Dieter Scholz

Programmieranleitung

und -richtlinie

Für das Projekt:
"Entwicklung eines DV-Werkzeuges zum Entwurf von Flugsteuerungssystemen und ihrer Energieversorgung"

Programmierung mit ANSI-C

am Arbeitsbereich Flugzeug-Systemtechnik

Version 1.1

Vorwort

Diese Programmieranleitung und Programmierrichtlinie wurde erstellt für Studenten am Arbeitsbereich Flugzeug-Systemtechnik, die sich mit einer Programmierung in ANSI C beschäftigen. Die Schrift versucht - nach bestem Wissen - eine Anleitung zu geben. Die vorliegende Version stellt die Grundlage dar für weitere Ergänzungen. Insbesondere zu Kapitel 5 "Spezielle Programmierhinweise" können im Laufe der Zeit noch weitere Unterkapitel aufgenommen werden. Die praktische Erfahrung, die dieser Schrift zugrunde liegt wurde von Studenten und Betreuer mit Microsoft C/C++ in der Version 7.0 gewonnen. Wenn in Zukunft Erfahrungen mit anderen Compilern vorliegen, können entsprechend Kapitel 6 "Hinweise zur Programmierung mit Microsoft C/C++" weitere Kapitel folgen. Es wurde versucht, die Aussagen der Kapitel 1 bis 4 so allgemein zu halten, daß sie vielfältig anwendbar bleiben. Änderungen an den Aussagen dieser Kapitel sollten nur notwendig werden, wenn die weitere Programmierpraxis einen erheblichen Bedarf für eine Korrektur aufzeigt.[1]

Jede Aussage in einer Programmierrichtlinie ist angreifbar und offen für Diskussionen. Das liegt in der Natur der Sache. Fest steht, daß eine Programmierung im Team - ohne Festlegungen irgendwelcher Art - in keinem Fall zum Ziel führen kann.

Hamburg, Juli 1994 Dieter Scholz

---

[1] Die Version 1.0 der Programmierrichtlinie wurde eingehend diskutiert. Insbesondere Dank der Durchsicht der Version 1.0 durch Herrn Dr. Husung des Rechenzentrums der TUHH konnte die vorliegende Version 1.1 ergänzt werden. Änderungen gegenüber der Version 1.0 sind durch eine Korrekturkennung in Form eines senkrechten Striches am linken Rand gekennzeichnet.

Inhalt

1. Einleitung

Bekanntlich führen viele Wege nach Rom - wir haben uns vorgenommen einen gemeinsamen Weg zu gehen. Das ist jedoch nur möglich, wenn man sich bei einer Arbeit im Team auf ein gemeinsames Vorgehen verständigt. Warum ist dies notwendig? Kann nicht jeder sein Programm so schreiben wie es ihm/ihr gefällt, unter der Voraussetzung, daß es "läuft"? Nein, denn: 70 Prozent der Softwarekosten entfallen auf Wartung, Fehlerfindung und Korrekturen [1]. Diesen Aufwand gilt es zu reduzieren.

Wie soll das erreicht werden?

1.) **Einheitliche Programmgestaltung** --- erhöht die Lesbarkeit.
2.) **Strukturierte Programmierung** --- im Sinne der Erfinder vermindert den Wartungsaufwand.
3.) **Komplexitätsreduktion** von Funktionen und **Modularisierung** --- gewährleistet schnelles Einlesen in Algorithmen und erhöht ihre Nutzbarmachung.

Der erfahrene C-Programmierer wird in dieser Programmieranleitung/Programmierrichtlinie sicher keine Neuigkeiten entdecken. Ein Einsteiger in das Programmieren in C hingegen wird hoffentlich einige nützliche Hinweise und Empfehlungen finden, die in Lehrbüchern zur Programmiersprache nicht enthalten sind. Die hier aufgenommenen Hinweise sind deshalb in C-Büchern nicht zu finden, weil C dem Programmierer viele Freiheiten läßt. Ein allgemeines Lehrbuch kann daher keine Einschränkungen machen, die vom Standard nicht gefordert werden. Auf den Einsteiger können diese Freiheiten der Sprache C jedoch verwirrend wirken, so daß einige richtungsweisende Hinweise sicher willkommen aufgenommen werden können.

Diese Schrift hat nicht den Anspruch ein C-Lehrbuch zu ersetzen. Es ist vielmehr daran gedacht, daß das Lehrbuch immer dann zu Rate gezogen wird, wenn Aussagen oder Beispiele hier im Text noch Fragen offen lassen.

Aussagen einer Programmierrichtline können in drei Kategorien gegliedert sein:

(1) zwingend vorgeschrieben
(2) vorgeschrieben, begründete Ausnahmen (nach Absprache) erlaubt
(3) erwünscht, aber nicht vorgeschrieben

Diese Programmieranleitung und -richtlinie verzichtet auf diese strenge Einordnung. Die Aussagen in diesem Text fallen in Kategorie (2) solange im Text durch entsprechende Wortwahl nichts anderes erwähnt wird.

Zur Einarbeitung in C sollte eines der vielen C-Lehrbücher herangezogen werden. Als Beispiel sei nur das C-Standardwerk von Kernighan und Ritchie [2] erwähnt. Als preiswertes Nachschlagewerk eignet sich die Schrift des "Regionalen Rechenzentrum für Niedersachsen" [3].

Diese Schrift wurde erstellt, aufbauend auf [4] und [5].

2. Programmierstil

2.1. Allgemeines Layout

C-Programme können im Prinzip formatfrei - also beliebig - eingegeben werden. Programmierer, die von dieser Freiheit zu großen Gebrauch machen, erstellen Programme, die unlesbar und damit unbrauchbar sind, auch wenn die Programme zufällig mal "laufen" sollten.

In der "C-Gemeinde" hat sich ein gewisser informeller Standard mit verschiedenen kleinen Varianten entwickelt, der besagt, wie C-Programme formatiert werden sollten. Die in Lehrbüchern aufgelisteten C-Programme geben diesen informellen Standard implizit wieder.

Die Möglichkeiten zur Formatierung sind jedoch derart vielfältig, daß eine lückenlose explizite Beschreibung hier einfach zu aufwendig erschien. Stattdessen soll folgendermaßen vorgegangen werden:

1.) Einige Grundsätze zur Formatierung werden im folgenden Text angegeben.
2.) Am Arbeitsbereich steht ein Programm zur Verfügung, mit dessen Hilfe ein C-Programm in eine Variante des informellen C-Standards umgeschrieben werden kann.[2]

Es wurde dabei sichergestellt, daß die hier aufgeführten Hinweise zur Formatierung (1.) und die Ausgabe des Programms (2.) sich sinnvoll ergänzen und nicht zueinander im Widerspruch stehen.

Hier also zu (1.): Um übersichtliche Programme zu erhalten, müssen die Programmzeilen vollständig lesbar sein. Man kann davon ausgehen, daß Editoren 78 Zeichen pro Zeile anzeigen können. Damit der Cursor noch hinter dem letzten Zeichen der Zeile stehen kann, ohne daß ein "Scrollen" des Bildschirms erforderlich ist, dürfen demnach nicht mehr als 77 Zeichen in eine Zeile eingegeben werden.

Einrückungen sind in C wichtig. Für Einrückungen wird die Tabulatortaste genutzt. Einrückungen sollen deutlich sichtbar sein, jedoch den Programmtext nicht schon bei wenigen Einrückungen ganz nach rechts verschieben. Als Kompromiß hat sich ein Tabulatorschritt von 4 Zeichen bewährt.

---

[2] Es handelt sich um das Programm INDENT. Wird das Programm mit den Parametern

```
-bap -bad -bbb -nbc -br -c0 -ncdb -nce -cil -clil
-d0 -dil6 -nfcl -nip -l77 -nlp -npcs -npsl -sc -nsob
```

aufgerufen und im Editor ein Tabulatorschritt von vier Zeichen eingestellt, so erhält man das gewünschte Ergebnis.

Ein Programm, welches mit den oben beschriebenen Vorgaben erstellt wurde, kann dann problemlos so ausgegeben werden:

Schriftart:	**keine** Proportionalschrift; z.B. Courier verwenden
Schriftgröße:	12 cpi bzw. 10 Points
Tabulator:	alle 0,81 cm
linker Rand:	3 cm
rechter Rand:	1 cm
Seitenzahl:	oben auf jede Seite.

Platz sparen ist beim Schreiben der Programme nicht das oberste Ziel. Jedoch ist zu beachten, daß man leichter einen Überblick erhält, wenn man mehr Text gleichzeitig im Blick hat. Demnach sind unnötige Leerzeilen zu vermeiden. Eine logisch zusammenhängende Anzahl von Programmzeilen sollte jedoch als Block durch eine Leerzeile von weiteren Blöcken getrennt werden. Jeder Block soll mit einem Blockkommentar beginnen (siehe Kapitel 2.4).

Leerzeichen können die Übersicht erhöhen. Bewährt hat sich Leerzeichen zu setzen um "=", "+", "-". Weiterhin dann wenn viele Klammern "()" aufeinandertreffen um die visuelle Gruppierung zu unterstützen.

