Querschnitte durch C

Franz Fiala, EN, TGM

Dieser Kurzbeitrag ist im Rahmen des C/C++-Seminars des PCC-TGM entstanden und beschreibt die verschiedenartige Bedeutung der Schlüsselwörter `const` und `static`. Nicht nur, daß den Autoren von C eine schreibökonomische Ausdrucksweise vorschwebte, auch verwenden sie dieselben Schlüsselwörter in einem zwar ähnlichen Zusammenhang, der aber doch - je nach Stellung - etwas anderes ausdrückt, sodaß es für Uneingeweihte zu einer unklaren Interpretation kommen kann.

const

(1) const zur Kennzeichnung unveränderlicher Variablen

```
const int SIZE = 5;
```

`SIZE` kann in der Folge wie eine Variable verwendet werden und wird auch beim Debuggen als Bezeichner erkannt. Bisher hat man in C dafür eine `#define`-Präprozessor-Anweisung verwendet.

```
#define SIZE 5
```

Der Nachteil: `SIZE` hat keinen Typ, es wird nur die Buchstabenkombination `SIZE` durch 5 ersetzt. Es kann keinerlei Typenprüfung erfolgen. `SIZE` ist beim Debuggen nicht sichtbar und wird durch 5 ersetzt.

(2) const zur Bezeichnung unveränderlicher Pointer

```
char *const ptr = mybuf;
```

Diese Zeile besagt, daß `ptr` nicht verändert werden kann, d.h. der Ausdruck `prt=newbuf;` vom Compiler als unerlaubt eingestuft wird. Dagegen ist `*ptr='A';` erlaubt.

(3) const zur Bezeichnung eines Pointers auf eine Konstante

```
const char *ptr = mybuf;
```

Diese Zeile besagt, daß `ptr` auf die Speicherstelle `mybuf` zeigt, die nicht verändert werden darf. In diesem Fall ist der Ausdruck `ptr=newbuf;` erlaubt, dabei zeigt jetzt der Pointer `ptr` auf einen neuen Puffer `newbuf`, der ebenso wie `mybuf` konstant sein muß. Dagegen ist `*ptr='A';` nicht erlaubt, da dadurch der Puffer verändert werden würde.

(4) const bei Parameterübergabe

Bei der Parameterübergabe bei Funktionen dient `const` als Hinweis, daß eine Funktion einen übergebenen Parameter nicht verändern darf.

```
void fkt(const char a, char b);
```

In diesem Prototyp wird ausgedrückt, daß die Funktion den Parameter `a` nicht verändern darf, den Parameter `b` aber schon. Das ist aber bei dieser Übergabe als Wert nicht wichtig, da `a` und `b` lokale Parameter sind und ohnehin keinerlei Rückwirkung auf die rufende Funktion haben. Bedeutender wird `const` im folgenden Fall:

```
void fkt(char * const a, char *b);
```

In diesem Beispiel werden zwei Pointer übergeben; die Daten auf die `a` zeigt können durch die Funktion nicht verändert werden; die Daten, auf die `b` zeigt können verändert werden.

Ebenso wirkt der Bezeichner const im Zusammenhang mit Referenzen.

```
void fkt(const char &a, char &b);
```

Eine neue Dimension erhält `const` im Zusammenhang mit Klassen:

(5) const im Zusammenhang mit Objekten

Ebenso wie bei Variablen kann `const` auch bei Objekten angewendet werden. Eine Klasse `DATUM` kann folgendermaßen angewendet werden:

```
const DATUM Geburtstag (12,11,48);
```

Die Werte von `Geburtstag` können nicht geändert werden.

Hier kommt es aber zu einer zusätzlichen Schwierigkeit: Elementfunktionen von konstanten Objekten können nicht gelesen werden! Warum das, wird man sich fragen? Der Compiler kann bei konstanten Variablen alle Operationen identifizieren, die ihre Veränderung bewirken; er kann aber nicht wissen, welche der Elementfunktionen Veränderungen an den Daten des Objekts vornehmen. Daher verlegt er sich auf die sichere Seite und erlaubt gar keinen Zugriff auf Elementfunktionen. Um dennoch mit konstanten Objekten umgehen zu können, wird der Bezeichner zur Kennzeichnung jener Elementfunktionen verwendet, deren Verwendung auch bei konstanten Objekten sicher ist.

```
class DATUM
{
public:
  DATUM(int t, int m, int j);
  int LTag() const;
  int LMonat() const;
  int LJahr() const;
  void STag(int t);
  void SMonat(int m);
  void SJahr(int j);
private:
  int Tag, Monat, Jahr;
};
```

In diesem Beispiel können bei konstanten Objekten die mit `const` bezeichneten Funktionen `LTag, LMonat, LJahr` zu Lesen der Daten verwendet werden, dagegen können die Funktionen `STag, SMonat` und `SJahr` nicht angewendet werden.

Bei nicht-konstanten Objekten ist der Bezeichner `const` ohne Bedeutung. Der Bezeichner `const` muß gleichzeitig in der Deklaration und auch in der Definition angewendet werden. Würde eine der mit `const` gekennzeichneten Elementfunktionen eine private Variable verändern wollen, würde dies der Compiler ebenso ahnden, wie wenn ein konstantes Objekt dieser Klasse versuchen würde, eine nicht konstanten Elementfunktion aufzurufen.

