4.11 Datenstrukturen/Rückgabe von Arrays durch Funktionen

4.11.1 Array als Rückgabewert

In einem größeren Programm soll eine eigenständige Funktion eingerichtet werden, die die Programmversion (z.B. `"2.40 XYZ Professional"`) als String zurückgeben soll. Hier ist eine solche Funktion, eingebettet in ein Testprogramm:

```
1  #include <stdio.h>
2  #include <string.h>
3  /**************************/
4  char getversion(void)[80]
5  {
6     char v[80];
7     strcpy(v, "Version 4.10 DE");
8     return v;
9  }
10 /**************************/
11 int main(void)
12 {
13    char *ver;
14    ver = getversion();
15    printf("%s\n", ver);
16    return 0;
17 }
```

Laut Funktionskopf ist `getversion()` eine Funktion (Parameterliste `void`) mit Rückgabetyp Array (80 Elemente) von `char`. Der Compileraufruf ergibt:

Terminal

```
schueler@debian964:~$ gcc getversion1.c
getversion1.c:4:6: error: 'getversion' declared as function returning an
array
getversion1.c: In function 'getversion':
getversion1.c:8:4: warning: return makes integer from pointer without a cast
[enabled by default]
getversion1.c:8:4: warning: function returns address of local variable
[enabled by default]
getversion1.c: In function 'main':
getversion1.c:14:8: warning: assignment makes pointer from integer without a
cast [enabled by default]
```

Nach Ansicht des Compilers über Zeile 4 darf eine Funktion kein Array zurückgeben. Der Compiler hat Recht: **In C ist für Funktionen der Rückgabetyp Array nicht erlaubt.**

4.11.2 Zeiger (auf die Startadresse) als Rückgabewert

Für einen zweiten Versuch kann man sich überlegen: In Zeile 8 gibt man ja den Namen des Arrays an – vielleicht gibt man ja hier „nur“ einen Zeiger zurück. Und dann kann man vielleicht ja schon im Funktionskopf als Rückgabetyp einen Zeiger auf `char` verwenden?

Das zweite Programm macht genau das:

```
1  #include <stdio.h>
2  #include <string.h>
3  /**************************/
4  char* getversion(void)
5  {
```

```
6     char v[80];
7     strcpy(v,"Version 4.10DE");
8     return v;
9  }
10 /*************************/
11 int main(void)
12 {
13    char *ver;
14    ver = getversion();
15    printf("%s\n", ver);
16    return 0;
17 }
```

`getversion()` ist jetzt eine Funktion (Parameterliste `void`) mit Rückgabetyp Adresse von `char`. Der Compileraufruf ergibt jetzt:

Terminal

```
schueler@debian964:~$ gcc getversion2.c
gcc getversion2.c
getversion2.c: In function 'getversion':
getversion2.c:8:4: warning: function returns address of local variable [enabled by d
```

Das Programm ist also C-gemäß. Jetzt muss es nur noch starten:

Terminal

```
schueler@debian964:~$ ./a.out
$xc%%w_#!u&ml
```

Das Programm gibt also Unsinn aus. Verwendet man einen anderen Compiler, kann dies passieren:

Terminal

```
schueler@debian964:~$ ./a.out
Speicherzugriffsfehler
```

4.11.3 Ursachenforschung

Warum funktioniert das Programm nicht so, wie es soll? In einem ersten Lösungsversuch fügen wir die Anweisung printf("%s\n", v); zwischen die Zeilen 7 und 8 ein.

Terminal

```
schueler@debian964:~$ gcc getversion2a.c
gcc getversion2a.c
getversion2a.c: In function 'getversion':
getversion2a.c:9:4: warning: function returns address of local variable
[enabled by default]
schueler@debian964:~$ ./a.out
Version 4.10DE
Verssxl~4.10DE
```

In der Funktion `getversion()` selbst ist also alles in Ordnung, aber in `main()` ist immer noch ein Problem.

In einem zweiten Lösungsversuch wird `version` global statt lokal deklariert:

Terminal

```
schueler@debian964:~$ gcc getversion2b.c
schueler@debian964:~$ ./a.out
Version 4.10DE
Version 4.10DE
```

Jetzt funktioniert also alles. Globale Variablen sollten aber nicht benutzt werden, da sind sich nahezu alle Experten einig. Aber woran liegt es, dass das Original-Programm nicht funktioniert?

Dazum muss man sich überlegen, worin sich globale und lokale Variablen unterscheiden: Es sind ihre Sichtbarkeit und ihre Lebensdauer. Globale Variablen sind im ganzen Programm sichtbar, und ihre Lebensdauer ist so lang wie die Programmdauer.

Wenn die lokale Variable `v` nicht genügend sichtbar wäre, dann könnte der Quellcode nicht compiliert werden. Diese Möglichkeit scheidet damit aus. Wenn Variable nicht genügend lange lebte, dann könnte ihr Inhalt überschrieben werden. Genau das scheint hier aber passiert zu sein. Also: Es hat mit der Lebensdauer zu tun, weil der Variablen-*Inhalt* falsch ist.