In Formeln sind zusätzliche **Klammern** zu setzen um die Lesbarkeit zu erhöhen, auch dann wenn sie aus mathematischen Gründen nicht zwingend erforderlich sind. So hat es sich z.B. bewährt bei größeren Brüchen die Ausdrücke im Zähler und Nenner jeweils in Klammern zu setzen.

2.2. Zeichensatz

Es ist nur der in C zulässige Zeichensatz zu verwenden:

```
A B C D ... Z
a b c d ... z
0 1 2 3 ... 9
( ) [ ] { } < > + - * ^ / % ~ & | _ = ! ? # \ , . ; : ' "
```

Dies gilt für Kommentare ebenso wie für Zeichenfolgen-Konstanten. Weil der ASCII-Zeichensatz für Umlaute nicht genormt ist, ist von deren Verwendung abzuraten. Stattdessen sind Umlaute als ae, ue ... zu schreiben.

2.3. Bezeichner

Bezeichner beginnen mit einem Buchstaben und können durch einen Buchstaben oder eine Ziffer fortgesetzt werden. Formal gilt der Unterstrich als Buchstabe.

Um nicht mit internen Bezeichnern der Sprache C zu kollidieren dürfen keine Bezeichner verwendet werden, die mit einem Unterstrich: "_", zwei Unterstrichen: "__", "str" oder "mem" beginnen [6].

Bei **intern**en Bezeichnern sind nur die ersten 31 Zeichen signifikant. Darum sollen interne Bezeichner nicht länger als 31 Zeichen sein. Bezeichner, die **extern** sind, sollen möglichst kurz gewählt werden. Wenn möglich, maximal 6 Zeichen. Man kann bei externen Bezeichnern nicht davon ausgehen, daß Groß- und Kleinschreibung unterschieden werden [7].

Bezeichner sollten volle Worte und keine Kürzel sein. Ausnahmen stellen Indizes und lokal benutzte Hilfsvariable (z.B. Zwischenspeicher) dar. Die verwendeten Worte sollen aus dem jeweiligen Umfeld der Aufgabenstellung kommen. Wenn mit englischer Literatur gearbeitet wird, so können englische Worte genutzt werden. Auf eine Einheitlichkeit der Benennung ist jedoch unbedingt zu achten.

Variablen

Der Name einer Variablen beschreibt ihren Inhalt. Er ist ein Substantiv und beginnt mit einem kleinen Buchstaben. Bei Zeigervariablen kann dem Namen das Zeichen 'P' für Pointer nachgestellt werden. Wird eine Variable durch mehrere Worte beschrieben, so beginnt jedes einzelne Wort mit einem Großbuchstaben.
Beispiel: zeilenZaehler, fileName, userText, databaseP

Bei **Maße**n sollte die Einheit dann Bestandteil des Namens sein, wenn sie nicht die Standard-SI-Einheit ist.
Beispiele: volumenInGallonen

Formelzeichen werden so weit es geht direkt umgesetzt in die Programmierung. Griechische Kleinbuchstaben werden als "alpha", "beta"... dargestellt. Griechische Großbuchstaben werden als "Alpha", "Beta"... dargestellt. Falls Bezeichner mit griechischen Buchstaben bei dieser Art der Schreibweise zu lang werden, so können die griechischen Buchstaben in folgender Weise abgekürzt werden [8]:

alp, bet, gam, del, eps, zet, eta, the, iot, kap, lam, my, ny, xi, omi, pi, rho, sig, tau, yps, phi, chi, psi, ome.

Bei der Darstellung von Formelzeichen orientiert man sich am Vorgehen der Formeleditoren entsprechender Textverarbeitungsprogramme wie eqn (UNIX), TeX oder WordPerfect. Indizes werden mit Hilfe des Unterstrichs an das Formelzeichen angefügt (das ist kürzer als "Sub"). Hochgestellte Zeichen werden mit "Sup" eingeleitet. Nachfolgend ist eine (unvollständige) Liste der entsprechenden Symbole angegeben, angepaßt an die Verhältnisse von C:

Dot	ein Punkt auf einem Zeichen
DDot	zwei Punkte auf einem Zeichen
DDDot	drei Punkte auf einem Zeichen
Hat	ein Dach auf einem Zeichen
Bar	ein Querstrich auf einem Zeichen
Vec	ein Vektorpfeil auf einem Zeichen
Tilde	eine Tilde auf einem Zeichen
Prime	ein Strich hinter einem Zeichen

PPrime zwei Striche hinter einem Zeichen
PPPrime drei Striche hinter einem Zeichen

Beispiele: `v_max, Lamda_c4, vDot_0, vBar, c_b_t, xSup0, x_aSup0, xPPrime`

Literale Ersetzungen

Das Schlüsselwort **#define** leitet literale Ersetzungen ein. Alle literalen Ersetzungen sind durchgängig in Großbuchstaben zu schreiben. Besteht eine literale Ersetzung aus mehreren Worten, so werden die einzelnen Worte durch einen Unterstrich voneinander getrennt.

Beispiele:
```
#define     MAX_ANZAHL      10
#define     TESTSTRING      "Testwert"
```

Funktionen

Ein C-Programm kann durch die intelligente Nutzung von Funktionen aufgeteilt werden. Es kann es hilfreich sein, folgende Arten von Funktionen zu unterscheiden:

(A) Funktionen, die ihr Ergebnis nur über den Funktionswert an den aufrufenden Programmteil zurück geben.

(B) Funktionen, die ihre Ergebnisse über die Parameterliste an den aufrufenden Programmteil zurück liefern. Mit dem Funktionswert wird die Fehlerinformation übergeben.

Diese Unterscheidung sollte auch bei der Wahl des Bezeichners der Funktion berücksichtigt werden:

Der Name einer Funktion gemäß (A) beschreibt ihr Ergebnis. Der Name beginnt mit einem Großbuchstaben. Eine Funktion unterscheidet sich von einer Variablen weiterhin noch durch das stets anzugebende - evt. auch leere - Klammerpaar.
Beispiele: `Umfang, EmpfangenesZeichen, Verbrauch`

Der Name einer Funktion gemäß (B) beschreibt ihre Aktion. Es sind einleitende Verben zu verwenden. Der Name beginnt mit einem Großbuchstaben.
Beispiele: `BildePruefsumme, ReadFile, GetToken`

Auf die Verwendung von **Konstanten** (const) ist zu verzichten. Stattdessen sollte mit literalen Ersetzungen (#define) gearbeitet werden.

Aufzählungskonstanten werden durchgehend mit Großbuchstaben geschrieben.
Beispiel: `enum Bool {FALSE, TRUE};`

Bezeichner von **Aufzählungen** beginnen mit einem Großbuchstaben.
Beispiel: `enum Jahreszeiten { FRUEHJAHR, SOMMER, HERBST, WINTER };`

Bezeichner von **Strukturen** beginnen mit einem Großbuchstaben.
Beispiel:
```
struct Complex {
      double re;
      double im;
};
```

Namen von eigenen **Typen** werden durchgehend mit Großbuchstaben geschrieben. (Die Basistypen von C wie "int", "char" und "float" werden mit kleinen Buchstaben geschrieben.)
Beispiel:
```
typedef enum Bool BOOL;
typedef enum {FALSE, TRUE} BOOL;
typedef struct Complex COMPLEX;
BOOL fehler;
COMPLEX a, b, c;
int i, j, k;
```

2.4. Kommentare

Ein Programm sollte einsprachig kommentiert werden. Wir haben uns für deutsch entschieden.

Kommentare müssen inhaltlich Bezug nehmen und sollen nicht das ausdrücken, was ohnehin aus dem Code ersichtlich ist.
Negativbeispiel:
```
i = i + 1;  /* i wird um 1 erhoeht */
```

Folgen zusammengehöriger Anweisungen sind durch einen Blockkommentar zu dokumentieren. Der Block (einschließlich Kommentar) ist mit einer Leerzeile, von den benachbarten Blöcken zu trennen. Der Blockkommentar steht vor dem Block. Der Blockkommentar hat die gleiche Tiefe der Einrückung wie der zu kommentierende Block. Blockkommentare sind problemorientierter als Zeilenkommentare. Die Übersichtlichkeit wird dadurch erhöht. Das Layout kann dem folgenden Beispielen entnommen werden:

```
/* Hier steht ein Blockkommentar, der nur aus einer Zeile besteht. */

/* Hier steht ein Blockkommentar, der
 * aus mehreren Zeilen
 * besteht. */
```

Zeilenkommentare können dort, wo erforderlich, ergänzende Hinweise geben. Der Kommentar steht dann am Ende der Zeile (siehe oben) und wird - falls erforderlich - eingerückt in der nächsten Zeile fortgesetzt.
Beispiel:
```
if(error) return(7); /* Fehler:
                      * Nicht genug Speicherplatz
                      * vorhanden */
```

2.5. Zuweisungen

Seiteneffekte in Funktionsaufrufen sind verboten. Grund: Das Ergebnis kann unvorhersagbare Werte annehmen!
Beispiel: `c = getc(i++);  /* VERBOTEN */`

Multiple Zuweisungen sind verboten, weil sie unnötig sind.
Beispiel: `int a = b = c = 0;  /* VERBOTEN */`

Eingebettete Zuweisungen sind nur mit äußerster Vorsicht und sparsam zu gebrauchen.
Beispiel: `if( c = getc(input) ) return(0);`[3] `/* VORSICHT */`
Die Gefahr bei diesem Beispiel liegt darin, daß der Programmierer '==' gemeint, jedoch '=' programmiert hat, oder, daß dem Leser eine entsprechenden Verwechslung unterläuft.

