. Bei Zeigern muss man den Zeiger und das referenzierte Objekt voneinander unterscheiden:

```
void f(const int * a, int * const b, const int * const c)
{
    ++a;     // a selbst ist nicht konstant
    ++*a;    // wohl aber die Variable, auf die a zeigt

    ++b;     // b selbst ist konstant
    ++*b;    // aber nicht die Variable, auf die b zeigt

    ++c;     // c selbst ist konstant
    ++*c;    // und auch die Variable, auf die c zeigt
}
```

Strukturen

Eine Struktur fasst beliebig viele bestehende Typen zu einem neuen Typ zusammen:

```
struct Person
{
    char nachname[30];
    char vorname[30];
    int alter;
};
```

Zur Initialisierung eines Objekts einer solchen Struktur gibt es eine spezielle Syntax:

```
struct Person max = {"Mustermann", "Max", 25};
```

Auf die einzelnen Attribute einer Struktur kann man mit der Punktnotation zugreifen:

```
printf("%s %s: %d\n", max.vorname, max.nachname, max.alter);
```

Strukturen sind first-class-citizens, d.h. man kann sie einander zuweisen, an Funktionen übergeben und aus Funktionen zurückliefern. Dabei wird eine (flache) Kopie erstellt.

```
struct Person moritz = max;
strcpy(moritz.vorname, "Moritz");
printf("%s und %s\n", max.vorname, moritz.vorname);
```

Möchte man in Funktionen auf dem Original statt auf einer Kopie arbeiten, etwa weil man dieses verändern möchte, so muss man wie gewohnt einen Zeiger übergeben:

```
void person__feiere_geburtstag(struct Person * self)
{
    (*self).alter++;
}
```

Aufrufen kann man eine solche Funktion dann wie folgt:

```
person__feiere_geburtstag(&max);
```

Da man häufig indirekt auf Attribute zugreift, gibt es dafür eine spezielle Syntax:

```
self->alter++;
```

Falls eine Funktion keine Veränderungen an einem Objekt vornehmen muss, kann man entweder das Objekt direkt übergeben (und somit auf einer Kopie arbeiten) oder einen Zeiger auf ein konstantes Objekt übergeben (und somit indirekt auf dem Original arbeiten). Im ersten Fall ist das Kopieren des Objekts aufwändiger, im zweiten Fall der Zugriff auf die Attribute, da jedes Mal ein zusätzlicher Dereferenzierungsschritt nötig ist.

Für komplexere Strukturen, bei denen einige Attribute implizite Startzustände haben, teilweise voneinander abhängen oder vorher validiert werden müssen, sollte man Funktionen bereitstellen, die eine sinnvolle Initialisierung durchführen („Konstruktoren“):

```
void Bruch__init(struct Bruch * self, int zaehler, int nenner)
{
    if (nenner < 0) { zaehler = -zaehler; nenner = -nenner; }
    int t = ggt(zaehler, nenner);

    self->zaehler = zaehler / t;
    self->nenner = nenner / t;
}
```

Funktionszeiger

Zu jedem Funktionstyp gibt es einen weiteren Typ „Zeiger auf eine Funktion dieses Typs“. Funktionszeiger dienen der Parametrisierung von Algorithmen mit Funktionen. So kann die Entscheidung, welche Funktion aufzurufen ist, in die Laufzeit verlagert werden.

```
int    f(int, int);   // f ist eine Funktion, die
                      // zwei ints entgegennimmt und int liefert
int (*p)(int, int);   // p ist ein Zeiger auf eine Funktion, die
                      // zwei ints entgegennimmt und int liefert
int x, y;
x = f(1, 2);          // rufe f direkt auf
p = &f;               // p zeigt auf f
y = (*p)(1, 2);       // rufe f indirekt über p auf
```

Die explizite Referenzierug und Dereferenzierung ist dabei optional:

```
p = &f;               // p zeigt auf f
y = (*p)(1, 2);       // rufe f indirekt über p auf
```

Zeiger auf selbstgeschriebene Funktionen kommen häufig im Zusammenhang mit Algorithmen aus der Standardbibliothek vor:

```
int ascending(const void * v, const void * w)
{
    const int * p = v;
    const int * q = w;
    int i = *p;
    int j = *q;
    return (i > j) - (i < j);
}

int primes[] = {11, 5, 3, 7, 19, 13, 2, 17};
qsort(a, 8, sizeof(int), ascending);
```

Polymorphie

Strukturen und Funktionszeiger lassen sich elegant kombinieren, um in einem Objekt neben den Daten auch Verhalten zu speichern. Damit kommt man schon nahe an den Objektbegriff aus der „Objektorientierten Programmierung“.

```
struct Player
{
    int (* noch_eine)(const struct Player *);
    int punkte, asse, siege;
};
```

Die Entscheidung, welche Funktion aufzurufen ist, wird erst zur Laufzeit gefällt:

```
while (p->punkte < 21 && p->noch_eine(p) && ziehe_karte(p)) {}
```

Auf diese Weise kann die eigentliche Spiellogik für den menschlichen Spieler und den Computerspieler zusammengefasst werden – lediglich die Bestimmung, ob noch eine Karte gezogen werden soll, ist separat zu programmieren und als Funktionszeiger im Objekt zu hinterlegen. Da Funktionszeiger zur Laufzeit geändert werden können, ist dieses Idiom sogar flexibler als der (statische) Vererbungsmechanismus aus der OOP.