8.2 Sonstiges/Lokale und globale Sprünge

8.2.1 Kleine Sprünge mit `goto`

In C gibt es eine Sprunganweisung, die allerdings recht selten verwendet wird. An jeder Stelle einer Funktion kann man ein Sprungziel setzen, gekennzeichnet durch einen Namen und einen Doppelpunkt; an jeder anderen Stelle derselben Funktion kann man mit der `goto`-Anweisung zu diesem Ziel springen. In `src/goto.c` sieht man ein Beispiel.

```c
#include <stdio.h>

int main(void)
{
   int lauf=0;
oben:
   ++lauf;
   printf("%d\n", lauf);
   if(lauf<10)
   {
      goto oben;
   }
   return 0;
}
```

Die Sprunganweisung ist – sehr sparsam verwendet – an einigen Stellen sinnvoll. Ausgiebige Verwendung dieser Anweisung führt aber zu einem schwer verständlichen und extrem unübersichtlichen *Spaghetti-Code*. Zur Überzeugung sollen die Programme `zufall1000a.c` und `zufall1000b.c` dienen. Der erste Quelltext ist wie üblich geschrieben:

```c
#include <stdio.h>
#include <time.h>
#define GROESSE 1000
int main(void)
{
   int arr[GROESSE];
   int max, lauf;
   srand(time(0));

   for(lauf=0; lauf<GROESSE; ++lauf)
      arr[lauf]=rand();

   max=arr[0];
   for(lauf=1; lauf<GROESSE; ++lauf)
      if(max<arr[lauf])
         max=arr[lauf];

   printf("Groesste von 1000 Zufallszahlen: %i\n", max);
   return 0;
}
```

der zweite dagegen wurde mit dem Programm `cunloop` aus dem Debian- oder Ubuntu-Paket `cutils` erstellt:

Terminal

```
schueler@debian964:~$ cunloop < zufall1000a.c > zufall1000b.c
```

```c
#include <stdio.h>
#include <time.h>
#define GROESSE 1000
int
main (void)
{
  int arr[GROESSE];
  int max, lauf;
  srand (time (0));
  {
    lauf = 0;
  l_1:
    if (!(lauf < GROESSE))
      goto l_3;
    goto l_4;
  l_2:++lauf;
    goto l_1;
  l_4:
    arr[lauf] = rand ();
    goto l_2;
  l_3:;
  }
  max = arr[0];
  {
    lauf = 1;
  l_5:
    if (!(lauf < GROESSE))
      goto l_7;
    goto l_8;
  l_6:++lauf;
    goto l_5;
  l_8:
    {
      if (!(max < arr[lauf]))
        goto l_9;
      max = arr[lauf];
    l_9:;
    }
    goto l_6;
  l_7:;
  }
  printf ("Groesste von 1000 Zufallszahlen: %i\n", max);
  return 0;
}
```

8.2.2 Große Sprünge mit `longjmp()`

Die `goto`-Anweisung in C kann nicht verwendet werden, um aus einer tief verschachtelten Kette von Funktionen herauszuspringen: Der Compiler verweigert einen solchen Sprung. Grund dafür ist die Tatsache, dass man mit einem solchen Sprung den Stack und damit den Programmzustand in Unordnung bringt.

Abhilfe bietet ein zweiteiliges Werkzeug, das mit der Headerdatei `<setjmp.h>` eingebunden

werden kann:

- `setjmp(x)` ist eine Pseudofunktion, die den aktuellen Programmzustand in der Variablen x (vom Typ `jmp_buf`) speichert. Das entspricht dem Setzen einer Sprungmarke für die `goto`-Anweisung.
- Mit `longjmp(x,rc)` stellt man den Programmzustand wieder her, der vorher mit `setjmp(x)` gespeichert worden ist. Das entspricht der `goto`-Anweisung selbst.

Ein Beispiel zeigt das Programm `src/longjmp1.c`:

```
1  #include <stdio.h>
2  #include <setjmp.h>
3  jmp_buf status;
4  /*****************************/
5  void f3(void)
6  {
7      printf("hier ist f3\n");
8      longjmp(status, 7);
9  }
10 /*****************************/
11 void f2(void)
12 {printf("hier ist f2\n");f3();}
13 /*****************************/
14 void f1(void)
15 {printf("hier ist f1\n");f2();}
16 /*****************************/
17 int main(void)
18 {
19     int rc, lauf=0;
20     printf("vor setjmp()\n");
21 /*-----------------------------------*/
22     rc=setjmp(status);
23     if(rc==0)
24         printf("Sprungzustand gespeichert.\n");
25     else
26         printf("Fehler Nr. %d aufgetreten!\n",rc);
27     ++lauf;
28     printf("Durchlauf: %d\n", lauf);
29     sleep(1);
30     f1();
31 /*-----------------------------------*/
32     printf("nach f1()\n");
33     return 0;
34 }
```

Eigentlich sieht das Programm so aus: `main()` ruft `f1()` auf, `f1()` ruft `f2()` auf und `f2()` ruft `f3()` auf (siehe Abbildung 1).