2.6. Deklarationen und Definitionen

Variablen sollte bei der Definition zusätzlich auch ein Wert gegeben werden, mit dem die Variable vorbesetzt (initialisiert) wird.
Beispiel:
```
int i=0;
```

Größere Felder sollten nicht lokal (also über "eckige Klammern") definiert werden. Stattdessen sollte nur ein Zeiger auf das Feld deklariert werden und der Speicherplatz mit malloc (oder calloc) besorgt werden.[4]
Beispiel:
```
int *ganzzahl;
ganzzahl = (int *)malloc(100*sizeof(int));
```

Das gleiche gilt für **größere Strukturen**:
Beispiel:
```
MEGARECORD  *info;
info = (MEGARECORD *)malloc(sizeof(MEGARECORD));
```

Der strukturierte Datentyp **union** sollte vermieden werden [6]. Eine Verwendung kommt erst dann in Frage, wenn ohne dessen Verwendung ein Mangel an freiem Speicher zu befürcheten ist.

---

[3] Das Beispiel könnte man evt. noch durch folgende Programmierung "in den Griff" bekommen:

```
if( (c = getc(input)) != 0 ) return(0);
```

[4] Bei der Programmierung von PCs ist zu beachten: Wird eine Variable lokal definiert, so wird der Speicherplatz im (begrenzten) Stack reserviert. Dies läßt sich vermeiden
1.) durch malloc oder calloc und Verwendung des Speichermodells "Large";
2.) dadurch, daß die Variable static definiert wird und durch Compileroption (MS C/C++ 7.0: /Gt 3) in den far heap gebracht wird.

Alle **Funktionen** sollen nach ANSI C deklariert werden. Eine solche Deklaration liefert den Funktionsprototypen[5], der in die zugehörige Deklarationsdatei (Header-Datei) geschrieben wird. Der Funktionsprototyp ermöglicht dem Compiler eine Kontrolle über die typgerechte Wertübergabe an die Funktion. Funktionen, die ihre Ergebnisse über die Parameterliste an den aufrufenden Programmteil zurück liefern, sollen zur Übergabe von Fehlermeldungen - gemäß Kapitel 5.1 - vom Typ "unsigned long" sein.

Jeder Funktionsparameter steht bei der Funktionsdeklaration und -definition auf einer eigenen Zeile. Funktionsparameter können dann in eine Zeile geschrieben werden, wenn alle Funktionsparameter zusammen mit dem Funktionsnamen in eine Zeile passen. (Siehe weiterhin Kapitel 4.2 zum Thema "Funktionsdefinition").

2.7. Kontrollstrukturen

Die Schachtelungstiefe von Blöcken sollte 5 nicht überschreiten. Werden erheblich größere Schachtelungstiefen notwendig, so ist zu überlegen, ob das Problem nicht in mehrere Funktionen zerlegt werden kann.

Die **goto-Anweisung** soll nie verwendet werden. Ihre Nutzung widerspricht den Grundsätzen der strukturierten Programmierung. Programme werden unübersichtlich. Die goto-Anweisung kann immer durch andere Konstrukte ersetzt werden. Weitere Erklärungen entnehme man [2].

Die nachfolgenden Beispiele von Kontrollstrukturen zeigen ein sinnvolles Layout. Der abschließende Kommentar am Ende der Kontrollstruktur kann bei kurzen übersichtlichen Kontrollstrukturen entfallen.

Entscheidung (if-Anweisung)

```
if ( Bedingung ) {
   Anweisung1;
   Anweisung2;
}
else {
   Anweisung3;
   Anweisung4;
} /* if Bedingung */
```

wenn der else-Zweig entfällt:

```
if ( Bedingung ) {
   Anweisung1;
   Anweisung2;
} /* if Bedingung */
```

---

[5] Mit Microsoft C/C++ lassen sich die Funktionsprototypen automatisch mit der Compiler-Option /Zg erzeugen.

Auswahl (switch-Anweisung)

```
switch ( auswahlVariable ) {
   case A:
      Anweisungen;
      break;
   case B:
   case C:
      Anweisungen;
      break;
   default:
      Anweisungen;
      break;
} /* switch auswahlVariable */
```

Wiederholungen

while-Schleife:

```
while ( Bedingung ) {
   Anweisungen;
} /* while Bedingung */
```

do-while-Schleife:

```
do { /* Bedingung */
   Anweisungen;
} while ( Bedingung );
```

for-Schleife:

```
for ( Initialisierung; Bedingung; Inkrementierung ) {
   Anweisungen;
} /* for Index */
```

3. Portierbarkeit

Die Portierbarkeit eines Programms stellt die Möglichkeit dar, das entsprechende Programm auf einem anderen Rechner laufen zu lassen. Die Portierbarkeit eines Programms erhöht dessen Wieder- und Weiterverwendbarkeit. Portierbarkeit ist somit Voraussetzung für einen langfristigen Schutz der Investition von Zeit und Geld in das erstellte Programm.

Eine Programmierung in ANSI-C ist ein wichtiger Schritt zur Portierbarkeit. Eine Programmierung von Graphik ist in ANSI-C jedoch nicht möglich. Die Datenverwaltung erfordert weitere Gedanken, um eine Portierbarkeit sicherzustellen.

3.1. ANSI-C

Eine Kommission des American Natioanl Standards Institute hat den Versuch unternommen, "eine eindeutige und maschinenunabhängige Definition der Sprache C" [2] zu erarbeiten. Eine Programmierung in ANSI-C stellt somit die Basis dar, für eine spätere Möglichkeit der Portierung der Software.

C-Compiler können in der Regel mehr als durch den ANSI-Standard gefordert. Diese "Mehr" ist dann aber i. d. R. nicht portierbar. Aus diesem Grund ist nur in ANSI-C zu programmieren. Eine Prüfung des Programms auf ANSI-Kompatibilität ist erforderlich und kann durch entsprechende Compiler-Optionen eingestellt werden.[6] Nicht alle Verstöße gegen ANSI-C werden jedoch durch eine ANSI-C Compiler-Option entdeckt. Es bleibt damit die Verantwortung eines jeden Programmierers, auf die Einhaltung einer ANSI-C-gerechten Programmierung zu achten. Hier einige weitere Hinweise (ohne Anspruch auf Vollständigkeit) aus [6] und [7]:

- Geschachtelte Kommentare sind verboten, weil sie nicht auf jedem Rechner möglich sind.

- Eine Annahme über die Größe von Datentypen darf nicht gemacht werden. Stattdessen ist die entsprechende Größe vom Programm mit Hilfe des sizeof-Operators zu ermitteln. Andere maschinenabhängige Größen stehen in der Include-Datei limits.h und sind bei Bedarf zu verwenden.

- Für eine Portabilität dürfen Funktionen keine variable Anzahl von Argumenten besitzen.

- Wenn auf Dateien zugegriffen wird, so dürfen Datei- und Verzeichnisnamen nicht fest codiert werden. Ebensowenig darf das Verhalten eines bestimmten Betriebssystems vorausgesetzt werden. Hinweis: In C-Programmen unter DOS können Pfade mit "\\" oder mit "/" geschrieben werden. "/" ist zu verwenden,

---

[6] Im Fall von Microsoft C/C++ ist dies die Compiler-Option /Za .

weil diese Schreibweise ebenfalls für UNIX gilt.

3.2. Graphik

Graphik-Funktionen gehören nicht zum ANSI-Standard. C Funktionen zur Graphik dürfen daher nicht verwendet werden, wenn eine Portierbarkeit gefordert ist. Stattdessen können - je nach Anwendung - **portable Graphikbibliotheken** verwendet werden oder es kann ein "User Interface Management System" (**UIMS**) eingesetzt werden. Als technisch-wissenschaftliches Plottprogramm kann das Public Domain Tool **GnuPlot** eingesetzt werden. GnuPlot ist auf alle interessierende Plattformen portiert worden. Beispiele zur Einbindung von GnuPlot in C liegen am Arbeitsbereich bereit.

3.3. Datenverwaltung

Daten, die auch nach dem Ende des Programms noch Bestand haben sollen, müssen in eine Datei auf Festplatte oder Diskette geschrieben werden.

Wenn die Daten nur auf dem jeweiligen Rechner von Bedeutung sind, so können die Daten mit den Befehlen **fread** und **fwrite** gelesen bzw. geschrieben werden. Für eine Portierbarkeit der Programme ist hierbei wichtig, das die Größe der bewegten Struktur mit Hilfe des sizeof Operators vom Programm berechnet werden muß.

Wenn die Portierbarkeit der Daten selbst erforderlich ist, Daten also auch auf einen anderen Rechner übertragen werden müssen, dann dürfen nur ASCII-Dateien geschrieben bzw. gelesen werden. Es soll nur der 7-bit-ASCII-Zeichensatz Verwendung finden. Das bedeutet vor allem, daß Umlaute nicht erlaubt sind. Da man nie sicher sein kann, ob Daten später nicht doch portiert werden sollen, ist eine Speicherung im 7-bit-ASCII-Zeichensatz immer zu empfehlen.

