`skeleton.c` — Ein Grundgerüst eines Kommandozeilenprogramms

Die Datei `skeleton.c` enthält ein Grundgerüst eines Kommandozeilenprogramms, wie es für eine Vielzahl von Anwendungsprogrammen verwendet werden kann. Es zeigt die Auswertung der Kommandozeile, wobei zwischen festen Programmparametern, die angegeben werden müssen, und optionalen Parametern (auch Optionen oder Schalter genannt), die angegeben werden können, unterschieden wird. Außerdem enthält es eine Fehlerfunktion, die bei Auftreten eines nicht behebbaren Fehlers aufgerufen wird. Diese Funktion gibt eine Fehlermeldung aus und beendet das Programm. Im folgenden wird die Funktionsweise des Programms `skeleton.c` im einzelnen beschrieben.

Allgemeine Programmstruktur

Die allgemeine Struktur des Programms, die sich für beliebige Programme und Programmodule als sinnvoll herausgestellt hat, ist die folgende:

- Einbinden der benötigten Header-Dateien (`#include`)
- Präprozessordefinitionen (`#define`)
- Typdefinitionen (`typedef`, fehlen in diesem Programm)
- Konstantendefinitionen (als `const` definierte globale Bezeichner)
- Globale Variablen (nicht als `const` definierte globale Bezeichner)
- Funktionen

Wir gehen diese Struktur von oben nach unten durch.

Einbinden der Header-Dateien

Gleich am Anfang des Programms finden wir bereits eine Ausnahme von dem oben angegebenen Schema, da eine Präprozessordefinition vor dem Einbinden der Header-Dateien steht.

```
#define _POSIX_SOURCE
```

Dies hat jedoch einen besonderen Grund. POSIX steht für das „IEEE Portable Operating System Interface for Computing Environments“. POSIX ist ist ein Standard, der die Betriebssystemaufrufe in C-Programmen vereinheitlichen soll. Er definiert eine bestimmte Menge von Header-Dateien, Funktionen, Konstantendefinitionen, etc., die verfügbar sein müssen. In fast jedem Betriebssystem gibt es aber auch Funktionen, Datentypen, Konstanten, etc., die nur von diesem speziellen Betriebssystem angeboten bzw. in dieser Form angeboten werden. Wenn man solche Funktionen benutzt, ist das sich ergebende Programm i.a. an das Betriebssystem gebunden, für das es entwickelt wurde. Es läuft dann z.B. nur auf HP-Rechnern, oder nur auf Sun, usw. Will man ein Programm schreiben, das im Prinzip auf jedem System übersetzt und ausgeführt werden kann, so ist man gut beraten, sich auf eine Menge von Standardfunktionen zu beschränken — eben auf die vom POSIX-Standard (und vom ANSI-C-Standard) festgelegten Funktionen.

Um sicherzugehen, daß man auch wirklich nur POSIX-Funktionen und keine systemspezifischen Funktionen verwendet hat, wird vom C-Compiler folgende Hilfe angeboten: Wird die Präprozessorkonstante `_POSIX_SOURCE` definiert, bevor eine der Standard-Header-Dateien eingebunden wird, so werden in diesen Header-Dateien nur genau die Funktionen, Typen und Konstanten definiert, die dem POSIX-Standard entsprechen. Auf diese Weise erhält man vom Compiler auf jeden Fall eine Warnung, wenn man z.B. eine nicht portable Funktion verwendet. Da dies nur geschieht, wenn die Definition von `_POSIX_SOURCE` vor dem Einbinden der Header-Dateien steht, müssen wir in diesem Fall von dem oben angegebenen allgemeinen Schema abweichen.