Regel: man sollte den Bezeichner `const` wo immer anwendbar verwenden, um dem Compiler die Möglichkeit zu geben, Flüchtigkeitsfehler beim Programmieren zu verhindern.

static

Es gibt einen weiteren Bezeichner, nämlich `static`, der in unterschiedlichen Stellungen unterschiedliche Wirkung zeigt:

(1) static global

Der Bezeichner `static` drückt im Zusammenhang mit globalen Variablen (Variablen außerhalb von Funktionen) oder Funktionen aus, daß der Gültigkeitsbereich dieser Variablen oder dieser Funktion auf den aktuellen Modul(Datei) beschränkt ist und der Name der Variablen oder der Funktion nicht an den Linker weitergegeben wird. Das ist eine einfache Form von Kapselung einer Variablen oder einer Funktion in C und Schutz vor versehentlichem Zugriff in einem anderen Modul.

```
DATEI1                    DATEI2
static int i;             static int i;
static void func(void);   static void func(void);
```

Die beiden `i` und die beiden `func()` kommen einander auch nach dem Binden der beiden Module nicht in die Quere.

(2) static innerhalb von Funktionen

Der Bezeichner `static` innerhalb von Funktionen bewirkt, daß die so gekennzeichnete Variable die Lebensdauer des Programms bekommt und nicht automatisch nach Ablauf der Funktion wieder verschwindet.

Beispiel:

Zählen, wie oft eine Funktion aufgerufen wurde:

```
                void fkt(void)      void fkt(void)
                {                   {
                  static count=0;     count=0;
                  count++;            count++;
                }                   }

                count in fkt()      count in fkt() ohne static
                  0                   ?

void main(void)
{
  fkt();          1                 1 ?
  fkt();          2                 1 ?
}
```

Die lokale Variable `count` wird ohne den Bezeichner `static` immer wieder auf den Wert 0 initialisiert, daher hat `count` nach `count++;` immer den Wert 1.

Hat die Variable `count` den Bezeichner `static`, wird bei Programmbeginn `count` auf 0 gesetzt. `count` existiert also schon bevor die Funktion gerufen wurde. Jeder Aufruf von `fkt()` bewirkt die Erhöhung von `count`, so als wäre `count` eine globale Variable. Gegenüber einer globalen Variablen genießt `count` jedoch den Vorzug, nur innerhalb der Grenzen von `fkt()` zu gelten.

(3) static in Klassen

Jedes Objekt einer Klasse enthält normalerweise einen Satz aller Variablen dieser Klasse; es gibt also ebensoviele Variablen als es Objekte dieser Klasse gibt. Ist aber eine Variable als `static` deklariert, existiert diese Variable nur einmal, unabhängig von der Zahl der Objekte.

Ähnlich, wie schon bei `static` innerhalb von Funktionen, kann `static` innerhalb von Klassen dazu benutzt werden, zu zählen, wievele Objekte dieser Klasse existieren. Es sind zwei Fälle möglich:

Man kann sagen, daß eine mit `static` gekennzeichnete Variable innerhalb einer Klasse etwa so wirkt, wie eine gewöhnliche globale Variable, die aber den eingeschränkten Geltungsbereich der Klasse besitzt inklusive der Zugriffsbeschränkungen `public` oder `private`.

(3a) static als öffentliches Datum

```
class MYCLASS
{
public:
  static int count;
};
```

In diesem Fall kann die Variable `count` von aussen initialisiert werden:

```
  MYCLASS m;
  m.count = 0;
```

Diese Schreibweise ist allerdings zu vermeiden, erweckt man doch den Eindruck, `count` wäre eine Variable von `m` allein. Besser ist es, den Klassennamen zu Kennzeichnung zu verwenden, um anzuzeigen, daß es sich um eine Variable handelt, die allen Objekten dieser Klasse gemeinsam ist:

```
  MYCLASS::count = 0;
```

Man kann eine Variable mit dem Bezeichner `static` nicht durch einen Konstruktor initialisieren, wie denn auch, es werden ja viele Konstruktoren gerufen aber eine Variable kann nur einmal initialisiert werden.

(3b) static als privates Datum

```
class MYCLASS
{
privat:
  static int count;
};
```

Private Variablen mit dem Bezeichner `static` werden außerhalb von Funktionen und Klassen wie eine globale Variable mit voller Bezeichnung initialisiert:

```
int MYCLASS::count = 0;
```

Für diesen Sonderfall ist der Zugriff auf eine private Variable einer Klasse erlaubt. Es ist zu beachten, daß diese Zeile die Variable deklariert und initialisiert und nicht die entsprechende Zeile innerhalb der Klasse.

(3c) static als Bezeichner für eine Funktion

Eine Funktion einer Klasse, die lediglich die mit `static` bezeichneten Variablen bearbeitet kann ebenso den Bezeichner `static` erhalten. Das hat zur Folge, daß diese Funktion auch wirklich nur die Variablen mit `static` erreichen kann, nicht die anderen. Während alle anderen Funktionen einen `this`-Pointer besitzen, besitzen `static`-Funktionen keinen.

Folgendes Beispiel zeigt eine Klasse `BALL`, die zählt, wieviele Bälle kreiert wurden:

```
class BALL
{
public:
             BALL() { count++; }
             ~BALL() { count--; }
  static int WieViele() { return count; }
private:
  static int count;
};

int BALL::count = 0;
```

Die Funktion `WieViele()` gibt Aufschluß über die Anzahl exisitierender Bälle. □