Wenn die Daten immer in gleicher Anzahl vorliegen und vor allem in der gleichen Reihenfolge, so können die Daten unmittelbar, sequentiell in die Datei geschrieben werden. Liegen die Daten jedoch nur vereinzelt vor und ist zudem die Gesamtzahl unbekannt, so müssen die Daten in der Form Schlüsselwort / Wert abgespeichert werden. Entsprechende Funktionen zur Manipulation solcher Daten liegen am Arbeitsbereich vor. Diese Funktionen legen u. a. eine verkettete Liste mit den Schlüsselworten und Werten an.

4. Programmstruktur

Funktionen stellen die kleinste Einheit bei einer modularen Programmierung dar. Eine Funktion hat definierte Eingangs- und Ausgangsparameter. Diese definierten Parameter stellen die Schnittstelle zur Funktion dar. Funktionen dienen als "black box". D. h., daß das "Innenleben" der Funktion dem Anwender verborgen bleibt. Funktionen können als eigene Programmeinheit ausgeführt werden (diese Programmeinheit wird dann auch "Funktion" genannt), oder als Textersetzung des Präprozessors (in diesem Fall spricht man von einem "Makro"). Funktionen (und Makros), die in einer Datei zusammengefaßt sind ergeben ein Modul. Eine logische Gliederung der Programme in Funktionen und Module sind für das Gelingen eines Softwareprojektes von entscheidender Bedeutung. Dies soll durch Anwendung von Methoden des "Computer Aided Software Engineering" (CASE) sichergestellt werden.

4.1. Makros

Eine Funktion kann in C als eigene Programmeinheit ausgeführt sein, die übersetzt und mit anderen Programmeinheiten zusammen gebunden wird. Eine Funktion kann jedoch auch als Makro ausgeführt sein. In diesem Fall wird Quelltext an der entsprechenden Stelle des Programms durch den Präprozessor eingefügt. Für den Nutzer einer Funktion ist nicht zu unterscheiden, ob eine Funktion als Programmeinheit oder als Makro realisiert ist. Makros sind schneller als die eigentlichen Funktionen, die als Programmeinheit realisiert sind. Makros ergeben jedoch Programme, die mehr Programmspeicher erfordern. Beim Aufruf von Makros müssen Seiteneffekte auf jeden Fall vermieden werden, weil diese eine unkontrollierten Programmausführung bewirken können. Makros werden eingesetzt um einfache Funktionen zu erfüllen. Man wird in der Regel nur wenige Eingabeparameter für ein Makro vorsehen. Im Zweifelsfall ist eine Funktion als eigene Programmeinheit einem Makro vorzuziehen. Makros werden in die Deklarationsdatei des Moduls geschrieben, wenn sie dem Benutzer des Moduls zugänglich gemacht werden sollen. Makros, die nur innerhalb eines Moduls von Bedeutung sind werden am Anfang des Moduls nach dem Modulkopf (siehe Kapitel 4.3) definiert. Ein Makro erhält einen Kopf aus Kommentarzeilen, der im Aufbau dem Kopf einer Funktion (siehe Kapitel 4.2) entspricht.

4.2. Funktionen

Funktionen dürfen nicht zu lang sein, weil lange Funktionen leicht unübersichtlich werden. Eine Funktion sollte weniger als 120 Zeilen besitzen (ohne Funktionskopf), so daß sie der Übersichtlichkeit wegen noch auf zwei Seiten gedruckt werden kann. Sehr kurze Funktion (z.B. eine oder zwei Zeilen ohne Funktionskopf) sind dann gerechtfertigt, wenn eine klar umrissene Aufgabe mit der Funktion erfüllt wird. Es sollte jedoch auch bedacht werden, daß kurze Funktionen, die nur eine Anzahl von anderen Funktionen aufrufen, die Verschachtelungstiefe erhöhen, was ebenfalls unübersichtlich sein kann.

Funktionen sollten maximal 7 Parameter haben. Werden mehr Parameter notwendig, ist die Funktion wahrscheinlich zu komplex geraten, reicht nur Daten nach "unten" durch oder die Daten sind schlecht strukturiert. Abhilfe schafft die Zerlegung in mehrere Funktionen, oder die Verwendung von Strukturen.

Jeder Funktionsparameter steht bei der **Funktionsdefinition** auf einer eigenen Zeile. Funktionsparameter können dann in eine Zeile geschrieben werden, wenn alle Funktionsparameter zusammen mit dem Funktionsnamen in eine Zeile passen.
Beispiel:

```
unsigned long CopyFile(
            char *source,
            char *destination
)

unsigned long DisplayList(RECORD *database)
```

Jede Funktion muß mit einem Typ versehen sein - auch wenn die Funktions vom Standardtype "int" ist. Eine Funktion, die keinen Wert zurück gibt, ist vom Typ void. In der Regel wird eine etwas umfangreichere Funktion ihre Ergebnisse über die Parameterliste an den aufrufenden Programmteil zurück liefern. Der Funktionswert dient dann zur Übergabe der Fehlerinformation. (Siehe Kapitel 5.2). In diesem Fall ist die Funktion vom Type "unsigned long".

Gemäß ANSI-Standard wählen wir die Kurzform der Funktionsdefinition, bei der die Deklaration der formalen Parameter mit der ersten Zeile der Funktionsdefinition kombiniert wird. (Siehe Beispiel unten).

Funktionen müssen mit einem **Funktionskopf** (aus Kommentaren) versehen werden, um die Aufgabe und die Schnittstelle der Funktion zu dokumentieren. Das Layout wurde [9] entnommen.

Die erste Zeile besteht aus einer Strichlinie (/*---- ... ----*/), die bis in die 77. Spalte reicht.

Danach folgt die erste "Zeile" der Funktionsdefinition (wobei i.d.R. jeder Parameter in einer eigenen Zeile steht).

Der Informationsteil des Funktionskopfes beginnt und endet mit eine Linie aus Sternen, die ebenfalls bis in die 77. Spalte reicht (siehe Beispiel). Die erste Zeile des Informationsteiles des Funktionskopfes liefert eine Kurzbeschreibung der Funktion. (Diese Kurzbeschreibung sollte - wenn möglich - auch wirklich nicht länger als eine Zeile sein.)

In dem folgenden Absatz wird die Aufgabe der Funktion im einzelnen beschrieben. Die Beschreibung muß so ausführlich sein, daß ein Benutzer, der das Innere der Funktion nicht kennt mit der Funktion als "black box" arbeiten kann.

In weiteren Absätzen sind die Eingabeparameter, Ausgabeparameter, sowie der

Funktionswert der Funktion erklärt.

Alle benutzten globalen Namen der Funktion werden in einem weiteren Abschnitt aufgelistet. Externe Variablen dürfen - gemäß dieser Programmieranleitung - nicht genutzt werden. Dennoch gibt es globale Namen. Diese ergeben sich aus aufgerufenen Funktionen, oder von literalen Ersetzungen.

Zum Schluß kann ein Verweis auf ähnliche Funktionen gemacht werden.

Beispiel:

```
/*---------------------------------------------------------------------------*/
unsigned long ReadFile(
               char *fileName,
               RECORD **databaseP,
               char *tempFileName
)
/*****************************************************************************
* Lesen der Daten aus einer Datenbasis (ASCII-Datei)
*
* Daten aus dem Abschnitt "Datenliste" der Datenbasis werden eingelesen
* und in einer einfach verketteten Liste abgespeichert.
* Daten anderer nicht interessierender Abschnitte der Datenbasis werden
* in einer temporaere Datei bis zum Aufruf von WriteFile zwischengelagert.
*
* EINGABEPARAMETER:
* =================
* fileName      Zeiger auf den Namen einer Datei, aus der gelesen werden soll.
* tempFileName  Zeiger auf den Namen einer temporaeren Datei.
*
* AUSGABEPARAMETER:
* =================
* databaseP     Zeiger auf die Liste aus "Datenliste"
*
* FUNKTIONSWERT:
* ==============
* 0   kein Fehler
* 1   Datei aus der gelesen werden soll kann nicht geoeffnet werden
* 2   Datei kann nicht als Datenbasis ausgewertet werden
* 3   temporaere Datei kann nicht geoeffnet werden
* 4   Fehler bei der Anwendung von fprintf
* 5   Nicht genug Speicherplatz fuer malloc
* 6   Datenbasis enthaelt unvollstaendigen Datensatz
* 7   Fehler beim Schliessen der temporaeren Datei
* 8   Fehler beim Schliessen der Datei aus der gelesen wurde
* 9   Standardliste ist nicht in der Datei enthalten
*
* BENUTZTE GLOBALE NAMEN:
* =======================
* CheckFile, GetToken, ReadList, RECORD, FILE, fopen, fclose, fprintf,
* strcmp, strlen, malloc, EOF, NULL, FALSE, TRUE
*
* SIEHE AUCH:
* ===========
* WriteFile
*****************************************************************************/
{
       ...
```

4.3. Module

Ein Modul ist erst einmal ein Programmteil, der in einer eigenen Datei steht. Die Frage ist nun, wie viele und welche Funktionen sinnvoll zu einem Modul zusammengestellt werden sollen.