Die Definition von `_POSIX_SOURCE` ist der Angabe des Compilerschalters `-ansi` (Gnu-C-Compiler) verwandt. Durch Angabe des Schalters `-ansi` wird die Konstante `__STDC__` definiert, die bewirkt, daß in den Header-Dateien nur die Funktionen, Konstanten, etc. definiert werden, die zum ANSI-C-Standard gehören. Wird sowohl `-ansi` angegeben, als auch `_POSIX_SOURCE` definiert, stehen sowohl die Funktionen des ANSI-C-Standards als auch die Funktionen des POSIX-Standard zur Verfügung. Zwar spielen die Funktionen des POSIX-Standards, die über den ANSI-C-Standard hinausgehen, für das vorliegende Programm keine Rolle, aber es empfiehlt sich, die Definition von `_POSIX_SOURCE` in ein allgemeines Programmgerüst aufzunehmen. Erstens hat man sie dann, wenn sie benötigt wird, schon zur Verfügung, zweitens — und das ist der wichtigere Grund — geht man so sicher, daß man keine Konstanten und Funktionen mit Namen definiert, die im POSIX-Standard verwendet werden (in einem solchen Fall gäbe der Compiler eine Warnung aus). Letzteres ist besonders bei Programmodulen wichtig, die ja u.U. in Programme eingebunden werden, die POSIX-Funktionen benutzen.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
```

Zu der Header-Datei `stdio.h` muß wohl nichts gesagt werden. Die Header-Datei `stdlib.h` ist wegen der im Programm verwendeten Funktionen `exit`, `fflush`, `strtol` und `strtod` notwendig. Die Header-Datei `stdarg.h` enthält die Definitionen jener Funktionen, die in der Fehlerfunktion `error` benutzt werden, um eine variable Anzahl von Funktionsargumenten auszuwerten. Diese Funktionen werden weiter unten besprochen.

Präprozessordefinitionen

Nach dem Einbinden der Header-Dateien werden die programmspezifischen Präprozessorkonstanten definiert. In diesem Programm sind es nur einige Fehlernummern, die bei einem Aufruf der Funktion `error` verwendet werden.

```
/* --- error codes --- */
#define OK            0        /* no error */
#define E_NONE        0        /* no error */
#define E_NOMEM     (-1)       /* not enough memory */
#define E_FOPEN     (-2)       /* file open failed */
#define E_FREAD     (-3)       /* file read failed */
#define E_FWRITE    (-4)       /* file write failed */
#define E_OPTION    (-5)       /* unknown option */
#define E_OPTARG    (-6)       /* missing option argument */
#define E_ARGCNT    (-7)       /* wrong number of arguments */
#define E_UNKNOWN   (-8)       /* unknown error */
```

Eine Fehlernummer wird gewöhnlich von einer Funktion zurückgeliefert, wenn sie die ihr zugedachte Aufgabe nicht erfolgreich ausführen konnte. Viele Funktionen der Standardbibliotheken halten sich dabei an folgendes Schema: Konnte die Funktion erfolgreich ausgeführt werden, so ist der Rückgabewert 0, bei einigen Funktionen auch ≥ 0, wobei Werte > 0 zusätzliche Informationen enthalten. Ist die Rückgabe dagegen ungleich 0 (im ersten Fall) bzw. < 0 (im zweiten Fall), so ist ein Fehler aufgetreten. Z.B. liefert die Funktion `fclose`, mit der eine geöffnete Datei geschlossen wird, im Erfolgsfall 0 und sonst einen Wert ungleich 0 zurück. Auch die Funktion `main`, die es ja in jedem C-Programm geben muß, sollte im Erfolgsfall 0 und im Fehlerfall einen Wert ungleich 0 zurückgeben. Die Funktionen `printf` und `fprintf` liefern im Erfolgsfall die Anzahl der geschriebenen Zeichen und sonst einen negativen Wert.