Die Lebensdauer des Arrays `v` ist nach Definition so groß wie die Zeit, in der das Programm innerhalb des Blockes abläuft, in dem `v` definiert wurde. `v` wird innerhalb der Funktion `getversion()` definiert, also lebt `v` genau so lange wie diese Funktion.

Damit wird klar: Das Array `v` wird genau in *dem* Moment überschrieben, wenn die Funktion beendet wird. Der Zeiger selbst ist zwar noch gültig, er lebt noch, aber das Objekt, auf das er zeigt, ist in dem Moment schon nicht mehr gültig.

4.11.4 Abhilfen: `static`-Variable

Was kann man tun?

a) Man kann eine globale Variable benutzen (`getversion3.c`). Die globale Variable ist jedoch überall sichtbar, sie kann von jedem Programmteil aus verändert werden. Deshalb ist diese Lösung ungünstig.

b) Man kann In-Place-Substitution benutzen (`getversion4.c`). Das hat Vorteile, ist aber auch umständlich:

- Die aufrufende Funktion muss die Länge des Puffers angeben (2. Parameter)
- Die Funktion `getversion()` muss zurückmelden, ob der Puffer groß genug war.
- Die aufrufende Funktion muss darauf wiederum reagieren.
- Oder die aufgerufene Funktion könnte vor dem richtigen Aufruf zurückgeben, wie groß der Puffer sein müsste (vielleicht als Rückgabewert bei Übergabe eines NULL-Zeigers).
- Oder aber die Funktion `getversion()` ist so dokumentiert, dass man weiß, wie groß der Puffer sein muss.
- Ein Beispiel dafür, wie umständlich die In-Place-Substitution in so einem Fall sein kann, ist die Funktion `fgets()`.

c) Man kann eine Variable mit erhöhter Lebensdauer einrichten, eine so genannte `static`-Variable (`getversion5.c`).

```
1  #include <stdio.h>
2  #include <string.h>
3  /**************************/
4  char* getversion(void)
5  {
6      static char v[80]; /* Lebensdauer=Programmdauer */
7      strcpy(v,"Version 4.10DE");
8      return v;
9  }
10 /**************************/
11 int main(void)
12 {
13     char *ver;
14     ver = getversion();
```

```
15      printf("%s\n", ver);
16      return 0;
17  }
```

Merksatz: **Wenn man in einer Funktion eine Variable mit dem Schlüsselwort static kennzeichnet, erhält sie die Lebensdauer des Programms.**

d) Auch die Verwendung dynamischen Speichers (`getversion6.c`) ist möglich (siehe C6.3).

4.11.5 Folgen der `static`-Variable

Warum macht man eigentlich nicht alle Variablen zu `static`-Variablen? Haben `static`-Variablen auch Nachteile? Dazu ein Überblick:

a) Die Funktion erhält durch diese Variable ein eigenes Gedächtnis. Wenn man die Variable in der Funktion schreibt, dann gibt die Funktion nicht mehr bei jedem Aufruf das gleiche Ergebnis.

Das Verhalten der Funktion hängt damit von vorhergehenden Aufrufen ab, anstatt wie bisher nur von Parametern und globalen Variablen. Man kann die Funktion nun nicht mehr dadurch prüfen, dass man bestimmte Parameter übergibt.

b) Bisher wurde bei jedem Eintritt in die Funktion auf dem Stapelspeicher (*stack*) eine neue Instanz (Version) der Variablen angelegt. Das ist bei einer `static`-Variablen nicht der Fall, es gibt nur eine einzige Instanz, die zum Programmbeginn angelegt und initialisiert wird.

Damit kann ein Problem entstehen, wenn mehrere Funktionen die Variable gleichzeitig nutzen (und zwar als Variable!) möchten:

- Für die Funktion ist keine Rekursion möglich, die Funktion kann sich nicht selbst aufrufen, denn es kann immer nur eine Instanz der Funktion gleichzeitig aktiv sein.
- Auch eine indirekte Rekursion geht aus diesem Grund nicht.
- Die Funktion ist für *multithreading*, eine Art der Parallelprogrammierung, ebenfalls nicht geeignet.
- Die Funktion kann nicht in Funktions-Bibliotheken verwendet werden, die von mehreren Programmen gleichzeitig benutzt werden (*shared libraries* bzw. DLLs).
- Und auch wenn innerhalb eines Programms die Funktion nacheinander an verschiedenen Stellen benutzt wird, kann es Probleme geben, wenn sich das Programm auf die „alten“ Inhalte der Variablen verlassen muss.

  Ein Beispiel für dieses Problem liegt in der Standardbibliothek im Teil `<time.h>`: Die Funktionen `asctime()`, `ctime()`, `gmtime()` und `localtime()` geben Zeiger auf `static`-Variablen zurück. Der Inhalt dieser Variablen kann sich nun immer dann ändern, wenn eine dieser Funktionen aufgerufen wird.

  Ein anderes Beispiel aus der C-Standard-Bibliothek ist die Funktion `strtok()` (durch `<string.h>` eingebunden).

Funktionen mit dieser Eigenschaft heißen nicht-wiedereintrittsfähig (*non-reentrant*).

Man sollte deshalb `static`-Variablen so selten wie möglich benutzen. Wenn man sie doch benutzt, sollte man das unbedingt für jede einzelne betroffene Funktion dokumentieren.