Ein Modul sollte eine überschaubare Länge haben. Eine überschaubare Länge ist dann überschritten, wenn sich die Datei nur noch schwer mit einem Editor manipulie-

ren läßt. Module sollten nicht zu klein sein (etwa nur eine Funktion), weil dadurch zu viele Module zu verwalten wären.

Zwei extreme Zusammenstellungsformen von Funktionen lassen sich denken:

1.) Nur die erste Funktion des Moduls ist diejenige, die die Schnittstelle nach außen darstellt. Alle anderen Funktionen des Moduls werden direkt oder indirekt von dieser ersten Funktion aufgerufen. Die Schnittstelle der ersten Funktion nach außen ist dann auch die Schnittstelle des Moduls nach außen. Die erste Funktion ist dann global. Alle anderen Funktionen sind lokal und werden als static deklariert und definiert.

2.) Funktionen, die von ganz verschiedenen Stellen des Programms aufgerufen werden, stellen "Service-Funktionen" dar. Funktionen mit gleicher Thematik können zu einem Modul zusammengestellt werden. Jede der Funktionen hat dann eine Schnittstelle nach außen. Demzufolge ist keine der Funktionen static deklariert und definiert. Es bietet sich an, aus einem Modul aus Service-Funktionen eine Bibliothek zu generieren. Wenn ein Programm dann die Funktionalität dieser Bibliothek benötigt, so kann die Bibliothek von Linker durchsucht werden und es werden nur die Funktionen mit dem Rest des Programms gebunden, die wirklich gebraucht werden.[7]

Wenn keine der klaren oben genannten Formen anwendbar ist, dann muß eine Mischform aus den beiden oben aufgeführten Extremen gewählt werden. Dabei ist darauf zu achten, daß die Zusammenstellung der Funktionen weiterhin eine klare Gliederung ergibt.

Der **Aufbau eines Moduls** ist folgender:

1.) Modulkopf
2.) `#include` der Deklarationsdateien der aufgerufenen C-Standardbibliotheken
3.) `#include` der Deklarationsdateien von aufgerufenen Modulen
4.) Literale Ersetzungen mit `#define` soweit nur von lokaler Bedeutung
5.) Typdefinitionen mit `typedef` soweit nur von lokaler Bedeutung
6.) Prototypen lokaler Funktionen
7.) Implementation globaler Funktionen
8.) Implementation lokaler Funktionen

Wie oben angegeben, beginnt jedes Modul mit einem **Modulkopf** aus Kommentarzeilen. Ein Beispiel für das Layout des Modulkopfes ist unten angegeben. Die erste und letzte Zeile des Kopfes reicht bis in die 77. Spalte und besteht im wesentlichen aus

---

[7] Unter UNIX kann ein Linker unter Umständen nicht in der Lage sein, einzelne Funktionen aus einem Modul (bzw. einer Objektdatei) einer Bibliothek heraus zu lösen. Der Linker kann dann nur ganze Objektdateien einer Bibliothek linken. In diesem Fall muß man mit einer ausführbaren Datei zufrieden sein, die auch noch einige nicht benötigte Funktionen enthält. Wird in jedem Fall eine kleinstmögliche ausführbare Datei gefordert, so müssen von vornherein alle eigenständigen Funktionen in ein separates Modul geschrieben werden.

Sternen.

Besonders wichtig sind die folgenden drei Angaben im Modulkopf:

Zum einen ist dies die Liste der externen Funktionen des Moduls. Hier werden alle Funktionen des Moduls aufgelistet, die von außen aufgerufen werden können und damit anderen Benutzern als "black box" zur Verfügung gestellt werden. Die Schnittstelle dieser externen Funktionen entspricht der Schnittstelle(n) des Moduls. Unter "Liste der externen Funktionen" wird der Leser auf diese Schnittstellen hingewiesen. Näheres kann dann in den Funktionsköpfen der entsprechenden Funktionen nachgelesen werden. (Siehe Kapitel 4.2).

Die "Liste der Makros" enthält die externen Funktionen des Moduls, die als Makro ausgeführt wurden. Die Makros stehen nicht in der Datei des Moduls selbst, sondern in der zum Modul gehörigen Deklarationsdatei (siehe unten). Über die "Liste der Makros" erfährt der Leser demzufolge, welche Funktionen (d.h. Makros) in der Deklarationsdatei zu finden sind.

Rufen Funktionen des vorliegenden Moduls Funktionen aus anderen Modulen auf, so müssen diese Module mit dem vorliegenden Modul zusammen gebunden werden. Unter "Liste der benötigten Module" sind solche erforderlichen Module aufzulisten.

Für jedes Modul ist ein "Jackson Structure Chart" (Aufrufdiagramm) (siehe Kapitel 4.4) anzufertigen.

Beispiel:

```
/*****************************************************************************
* Modulname                     : test.c
*
* Thema                         : Zusammenstellung von Testfunktionen
* Liste der externen Funktionen : test1, test2
* Liste der externen Makros     : test3
* Liste der benötigten Module   : -
* Betriebssystem                : MS-DOS 5.0
* Compiler                      : Microsoft C/C++ 7.0
* Compiler-Optionen             : /Za /F C000 /Gt3 /W4
* Author                        : Egon Mueller
* Version                       : 1.0
* Letzte Änderung               : 07.06.1994
*****************************************************************************/
```

Zu jedem Modul ist eine zugehörige Deklarationsdatei zu erstellen. Die Deklarationsdatei hat den gleichen Namen wie das Modul, jedoch erhält sie die Endung ".h". Die Deklarationsdatei enthält alle externen Makros und Funktionsprototypen der externen Funktionen des Moduls. Weiterhin erhält die Deklarationsdatei literale Ersetzungen (#define) und Typvereinbarungen soweit sie für den Benutzer des Moduls von Bedeutung sind. Die Deklarationsdatei ist so anzulegen, daß ein Mehrfachaufruf nicht zu Fehlern führt. Ebenso ist sicherzustellen, daß durch unterschiedliche Deklarationsdateien Typen nicht mehrfach definiert werden. Die Deklarationsdatei enthält einen Kommentarkopf, der auf das zugehörige Modul hinweist.

Beispiel:

```
/*****************************************************************************
* database.h - Deklarationsdatei fuer das Modul database.c
*****************************************************************************/

/* Vorsorgen fuer den Fall, dass diese Deklarationsdatei mehrfach in
 * einen Quelltext aufgenommen wird.
 */
#ifndef DATABASE_H

/* Definition des Typs boolean */
#ifndef BOOL
typedef enum {FALSE, TRUE} BOOL;
#endif /* BOOL */

/* Deklaration der Funktionen  */
unsigned long ReadFile(
               char *fileName,
               char *tempFileName
);

#define DATABASE_H
#endif
/* DATABASE_H */
```

Wenn Module einem Benutzer als (übersetzte) Objektdateien zur Verfügung gestellt werden, so sind die Funktionsköpfe aus den Moduldateien "*.c" in die entsprechenden Deklarationsdateien "*.h" hinter die Prototypen der Funktionen zu kopieren.

4.4. CASE und Software-Dokumentation

Computer Aided Software Engineering (CASE) ist bei der professionellen Erstellung von Software nicht mehr weg zu denken. CASE hilft dem DV-Spezialisten und Programmierer die Software logisch und effizient strukturiert zu entwerfen und zu spezifizieren [10]. Eine richtige CASE-Nutzung ermöglicht, erheblich mehr Zeit bei der Codierung einzusparen, als zunächst für die Findung der optimalen Programmstruktur aufgewandt wurde. Wird eine Software gemäß Spezifikation erstellt, so können die Spezifikationsdokumente ebenfalls als Dokumentation der fertigen Software dienen. Software wird jedoch oft nicht bis ins kleinste Detail spezifiziert, weil das Wissen über diese Details in der Spezifikationsphase noch gar nicht vorhanden ist. Dann ist es wichtig, nach dem Abschluß der Programmierung, die Software so zu dokumentieren, daß eine andere Person sich ohne große Schwierigkeiten mit dem Vorliegenden vertraut machen kann. Die Grenze zwischen CASE und Software-Dokumentation ist also fließend.

Der Aufwand, den eine Software-Entwicklung erfordert, ist erheblich. Weitaus mehr Zeit, Kosten und Energie ist aber anschließend für die Wartung und Erweiterung erforderlich. Aus wirtschaftlichen Gründen ist zudem die Wiederverwendung von Software von Bedeutung. Jedes Modul sollte daher so allgemein gehalten werden, daß es potentiell wiederverwendbar ist. Eine ganz wesentliche Voraussetzung für wiederverwendbare Software mit langfristiger Nutzung ist aber gute Software-Dokumentation.

CASE

CASE, das sind in erster Linien einmal Methoden. Heute sind viele dieser Methoden bereits auf Rechnern programmiert worden, so daß Software automatisch mit Software entworfen werden kann. Für eine Softwareerstellung in kleinem Umfang reicht es aus, die Methoden des CASE "per Hand" zu nutzen.