Es empfiehlt sich, dieses Schema überall dort, wo es sinnvoll verwendbar ist, zu übernehmen. Da durch positive Werte oft zusätzliche Informationen übermittelt werden (etwa bei `printf` die Zahl der geschriebenen Zeichen), werden alle Fehlernummern negativ gewählt. Die Präprozessordefinitionen dienen dazu, die Zuordnung der Fehlernummern zu deren Bedeutung zu erleichtern. Es fällt sicherlich schwer, sich zu merken, daß in diesem Programm der Wert -5 dafür steht, daß der Benutzer eine ungültige Option angegeben hat, während es wesentlich leichter ist, sich zu merken, daß der Wert `E_OPTION` diese Bedeutung hat. Das `E` am Anfang des Names soll darauf hinweisen, daß es sich um einen „error code“, eine Fehlernummer, handelt. Auch der Unterstrich ist wichtig, da Namen, die mit einem `E` beginnen, auf das dann ein Großbuchstabe oder eine Ziffer folgt, von POSIX als Bezeichner für Fehlernummern verwendet werden. Um Konflikte mit diesen Fehlernummern zu vermeiden, wird empfohlen, keine solchen Bezeichner zu definieren. Durch den Unterstrich halten sich die obigen Definitionen an diese Empfehlung.

Konstantendefinitionen

Der konstante Vektor `errmsgs` enthält zu jeder Fehlernummer einen Text, der von der Fehlerfunktion ausgegeben wird, wenn der zugehörige Fehler auftritt. Einige dieser Fehlermeldungen enthalten Formatanweisungen, d.h. in sie werden von der Fehlerfunktion bzw. der von ihr aufgerufenen `vfprintf`-Funktion zusätzliche Informationen über den Fehler eingefügt. So wird bei der Angabe einer unbekannten Option der Buchstabe dieser Option ausgegeben, bei einem Dateifehler der Name der Datei.

Die Zusammenfassung aller Fehlermeldungen in eine Variable hat zwei Vorteile: Erstens muß der Text einer Fehlermeldung nur einmal im Programm abgelegt werden, auch wenn der zugehörige Fehler an mehreren Stellen auftreten kann, und zweitens kann man so, falls nötig, leichter ein Programm an eine andere Landessprache (z.B. Französisch) anpassen. Um die Texte der Fehlermeldungen zu ändern, muß man dann nur die Einträge des Vektors `errmsgs` übersetzen.

Globale Variablen

In diesem Programm gibt es nur eine globale Variable, nämlich die Zeichenkette `prgname`, in der am Anfang der Funktion `main` der Name des Programms abgelegt wird. Dies dient dazu, in der Fehlerfunktion `error` neben der Fehlermeldung auch den Namen des Programms ausgeben zu können, das den Fehler verursachte bzw. feststellte. Dies ist unter UNIX deshalb wichtig, weil Programme oft nicht direkt von der Kommandozeile sondern aus einem sogenannten Shellscript aufgerufen werden. Werden in einem solchen Script mehrere Programme aufgerufen, kann man nicht erkennen, in welchem Programm der Fehler auftrat, es sei denn, die Fehlermeldung enthält den Namen des Programms.

Die Funktion `error`

In der Funktion `error` wird eine Fehlermeldung ausgegeben, die durch die übergebene Fehlernummer bestimmt ist. Danach wird das Programm beendet, d.h. aus der Funktion `error` wird nicht in die aufrufende Funktion zurückgekehrt.

Zuerst wird eine Prüfung der übergebenen Fehlernummer vorgenommen, um fehlerhafte Zugriffe auf den Vektor der Fehlermeldungstexte zu vermeiden und so in jedem Fall einen sinnvollen Fehlertext ausgeben zu können. Mit der eventuell korrigierten Fehlernummer wird der Text der zugehörigen Fehlermeldung aus dem konstanten Vektor `errmsgs` ausgelesen. Dann wird noch einmal geprüft, ob ein gültiger Fehlermeldungstext vorliegt, da es u.U. sinnvoll sein kann, in der Liste der Fehlernummern (und damit auch in dem Vektor `errmsgs`) Lücken zu lassen (die dann durch den Eintrag des Wertes `NULL` markiert werden könnten). Liegt entweder die Fehlernummer nicht im Bereich der definierten Fehlernummern oder verweist sie auf einen nicht vorhandenen Fehlertext, so wird die Fehlermeldung „unknown error“ gewählt. Das ist zwar nicht besonderes hilfreich, aber besser als gar keine Fehlermeldung.