Aber nun wird in Zeile 22 der Programmzustand in der globalen Variablen `status` gesichert. Die Funktion `setjmp()` gibt dabei den Wert null zurück. Die anschließende `if`-Anweisung verzweigt daher in den oberen Zweig. Anschließend wird in Zeile 30 `f1()` aufgerufen, in Zeile 15 `f2()` und in Zeile 13 `f3()`. In Zeile 8 folgt schließlich ein Rücksprung ins Hauptprogramm. Als erster Parameter wird `status` angegeben, also wird der Zustand hergestellt, der beim Aufruf von `setjmp()` in Zeile 22 gespeichert worden war. Als zweiter Parameter wird hier willkürlich eine 7 angegeben.

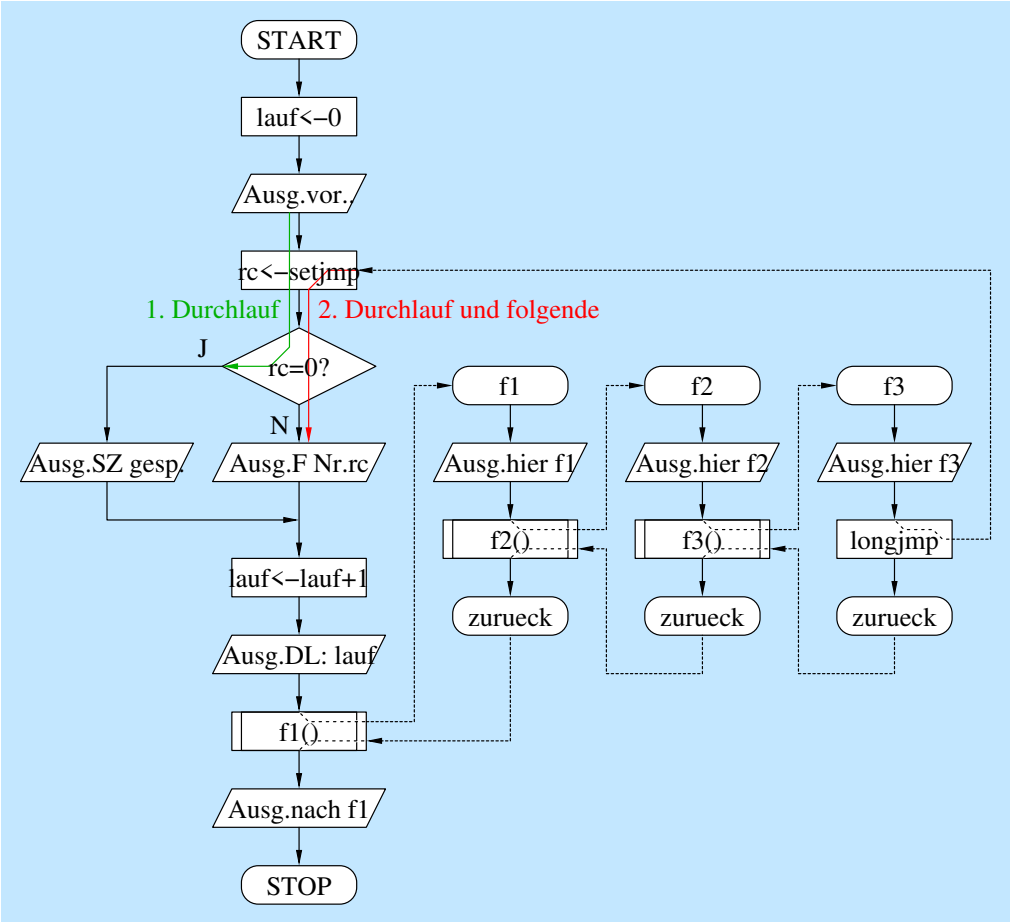

Abbildung 1: PAP zu `longjmp1.c`

Das Programm geht nun so weiter, als ob `setjmp` aufgerufen worden wäre und eine 7 zurückgegeben hätte. Nun – bei diesem zweiten Mal – verzweigt die anschließende `if`-Anweisung in den `else`-Zweig, der die Ausgabe einer Fehlermeldung bewirkt. Durch die Sprunganweisung ist also in diesem Fall eine Schleife entstanden, die von Zeile 21 bis 31 reicht, sich aber andererseits durch vier verschiedene Funktionen zieht. Vorsicht wäre geboten bezüglich des Inhalts von Variablen in `f1()`, `f2()`, `f3()`, die zwischen `setjmp()` und `longjmp()` verändert wurden; in diesem Beispiel gibt es aber keine solchen Variablen.

Im Programm `src/longjmp2.c` ist ein ähnliches Beispiel, bei dem aber *keine* Schleife entsteht, weil der Aufruf von `f1()` in den ersten Zweig der `if`-Anweisung gelegt wird.