Sind umfangreiche Datenmengen zu bewältigen, so sind Datenflußdiagramme (data flow diagrams) nützlich. Das Datenflußdiagramm kann der Ausgangspunkt sein für eine Strukturierung des Programms in Module und Funktionen (siehe [10] S. 100). Das Ergebnis dieser Strukturierung wird in der Form der "Structure Charts" (Aufrufdiagramm) dargestellt.

Structure Charts stellen die hierarchische Gliederung von Funktionen und Modulen dar. Es gibt verschiedene Möglichkeiten Structure Charts zu zeichnen. Wir wollen die "Jackson Structure Charts" anwenden. Werden Funktionen der Reihe nach aufgerufen, so werden sie auch einfach der Reihe nach aufgelistet. Eine Auswahl aus verschiedenen Funktionen wird durch eine "0" hinter dem Funktionsnamen dargestellt; eine Iteration durch "*" und eine rekursive Funktion durch "R". (Das letzte Zeichen geht schon über den Rahmen der eigentlichen Jackson Structure Charts hinaus). Zur Erstellung der Jackson Structure Charts sind die am Arbeitsbereich vorhandenen Anwendungen zu verwenden.

Software-Dokumentation

Im Unterschied zur Programm-Kommentierung durch Zeilenkommentare und Blockkommentare (Kapitel 2.4), Funktionsköpfe (Kapitel 4.2) und Modulköpfe (Kapitel 4.3), die im Quellcode erfolgt, handelt es sich bei der Software-Dokumentation um die Erstellung eines Dokumentes über die erstellte Software.

Eine Software-Dokumentation sollte mindestens folgende Bestandteile haben:

1. Verzeichnis der Module (Quelldateien)
2. Verzeichnis der Deklarationsdateien (Include-Dateien)

2.1 Eigene Deklarationsdateien

2.2 Erforderliche Deklarationsdateien für die aufgerufenen C-Bibliotheksfunktionen

3. Hierarchiediagramm (Structure Chart)
4. Aufrufdiagramm (Jackson Structure Chart)

Das Hierarchiediagramm stellt graphisch Import-/Exportbeziehungen zwischen Quelldateien (Modulen) eines Programmes in einer baumartigen Graphik dar.

Das Aufrufdiagramm ist identisch mit der Jackson Structure Chart. Zusätzlich zu den Zeichen '0', '*' und 'R' wird für die Software-Dokumentation hinter dem Funktionsnamen noch die Quell- oder Deklarationsdatei angegeben, in der die Funktion abgespeichert ist. Bei großen Programmen reicht es aus, das Aufrufdiagramm hinunter bis auf die Ebene der externen Funktionen zu zeichnen. Dadurch wird das Aufrufdiagramm kleiner und übersichtlicher. Die Details können dann den Aufrufdiagrammen zu den einzelnen Modulen entnommen werden. Besonders leicht läßt sich ein Aufrufdiagramm in der folgenden Form erstellen.[8]

---

[8] Mit Microsoft C/C++ 7.0 läßt sich eine Datenbasis erstellen, die in der Programmer's Workbench genutzt werden kann um eine Datei zu erstellen, die als Basis für ein Aufrufdiagramm verwendet werden kann. Der Ablauf zur Erstellung dieser Datei ist folgendermaßen:

1.) Mit
cl /Zs /Fr *module*.c
das Browse-Info-File *module*.sbr erstellen.
(/Zs führt dabei einen Syntyxcheck durch.)

2.) Mit
BSCMAKE *module*.sbr
die Browse-Database erzeugen.

3.) In der Programmer's Workbench mit
Browse -> Open Custom...
Use Custom Database...
module.bsc als Custom Database wählen.

4.) Mit
Browse -> Call Tree
eine Datei erstellen und so editieren, daß ein Aufrufdiagramm entsteht.

Beispiel:

```
main.......................... test.c
 ├─ReadFile................... test1.c
 │  ├─CheckFile............... test1.c
 │  │  └─GetToken............. test.h
 │  ├─GetToken................ test.h
 │  └─ReadList (R)............ test1.c
 │     └─GetToken............. test.h
 └─WriteFile.................. test2.c
```

Weiterer Bestandteile einer Software-Dokumentation könnte sein:

5. Typen-Dokumentation

Die Typen-Dokumentation führt zu jedem definierten Type auf:

- o Typdefinition
- o Definitionsort (Datei, Zeilennummer)
- o Namen der Unterprogramme, in deren Schnittstelle oder Anweisungsteil der Type benutzt wird
- o Namen der Daten, die den Typ haben.

Vergleichbar mit der Entwicklung beim CASE, wo Methoden auf Rechnern implementiert wurden, gibt es ebenfalls Programmsysteme, die eine automatische Software-Dokumentation ermöglichen. Bei kleinen Projekten ist jedoch zu überlegen, ob die Dokumentation nicht ebenso schnell von Hand erstellt werden kann.

5. Spezielle Programmierhinweise

5.1. System zur Verwaltung von Fehlermeldungen

In der Regel sollte jede Funktion einen Fehlercode zurück liefern. "0" bedeutet, daß kein Fehler aufgetreten ist. Der Fehlercode und die Funktion sind vom Typ "unsigned long". Der Aufruf einer Funktion erfolgt in der Form:

```
error = function( ... );
```

Ein Fehler in einer Funktion führt zum Rücksprung und der Übergabe eines entsprechenden Fehlercodes. In einem einfachen Fall kann das z.B. so aussehen:

```
if(temp_file == NULL)
      return(3);
```

Hat der Fehler seinen Ursprung in einer Fehlermeldung aus einer untergeordneten Funktion, so wird der Fehlercode der untergeordneten Funktion zusammen mit dem Fehlercode der aktuellen Funktion beim Rücksprung übergeben:

```
if(error = CheckFile(file))
      return(error*10 + 2);
```

Nehmen wir an, daß der Fehlercode aus CheckFile den Wert 3 hat, so liefert die Funktion den Wert 32 zurück. Beschränkt man sich in jedem Unterprogramm auf die Fehlercodes von 1 bis 9, so kann man die Fehlercodes der einzelnen Funktionen von der Funktion unten in der Hierarchie bis zur Funktion oben in der Hierarchie aus den Ziffern des Fehlercodes von links nach rechts ablesen.

Die Fehlercodes jeder Bibliotheksfunktion aus C sollten überprüft werden. Wenn man dies mit geringem Programmieraufwand erreichen will, so kann z.B. folgendermaßen vorgegangen werden:

```
if(fprintf(temp_file,"\n") < 0)
      return(4);
```

5.2. Dynamische Speicherverwaltung für mehrdimensionale Felder

Für Felder, deren Größe erst zur Programmausführung ermittelt wird, sollte nur so viel Speicherplatz reserviert werden, wie erforderlich ist. Die folgenden Programmzeilen sollen das Vorgehen erläutern. (Achtung: Es wurde festgestellt, daß diese Methode auf dem PC nur für Felder funktioniert, deren Dimension maximal 4 ist!).

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

    ...

    int i,j,k,l;
    int p=5;
    int q=3;
    int r=2;
    int s=6;
    char ****a;

    a = (char ****)malloc((p+1)*sizeof(char***));
    if(a == NULL) return(1);

    for (i=0; i<=p; i++) {
       a[i] = (char ***)malloc((q+1)*sizeof(char**));
       if(a[i] == NULL) return(1);

       for (j=0; j<=q; j++) {
          a[i][j] = (char **)malloc((r+1)*sizeof(char*));
          if(a[i][j] == NULL) return(1);

          for (k=0; k<=r; k++)   {
             a[i][j][k] = (char *)malloc((s+1)*sizeof(char));
             if(a[i][j][k] == NULL) return(1);
          }
       }
    }

    for (i=0; i<=p; i++) {
       for (j=0; j<=q; j++) {
          for (k=0; k<=r; k++)   {
             for (l=0; l<=s; l++)   {
                a[i][j][k][l] = 'c';
             }
          }
       }
    }
    printf("Wert aus a: %c\n", a[p][q][r][s]);

    for (i=0; i<=p; i++)
       for (j=0; j<=q; j++)
          for (k=0; k<=r; k++)
             free (a[i][j][k]);

    for (i=0; i<=p; i++)
       for (j=0; j<=q; j++)
          free (a[i][j]);

    for (i=0; i<=p; i++)
       free (a[i]);