Anschließend wird der Name des Programms ausgegeben (zu den Gründen siehe oben), worauf die Fehlermeldung folgt. Sowohl der Programmname als auch die Fehlermeldung werden nicht einfach mit `printf`, sondern mit `fprintf(stderr, ...)` ausgegeben, da der Ausgabekanal `stderr` speziell für Fehlermeldungen gedacht ist. Mit einem einfachen `printf` würde nur auf den normalen Ausgabekanal `stdout` geschrieben.

Da in die Fehlermeldungen ggf. zusätzliche Informationen eingefügt werden, müssen weitere (variable) Argumente der Funktion `error` geholt und der `fprintf`-Funktion übergeben werden. Dies geschieht mit Hilfe der Funktionen `va_start` und `va_end`, wobei das `va` stets „variable arguments“ bedeutet. `va_start` erhält als ersten Parameter eine Variable vom Typ `va_list`, in der ein Verweis auf die Liste der variablen Argumente abgelegt wird. Der zweite Parameter ist der Name des letzten festen Parameters der Funktion, da auf diesen die variablen Argumente folgen und er folglich den Anfang der Liste der variablen Argumente markiert. Die Auswertung der Liste der variablen Argumente, d.h. die Zerlegung in einzelne Argumente, braucht glücklicherweise nicht in der Funktion `error` zu geschehen, da mit `vfprintf` eine Standard-Bibliotheksfunktion zur Verfügung steht, die wie `fprintf` arbeitet und diese Auswertung übernimmt. Der einzige Unterschied zwischen `fprintf` und `vfprintf` besteht in der Angabe der Argumente zusätzlich zu dem Ausgabekanal und dem Formatstring. Bei `fprintf` werden die zusätzlichen Argumente hinter dem Formatstring aufgelistet, während `vfprintf` stattdessen eine Liste von Typ `va_list` erwartet.

Nachdem die Fehlermeldung ausgegeben wurde, muß nur noch die Auswertung der variablen Argumente abgeschlossen werden, was mit der Funktion `va_end` geschieht. Auf den meisten Systemen tut diese Funktion einfach nichts, so daß sie gefahrlos fehlen könnte, bei anderen Systemen kann ihr Fehlen jedoch zu schweren Fehlern führen. Man sollte daher bei der Verwendung der `va`-Funktionen darauf achten, die Auswertung der variablen Argumente mit `va_end` abzuschließen.

Nachdem die Fehlermeldung ausgegeben wurde, wird das Programm beendet. Falls das Programm jedoch noch Aufräumarbeiten vornehmen muß (z.B. temporäre Dateien löschen etc.), sollten diese vor dem Aufruf der Funktion `exit` ausgeführt werden. Der Funktion `exit` wird die Fehlernummer übergeben, um dem aufrufenden Programm das Auftreten und die Art des Fehlers mitzuteilen. Die Funktion `exit` kehrt nicht zurück, sondern beendet das Programm. Ein Aufruf `exit(n)` ist equivalent zu einem `return n;`, allerdings einem *in der Funktion* `main` stehenden.

Die Funktion `main`

Die Funktion `main` besteht aus drei Teilen: der Ausgabe einer Anfangsmeldung bzw. von Hinweisen zur Benutzung des Programms, der Auswertung der Kommandozeilenparameter und dem eigentlichen Anwendungsprogramm. Der dritte Teil, das eigentliche Anwendungsprogramm, ist in dem vorliegenden Programmgerüst, das keine echte Funktion hat, recht kurz. Er besteht lediglich aus der Ausgabe der über die Kommandozeile angegebenen optionalen und festen Programmparameter.