    free (a);
    return(0);
}
```

6. Hinweise zur Programmierung mit Microsoft C/C++

6.1. Speicherverwaltung

Bei der Programmierung auf dem PC stößt man schnell an die Grenzen des Speicherbereichs, wenn ein Verständnis über die Zusammenhänge nicht gegeben ist. Arbeitet man mit den Standardeinstellungen, so wird das Programm ab einer bestimmten Größe mit der Fehlermeldung "Stack Overflow" abbrechen. Was ist geschehen? Alle Daten haben sich in einen kleinen Teil des zur Verfügung stehenden Speicherplatzes gedrängt, der nun voll ist. Was muß unternommen werden?

1.) Alles was in andere Speicherbereiche (den far heap) verlagert werden kann muß ausgelagert werden.
2.) Für den begrenzten Speicherbereich (den near heap) muß eine optimale Aufteilung gefunden werden.

Mit geschickter Programmierung kann ein Programm so optimiert werden, daß der Speicherplatz ausreicht und das Programm klein und schnell wird. Hier soll ohne Rücksicht auf die letzten Feinheiten der Programmoptimierung beschrieben werden, was zu tun ist, damit ein Programm "auf jeden Fall läuft". Einzelheiten über die hier angesprochenen Compiler-Optionen können [11] entnommen werden.

Zu 1.)
Der Stack hat eine maximale Größe von 64 K. Diese Größe kann dem Stack mit der Option /F FFFE des Compilers zugewiesen werden. Mehr geht nicht. Daher soll alles was nicht in den Stack 'rein muß auch woanders hin geschrieben werden: in den Far Heap. Das geht so:

a) Es wird mit der Option /AL das große Speichermodell gewählt.

b) Felder und Strukturen erhalten ihren Speicherplatz mit **malloc**. (Siehe Kapitel 2.6).

c) Variablen (initialisierte und nicht initialisierte Variablen), die bei jedem Aufruf der Funktion auf dem gleichen Speicherplatz gespeichert werden können werden **static** deklariert.

d) Alle **static** deklarierten Variablen größer als 3 Byte werden mit der Compiler-Option /Gt3 in den Far Heap gebracht.

zu 2.)
Im near heap müssen noch mehr Daten untergebracht werden als nur der Stack - auch dann, wenn schon alle Maßnahmen unter 1.) getroffen wurden. Es kann also vorkommen, daß bei Wahl der Option /F FFFE der Stack ausreichend groß ist, nun aber nicht mehr genug Platz für die anderen Daten des near heap vorhanden ist. Man wird also individuell die optimale Größe für den Stack ermitteln müssen. Dabei wird die Größe des Stack hinter der Option /F in einer hexadezimalen Zahl angegeben. Als

ein Ausgangspunkt für eigene Versuche für die Ermittlung der optimalen Stackgröße soll hier der Wert C000 angegeben werden.

Mit den Maßnahmen unter 1.) und 2.) können die 640 K für DOS-Programme voll genutzt werden. Genügt das jedoch noch nicht, so kann der ganze RAM mit einem QuickWin-Programm genutzt werden. QuickWin-Programme erfordern ("normalerweise") keine Änderung am Quellcode. Es ist lediglich mit der Option /Mq zu complieren.

Bei dem oben beschriebenen Vorgehen kann mit einem einzigen Aufruf von **malloc** nur maximal ein Speicherbereich von 64 K angefordert werden. Das bedeutet jedoch nicht, daß mehrdimensionale Felder auf 64 K beschränkt sind. Bei dem in Kapitel 5.2 beschriebenem Vorgehen zur Allokierung von Speicherplatz kann ein Feld bis zur Auffüllung des RAM beliebig groß werden, solange die einzelnen Aufrufe von **malloc** nicht mehr als 64 K anfordern.

6.2. Compiler-Optionen

Als Standard Compiler-Optionen für die Speicherbereichsverwaltung wurden in Kapitel 6.1 vorgeschlagen: `/F C000 /AL /Gt3`

Für eine Überprüfung des Quelltextes auf eine Programmierung nach ANSI-C ist erforderlich:

`/Za`

Einen weitereren Schutz vor Programmierfehlern erhält man, wenn man sich auch die Warnungen anzeigen läßt. Das "Warning Level" 4 ergibt die größte Anzahl von Hinweisen. Dieses Level weist auch auf weitere Merkmale einer nicht ANSI-gerechten Programmierung hin. Unter Anwendung eines sauberen Programmierstils sollte es möglich sein, alle Warnungen des Levels 4 zu eliminieren. Die entsprechende Option ist:

`/W4`

6.3. Makefiles

Eine professionelle Programmierung wird von den Möglichkeiten der Makefiles gebrauch machen. Um den Einstieg in den Umgang mit den Microsoft-Makefiles zu erleichtern, sollen nachfolgend Beispiele gegeben werden, die sich an der modularen Programmstruktur orientieren, wie sie in dieser Programmieranleitung vorgeschlagen wurde: Das Hauptprogramm ruft verschiedene Module (Funktionen) auf. Zu jedem Modul gehört eine Deklarationsdatei. Die folgenden Beispiele können mit den Kapiteln 13, 14, 18 und 19 aus [11] nachvollzogen werden.

Beispiel 1: Übersetzen und binden von zwei Modulen mittels eines Hauptprogramms

Die Aktionen des Makefiles sample.mak werden mit folgendem Befehl gestartet:

```
nmake /F sample.mak
```

Datei: sample.mak

```
# Dieses makefile bildet sample.exe aus den zwei Modulen sample1.c
# und sample2.c, die im Hauptprogramm sample.c aufgerufen werden.
all: sample.exe
sample.exe : sample.obj sample1.obj sample2.obj
      link sample+sample1+sample2;
sample.obj: sample.c sample1.h sample2.h
      CL /c /Za /F C000 /Gt3 /AL /W4 sample.c
sample1.obj: sample1.c sample1.h
      CL /c /Za /F C000 /Gt3 /AL /W4 sample1.c
sample2.obj: sample2.c sample2.h
      CL /c /Za /F C000 /Gt3 /AL /W4 sample2.c
```

Datei: sample.c

```
#include "sample1.h"
#include "sample2.h"
void main(void)
{
      sample1();
      sample2();
}
```

Datei: sample1.c

```
#include "sample1.h"
#include <stdio.h>
void sample1(void)
{
      printf("sample1\n");
}
```

Datei: sample1.h

```
void sample1(void);
```

Datei: sample2.c

```
#include "sample2.h"
#include <stdio.h>
void sample2(void)
{
      printf("sample2\n");
}
```

Datei: sample2.h

```
void sample2(void);
```

Beispiel 2: Übersetzen und Binden von zwei Modulen als Bibliothek mittels eines Hauptprogramms. Das Makefile libsamp.mak nutzt die Dateien aus Beipiel 1.

Die Aktionen des Makefiles libsamp.mak werden mit folgendem Befehl gestartet:

```
nmake /F libsamp.mak
```

Datei: libsamp.mak

```
# Dieses makefile bildet libsamp.exe aus zwei Modulen
# sample1.c und sample2.c. Aus beiden Modulen wird zuerst eine
# Bibliothek generiert. Das Hauptprogramm sample.c wird dann mit
# der Bibliothek zusammen gebunden.
all: libsamp.exe
libsamp.exe: sample.obj sample.lib
      link sample.obj,libsamp.exe,,sample.lib,;
sample.lib: sample1.obj sample2.obj
      lib sample.lib +sample1.obj +sample2.obj;
sample.obj: sample.c sample1.h sample2.h
      CL /c /Za /F C000 /Gt3 /AL /W4 sample.c
sample1.obj: sample1.c sample1.h
      CL /c /Za /F C000 /Gt3 /AL /W4 sample1.c
sample2.obj: sample2.c sample2.h
      CL /c /Za /F C000 /Gt3 /AL /W4 sample2.c
```

Im Anhang sind Beispiele von Makefiles für Mircrosoft C/C++ 7.0 aufgelistet. Diese Makefiles können als Ausgangspunkt für eigene Makefiles größerer Projekte dienen. Wiederum ist das Handbuch "Environment and Tools" [11] erforderlich, um die Einzelheiten der Makefiles nachzuvollziehen.

7. Literaturverzeichnis

[1] Wix, B.; Balzert, H.: "Softwarewartung", BI Wissenschaftsverlag, 1988.

[2] Kernighan, B. W.; Ritchie, D. M.: "Programmieren in C", 2. Ausgabe - ANSI C, Hanser, Prentice-Hall International, 1990.

[3] Regionales Rechenzentrum für Nierdersachsen: "Die Programmiersprache C - Ein Nachschlagewerk", RRZN-Schlüssel: SPR.C1, 1.Auflage, 1991.

[4] Institut für angewandte Mikroelektronik: "Empfehlungen zur Implementierung von Programmen", Version 2.0, Braunschweig, 1992, unveröffentlicht.

[5] Institut für angewandte Mikroelektronik: "Standardisierte Softwaredokumentation", IAM-interne Richtlinie, Version 1.1, Dok.-Nr.: IAM-89925514D-V1.1, Braunschweig, 1988, unveröffentlicht.

[6] Schnupp, Peter: "Von C zu C: Problemlos Portieren", Hanser, München, 1990.

[7] Microsoft: "C/C++ Version 7.0, Programming Techniques", Microsoft Corporation, 1991.

[8] DIN 13 304, "Darstellung von Formelzeichen auf Einzeilendruckern und Datensichtgeräten".

[9] Engeln-Müllges, G.; Reuter, F.: "Numerische Mathematik für Ingenieure, mit C-Programmen", BI Wissenschaftsverlag, 1987.

[10] Fisher, Alan, S.: "CASE, Using Software Development Tools", Wiley, 1988.

[11] Microsoft: "C/C++ Version 7.0, "Environment and Tools", Microsoft Corporation, 1991.

Anhang

An dieser Stelle werden Makefiles für Mircrosoft C/C++ 7.0 aufgelistet. Diese Makefiles können als Ausgangspunkt für eigene Makefiles größerer Projekte dienen. Wiederum ist das Handbuch "Environment and Tools" [11] erforderlich, um die Einzelheiten der Makefiles nachzuvollziehen.