Wie bereits in den Erläuterungen zu der globalen Variable `prgname` bemerkt, wird zunächst der Programmname, der stets in `argv[0]` übergeben wird, in einer globalen Variable abgelegt, um in der Funktion `error` den Programmnamen zusätzlich zur Fehlermeldung ausgeben zu können. Anschließend wird abhängig davon, ob Kommandozeilenparameter angegeben wurden oder nicht, entweder eine Anfangsmeldung oder Hinweise zur Benutzung des Programms ausgegeben. D.h. dieses Skelett geht davon aus, daß das Anwendungsprogramm mindestens einen Parameter erhalten muß, um seine Funktion ausführen zu können. Ist dies nicht der Fall, ist die Bedingung entsprechend abzuändern. Ggf. sollten dann die Benutzungshinweise erst bei Angabe eines bestimmten Schalters oder bei Auftreten eines Fehlers in der Auswertung der Kommandozeilenparameter ausgegeben werden. Wichtig ist, daß die Anfangsmeldung nicht durch ein '`\n`' abgeschlossen wird, da in der Fehlerfunktion vor dem Fehlermeldungstext ein '`\n`' ausgegeben wird. Dadurch wird eine Leerzeile bei einem Fehler in der Auswertung der Kommandozeilenparameter vermieden. Erst bei fehlerfreier Ausführung wird die Anfangsmeldung durch ein '`\n`' abgeschlossen, und zwar vor dem Beginn des eigentlichen Anwendungsprogramms. Daß die Startmeldung auf jeden Fall sofort auf den Bildschirm geschrieben wird, wird durch einen Aufruf der Funktion `fflush` („file flush" — leere Dateipuffer) erzwungen.

Die Auswertung der Kommandozeilenparameter geschieht in einer `for`-Schleife, die alle Argumente außer dem nullten (dem Programmnamen) durchläuft. Wir überspringen zunächst die Abfrage der Variable `optarg`. Für jedes Argument wird geprüft, ob es mit einem '`-`' beginnt. Ein führendes '`-`' zeigt nach Konvention unter UNIX einen optionalen Parameter an. (Man sehe dazu auch die Handbuchseiten — „manual pages", vgl. den Befehl `man` — zu den Standard-UNIX-Befehlen.) Daher wird abhängig vom ersten Zeichen des Argumentes ('`-`' oder nicht) eine unterschiedliche Auswertung vorgenommen.

Nehmen wir zunächst an, daß der Kommandozeilenparameter nicht mit einem '`-`' beginnt, betrachten wir also den `else`-Zweig der `if`-Anweisung in der `for`-Schleife. Da das aktuelle Argument nicht mit einem '`-`' beginnt, handelt es sich um einen festen Parameter. In der Variable `k` werden die gefundenen festen Argumente gezählt. Durch eine Verzweigung gemäß dem Wert dieser Variable kann folglich das Argument an die passende Variable zugewiesen werden. Da das vorliegende Programmgerüst genau zwei feste Parameter erwartet, gibt es nur die Fälle `k = 0` und `k = 1`. Im ersten Fall wird die Variable `arg1`, im zweiten die Variable `arg2` gesetzt. Hat `k` einen Wert ≥ 2, so wird, im `default`-Teil des `switch`, eine Fehlermeldung ausgegeben, da ein dritter (vierter, etc.) fester Parameter nicht verarbeitet werden kann. Indem statt `switch (k)`, die Anweisung `switch (k++)` verwendet wird, wird der zugewiesene Parameter gleichzeitig gezählt.

Nehmen wir nun an, daß das aktuelle Argument mit einem '`-`' beginnt, daß es sich also um einen optionalen Parameter handelt. D.h. wir betrachten den ersten Zweig der `if`-Anweisung. In einer `while`-Schleife werden die nach dem '`-`' angegebenen Zeichen durchlaufen. Dies geschieht, weil eine Option gewöhnlich aus einem einzelnen Buchstaben besteht. Zum Beispiel gibt es für den `ls`-Befehl (list files) die Option `l` (Aufruf also als `ls -l`), die die „lange" Ausgabe der Dateien einschaltet, d.h. es werden zu jeder Datei nicht nur ihr Name, sondern auch das letzte Änderungsdatum, ihre Größe etc. ausgegeben. Zwar kann man bei einigen Programmen auch ganze Worte als Parameter angeben (z.B. beim Gnu-C-Compiler `gcc` die Option `ansi` — Aufruf also `gcc -ansi`), auf diese Möglichkeit wurde in dem vorliegenden Programmgerüst jedoch der Einfachheit halber verzichtet. Auch sind einzelne Buchstaben, so kryptisch ihre Angabe am Anfang erscheinen mag, bei häufiger Benutzung eines Programms wesentlich angenehmer, da deutlich weniger Zeichen auf der Kommandozeile einzugeben sind.

Durch die Schleife über die auf das '`-`' folgenden Buchstaben wird ermöglicht, mehrere Optionen in Kombination in einem Parameter anzugeben. Verzichtet man auf die Schleife über die Zeichen des optionalen Parameters, so muß für jede Option eigens ein '`-`' angegeben werden, was bei vielen anzugebenden Optionen unbequem ist.

Für jedes Zeichen wird durch eine `switch`-Anweisung auf den zugehörigen Fall verzweigt, durch das `s++` automatisch das Zeichen übersprungen. In dem vorliegenden Programmgerüst gibt es vier mögliche Optionen: `g`, `i`, `f` und `s`. Wird ein anderes Zeichen gefunden, so wird im `default`-Fall eine Fehlermeldung ausgegeben. (Dabei ist zu beachten, daß der Zeiger auf das aktuelle Zeichen zunächst mit `--s` erniedrigt werden muß, da der Optionsbuchstabe bereits durch das `s++` in der `switch`-Anweisung übersprungen wurde.) Die vier möglichen Optionen stehen für die Angabe eines optionalen Flags (`g`, nur der Buchstabe ist anzugeben), einer optionalen Ganzzahl (`i` — integer, Buchstabe gefolgt von einer ganzen Zahl), einer optionalen Fließkommazahl (`f` — float, Buchstabe gefolgt von einer Fließkommazahl) und einer optionalen Zeichenkette (`s` — string, Buchstabe direkt oder durch ein Leerzeichen getrennt gefolgt von einer Zeichenkette).

Im Falle des Flag ist die Auswertung der Option mit dem Setzen der zugehörigen Variable auf 1$\hat{=}$wahr abgeschlossen. Im Falle der ganzen Zahl wird die auf den Buchstaben `i` folgenden Zeichenkette (auf die ja die Variable `s` wegen des Überspringens des Optionsbuchstaben durch das `s++` in der `switch`-Anweisung zeigt), der in der Header-Datei `stdlib.h` definierten Funktion `strtol` („string to long int") übergeben. Diese erhält als dritten Parameter die Basis des Zahlensystems, in der die Zeichenkette interpretiert werden soll (2–36) und als zweiten Parameter einen Zeiger auf eine Variable, in der ein Zeiger auf das erste nicht mehr in eine Zahl zu wandelnde Zeichen abgelegt wird. Indem als zweiter Parameter die Adresse der Variable `s` übergeben wird, zeigt `s` nach der Ausführung der Funktion auf das erste auf die Zahl folgende Zeichen. Die Auswertung einer Fließkommazahl (Option `f`) erfolgt im Prinzip genauso wie die Auswertung einer ganzen Zahl, nur daß statt der Funktion `strol` die Funktion `strtod` („string to double") verwendet wird.

Sowohl bei ganzen Zahlen als auch bei Fließkommazahlen muß die Zahl direkt (also ohne dazwischenstehendes Leerzeichen) auf den Optionsbuchstaben folgen. Dies hat seinen Grund darin, daß erstens negative Zahlen wegen des führenden '`-`' nicht von Optionen unterschieden werden können, zweitens eine Auswertung eines durch ein Leerzeichen vom Optionsbuchstaben getrennten Optionsargumentes nur bei Zeichenketten mit geringem Aufwand erfolgen kann.