```
#############################################################################
#
# Makefile fuer DOS Anwendung mit
# Microsoft C/C++ 7.0, NMAKE
#
# Anwendung         : datatest
# Author            : Dieter Scholz
# Letzte Aenderung  : 19.07.94
#
#############################################################################

# C - Directory
CBASE = d:\c700

# Info zu Compiler-Optionen:
# /I    : search directory for include files
# /c    : compile without linking
# /AL   : large memory model
# /Gt3  : allocate new data segment for data >= 3 byte
# /Za   : disable language extensions (ANSI-C check)
# /W?   : warning Level; 0=no warning, 1=low level, 4=high level
# /O?   : optimization (see: "Environment and Tools")
# /Od   : disable optimization
# /Zi   : produce CodeView information
# /f    : fast compile
# /DMSW : #define MSW
# /Yc   : create precompiled header
# /Yu   : use precompiled header
# /f    : fast compile

# Compiler-Optionen, Standard:
CFLAGS = /I$(CBASE)\include /c /AL /F C000 /Gt3 /Za /W4

# Compiler-Optionen, Optimierung:
# CFLAGS = /I$(CBASE)\include /c /AL /F C000 /Gt3 /Za /W4 /Og /Oe /Oi /Oo /Ot

# Compiler-Optionen, Debugging:
# CFLAGS = /I$(CBASE)\include /f /c /AL /F C000 /Gt3 /Za /W4 /Od /Zi

# Info zu Link-Optionen:
# /NOE      : do not perform extended dictionary search
# /NOD      : no default library search
# /NOI      : preserve case in identifiers
# /FARCALL  : optimize far calls
# /PACK     : optimize code by packing (not recommended for Windows)
# /CO       : include debug information
# /BATCH    : suppress prompting for libraries

# Link-Optionen, Standard:
LFLAGS = /BATCH /NOI

# Link-Optionen, Optimierung:
# LFLAGS = /BATCH /NOI /FARCALL /PACK

# Link-Optionen, Debugging:
# LFLAGS = /BATCH /NOI /CO

# Name der Anwendung:
APPLICATION = datatest

# Eigene Module als *.obj-Dateien:     (Auflistung getrennt durch "Blank"):
OBJECTS = $(APPLICATION).obj

# Eigene Module als *.lib-Dateien:     (Auflistung getrennt durch "Blank"):
LIBRARIES = database.lib

#############################################################################
```

```
# Erstellung der EXE:
# NMAKE aufrufen mit   nmake /F anwendung.mak
#
all : $(APPLICATION).exe

# Linken:
# Dazu Erstellen einer Datei "linkargs" mit den Argumenten fuer den Linker
# und Aufruf des Linkers mit dem Inhalt dieser Datei.
# (Grund: DOS-Zeile max. 128 Zeichen)
#
$(APPLICATION).exe : $(OBJECTS) $(LIBRARIES)
        link $(LFLAGS) @<<linkargs
$(OBJECTS)
$(APPLICATION).exe
nul.map
$(LIBRARIES)
nul.def
<<KEEP

# Uebersetzen der Applikation:
#
$(APPLICATION).obj : $(APPLICATION).c database.h
        cl $(CFLAGS) $(APPLICATION).c

# Erstellen der Bibliothek fuer die Manipulation der Datenbasis
#
database.lib : database.obj
        lib database.lib +database.obj;

# Uebersetzen des Moduls zur Manipulation der Datenbasis
database.obj : database.c database.h
        cl $(CFLAGS) database.c

###############################################################################

# Erstellung der Browse-Info-Files:
# NMAKE aufrufen mit   nmake /F anwendung.mak browse
browse : $(APPLICATION).sbr database.sbr
        bscmake $(APPLICATION).sbr database.sbr

$(APPLICATION).sbr : $(APPLICATION).c
        cl /c /Zs /Fr /I$(CBASE)\include $(APPLICATION).c
        cl /c /Zs /Fr /I$(CBASE)\include database.c

###############################################################################
```

Das entsprechende Makefile für QuickWin unterscheidet sich durch:

1.) die zusätzliche Angabe der Compiler-Option /Mq
2.) der Angabe der QuickWin-Libraries
3.) der Angabe eines Definition Files für QuickWin (siehe unten)
4.) dem Wegfall der Compiler-Option für die Stackgröße (die Stackgröße wird für Windows-Programme in der Definitionsdatei festgelegt).

```
#############################################################################
#
# Makefile fuer QuickWin Anwendung mit
# Microsoft C/C++ 7.0, NMAKE
#
# Anwendung         : datatest
# Author            : Dieter Scholz
# Letzte Aenderung  : 23.07.94
#
#############################################################################

# C - Directory
CBASE = d:\c700

# Info zu Compiler-Optionen:
# /I    : search directory for include files
# /c    : compile without linking
# /AL   : large memory model
# /Gt3  : allocate new data segment for data >= 3 byte
# /Za   : disable language extensions (ANSI-C check)
# /W?   : warning Level; 0=no warning, 1=low level, 4=high level
# /O?   : optimization (see: "Environment and Tools")
# /Od   : disable optimization
# /Zi   : produce CodeView information
# /f    : fast compile
# /DMSW : #define MSW
# /Yc   : create precompiled header
# /Yu   : use precompiled header
# /f    : fast compile
# /Mq   : QuickWin program

# Compiler-Optionen, Standard:
CFLAGS = /I$(CBASE)\include /c /Mq /AL /Gt3 /Za /W4

# Compiler-Optionen, Optimierung:
# CFLAGS = /I$(CBASE)\include /c /Mq /AL /Gt3 /Za /W4 /Og /Oe /Oi /Oo /Ot

# Compiler-Optionen, Debugging:
# CFLAGS = /I$(CBASE)\include /f /c /Mq /AL /Gt3 /Za /W4 /Od /Zi

# Info zu Link-Optionen:
# /NOE      : do not perform extended dictionary search
# /NOD      : no default library search
# /NOI      : preserve case in identifiers
# /FARCALL  : optimize far calls
# /PACK     : optimize code by packing (not recommended for Windows)
# /CO       : include debug information
# /BATCH    : suppress prompting for libraries

# Link-Optionen, Standard:
LFLAGS = /BATCH /NOI

# Link-Optionen, Optimierung:
# LFLAGS = /BATCH /NOI /FARCALL /PACK

# Link-Optionen, Debugging:
# LFLAGS = /BATCH /NOI /CO

# Name der Anwendung:
APPLICATION = datatest

# Eigene Module als *.obj-Dateien:     (Auflistung getrennt durch "Blank"):
OBJECTS = $(APPLICATION).obj

# Eigene Module als *.lib-Dateien:     (Auflistung getrennt durch "Blank"):
LIBRARIES = database.lib

#############################################################################
```

```
# Erstellung der EXE:
# NMAKE aufrufen mit   nmake /F anwendung.mak
#
all : $(APPLICATION).exe

# Linken:
# Dazu Erstellen einer Datei "linkargs" mit den Argumenten fuer den Linker
# und Aufruf des Linkers mit dem Inhalt dieser Datei.
# (Grund: DOS-Zeile max. 128 Zeichen)
#
$(APPLICATION).exe : $(OBJECTS) $(LIBRARIES)
        link $(LFLAGS) @<<linkargs
$(OBJECTS)
$(APPLICATION).exe
nul.map
$(LIBRARIES) /NOD:LLIBCEW LLIBCEWQ.LIB LIBW
quickwin.def
<<KEEP

# Uebersetzen der Applikation:
#
$(APPLICATION).obj : $(APPLICATION).c database.h
        cl $(CFLAGS) $(APPLICATION).c

# Erstellen der Bibliothek fuer die Manipulation der Datenbasis
#
database.lib : database.obj
        lib database.lib +database.obj;

# Uebersetzen des Moduls zur Manipulation der Datenbasis
database.obj : database.c database.h
        cl $(CFLAGS) database.c

##############################################################################

# Erstellung der Browse-Info-Files:
# NMAKE aufrufen mit   nmake /F anwendung.mak browse
browse : $(APPLICATION).sbr database.sbr
        bscmake $(APPLICATION).sbr database.sbr

$(APPLICATION).sbr : $(APPLICATION).c
        cl /c /Zs /Fr /I$(CBASE)\include $(APPLICATION).c
        cl /c /Zs /Fr /I$(CBASE)\include database.c

##############################################################################
```

Das Definitions File QUICKWIN.DEF hat dabei folgendes Aussehen:

```
; Anwendung benennen
NAME            ALLGEMEINE_QWIN_ANWENDUNG

; Betriebssystems bestimmen
EXETYPE         WINDOWS

; Text in die EXE-Datei einbinden
DESCRIPTION     'QUICKWINDOWS: FST, TUHH'

; DOS-ausfuehrbares Programm einbinden (Kommentar bei Start von DOS)
STUB            'STUB.EXE'

; Default-Attribute setzen fuer Code und Daten
CODE            PRELOAD MOVEABLE DISCARDABLE
DATA            PRELOAD MULTIPLE

; Stacksize vereinbaren (Angabe in Byte)
STACKSIZE       42667

; Vereinbaren wieviel lokaler Heap in der DGROUP vorhanden sein soll
HEAPSIZE         1024
```