Im Gegensatz zu Zahlenargumenten darf zwischen einem Optionsbuchstaben und einem Zeichenkettenargument ein Leerzeichen stehen (man beachte: es kann, muß aber nicht stehen). Das Holen eines Optionsargumentes, das eine Zeichenkette ist, erfolgt über die Variable `optarg`, deren Abfrage wir bisher übersprungen haben. `optarg` wird mit `NULL` initialisiert, um anzuzeigen, daß zunächst kein Optionsargument erwartet wird. Tritt nun eine Option unter den Kommandozeilenparametern auf, die eine Zeichenkette als Argument erfordert (in diesem Beispielprogramm die Option `s`), so wird in `optarg` die Adresse derjenigen Zeichenkettenvariable gespeichert, in die ein Zeiger auf das Argument abgelegt werden soll. Dadurch wird `optarg` natürlich ungleich `NULL`, so daß nun nach einem Optionsargument gesucht wird.

Diese Suche geschieht durch die Abfrage, die wir in der bisherigen Besprechung der Kommandozeilenauswertung übersprungen haben, sowie die Abfrage nach der `switch`-Anweisung, die die verschiedenen Optionen abfragt. Durch letztere wird der eventuell vorhandene Rest eines Kommandozeilenparameters, der mit einem '`-`' begann, als Optionsargument interpretiert. Durch erstere wird dieses der normalen Argumentauswertung vorenthalten und stattdessen über `optarg` dem zugehörigen Zeiger auf eine Zeichenkette zugewiesen. Die normale Argumentauswertung wird durch den `continue`-Befehl, der den nächsten Schleifendurchlauf der `for`-Schleife anstößt, übersprungen.

Nachdem alle Kommandozeilenparameter durchlaufen wurden, wird geprüft, ob nicht noch ein Optionsargument aussteht und ob die richtige Anzahl fester Programmparameter angegeben wurden. Fehlt noch ein Options- oder ein festes Argument, so wird eine entsprechende Fehlermeldung ausgegeben. Anschließend wird, wie schon weiter oben angesprochen, die Startmeldung durch ein Zeilenendezeichen abgeschlossen.

Es folgt der programmspezifische Teil der `main`-Funktion. Er besteht in diesem Programmgerüst aus der einfachen Ausgabe des (ggf. durch die Kommandozeilenparameter geänderten) Flags, der ganzen und der reellen Zahl, der Zeichenkette, sowie der beiden festen Argumente. Danach wird das Programm regulär durch ein `return 0;` beendet.

Festlegen eigener Programmoptionen

Um eigene Programmoptionen festzulegen, muß man drei Dinge tun. Erstens muß man eine Variable definieren, die den über die Option anzugebenden Wert erhält. Sie kann, je nach Typ der Option, ganzzahlig, reellwertig oder eine Zeichenkette sein. Zweitens sollte man die Option in einer Zeile der `usage`-Meldung kurz beschreiben. Drittens hat man in die `switch`-Anweisung im `then`-Teil der `if`-Anweisung in der `for`-Schleife, die die Kommandozeilenparameter durchläuft, einen Fall (ein `case`) einzufügen, in dem entweder das Setzen eines Flags, das Einlesen einer Zahl oder das Ablegen der Adresse der zu setzenden Zeichenkettenvariable erfolgt.

Im Programm kann dann der Wert der definierten Variable abgefragt und das Programmverhalten diesem Wert entsprechend angepaßt werden. Ggf. ist es dazu nötig, die Variable nicht in der Funktion `main` sondern global zu definieren. Man sollte darauf achten, daß die Variable bei der Definition mit einem Default-Wert initialisiert wird, damit auch bei fehlender Angabe eines entsprechenden optionalen Parameters die Variable einen definierten Wert hat.

Festlegen eigener fester Programmparameter

Wie für Programmoptionen ist eine Variable anzulegen, die den Wert des festen Parameters aufnimmt. Dies muß nicht unbedingt (wie in diesem Programm) eine Zeichenketten-, sondern kann auch eine ganzzahlige oder reellwertige Variable sein. Die Zeichenkette, die man als Parameter erhält, kann ja direkt, z.B. durch eine der Funktionen `strtol` oder `strtod`, in eine Zahl gewandelt werden. Weiter ist die `switch`-Anweisung im `else`-Fall der `if`-Anweisung der `for`-Schleife, die die Kommandozeilenparameter durchläuft, mit einem entsprechenden Fall (einem `case`) zu versehen.

Christian Borgelt