

(C-)Programmier-Konventionen, Tipps und Werkzeuge

Version 3.2 — 6.7.2009

© 1998-2009 T. Birnthal, OSTC GmbH

eMail: tb@ostc.de

Web: www.ostc.de

„Ein gut geschriebenes Programm ist sinnvoll strukturiert, hat ein sauberes Layout, verwendet sinnvolle Namen, ist ausführlich kommentiert und verwendet Konstrukte der Sprache derart, daß maximale Sicherheit und Lesbarkeit des Programms erreicht werden. Die Erstellung eines solchen Programms erfordert vom Programmierer Sorgfalt, Disziplin und ein gutes Stück handwerklichen Stolz.“

Ian Sommerville, „Software Engineering“, Addison-Wesley Verlag 1987

Vorwort

Das Zitat auf der Titelseite soll ganz allgemein als Motto für die Erstellung (und Wartung) von Programmen dienen.

Etwas anders ausgedrückt soll beim Programmieren das Konzept des **Literate Programming** — ein von Donald E. Knuth eingeführter Begriff — verfolgt werden. Dies bedeutet, daß ein Programm **für alle Programmierer** so gut lesbar und verständlich sein soll, wie dies ein Buch für seine Leser ist. Dokumentation und Code sollen eine Einheit bilden, aus der wahlweise der zu übersetzende Code, eine (gedruckte oder elektronische) Schnittstellen-Dokumentation, eine (vollständige) gedruckte Programm-Dokumentation erstellt werden kann.

Dieses Dokument beschreibt Konventionen für die Programmierung in der Programmiersprache C (entsprechend dem ANSI/ISO-C-Standard). **Konventionen** sind Regeln, an die sich der Programmierer freiwillig hält, da ihre Einhaltung nicht vom Compiler erzwungen wird. Die Anwendung dieser Regeln soll dazu beitragen, die Qualität der entwickelten Software hinsichtlich der Kriterien

- Lesbarkeit
- Überprüfbarkeit
- Zuverlässigkeit
- Wartbarkeit
- Erweiterbarkeit
- Wiederverwendbarkeit

zu erhöhen. Die Regeln werden jeweils begründet, sofern sie nicht sowieso unmittelbar einsichtig sind. Die aufgeführten Konventionen stellen eine Synthese der Erfahrungen der Firma HEITEC, des C-Experten Andrew Koenig, Mitarbeitern der Abteilung SW-Entwicklung meines ehemaligen Arbeitgebers GfK AG, Mitarbeitern der Georg-Simon-Ohm-Fachhochschule Nürnberg und vieler anderer Personen im Umgang mit der Programmiersprache C dar. Ihre Beachtung hilft einen Großteil von typischen (C-)„Programmiersünden“ zu vermeiden.

Der Autor ist dankbar für Korrekturempfehlungen und Ergänzungsvorschläge.

Thomas Birnthaler, OSTC GmbH (E-Mail: tb@ostc.de)

Inhaltsverzeichnis

1	Grundregeln	1
2	Dokumentation	4
2.1	Allgemein	4
2.2	Versionierung	6
3	Programmtest	7
4	Namensvergabe	9
5	Übersicht erhalten	11
5.1	C-Code	11
5.2	Präprozessor	13
6	Quellcode-Formatierung	15
6.1	Einrückung und Tabulatoren	15
6.2	Leerzeichen	17
6.3	Konstanten und Variablen	17
6.4	Funktionen	18
6.5	Kommentare	19
7	Spezielle Programmelemente	20
7.1	Konstanten und Makros	20
7.2	Include-Dateien	22
7.3	Datentypen und Casts	24
7.4	Dateizugriff	26
7.5	Fließkommazahlen	28
7.6	Dynamischer Speicher	29
8	Häufige Fehler	30
9	Optimierung	37
10	Sonstiges	38
11	Vorschläge für Funktions- und Variablennamen	39
11.1	Generelle Konventionen	39
11.2	Vorschläge für Tätigkeitswörter von Funktionen	39
11.3	Vorschläge für Eigenschaftswörter von Funktionen	40
11.4	Zu vermeidende Funktionsnamen	41
11.5	Vorschläge für Variablennamen (oder Teilen davon)	41
11.6	Zu vermeidende Variablennamen	42
12	Erweiterung von C	43
13	Der Datentyp Boolean	45

14 Rekursive Datentypen	46
15 Variable Argumentlisten	47
16 Suchen und Sortieren	48
17 Datentypen und ungarische Notation	49
18 Vorrangtabelle	50
19 Ungewöhnliche Operatoren	51
20 Operatoren mit problematischer Priorität	53
21 Operatoren mit fixer Auswertungsreihenfolge	53
22 Auswertungs-Reihenfolge beeinflussen	54
23 C-Deklarationen	55
23.1 Lesen von C-Deklarationen	55
23.2 Schreiben von C-Deklarationen	57
23.3 Einschränkungen	59
23.4 Typdefinitionen und Casts	59
24 RCS	61
24.1 Übersicht	61
24.2 Beschreibung	63
24.3 Beispiel	64
25 make und Makefiles	67
25.1 Beschreibung	67
25.2 Ablauf	70
25.3 Beispiel	71
26 makedepend	75
27 texdoc	79
28 Sonstige Tools	80

1 Grundregeln

- @ Der **Programmmentwurf** (und falls vorhanden das **Datenbank-Design**) ist vor der Realisierung einem unabhängigen Programmierer (Consultant) zu erklären und mit ihm durchzusprechen.
- @ Der **Programmcode** (oder zumindest seine Grobstruktur) ist während und nach dem Abschluß der Programmierarbeiten mit einem unabhängigen Programmierer (Consultant) zur Kontrolle durchzugehen (sogenannter **Review**: *vier Augen sehen mehr als zwei*).
- Um den **Aufbau einer C-Quelltext-Datei** übersichtlich zu gestalten, sind die Anweisungen (soweit möglich) gemäß ihrer Art folgendermaßen zu sortieren:
 - Datei-Kommentarkopf (**Header**) mit folgenden Angaben:
 - * Modulname
 - * Version
 - * Autor
 - * Projekt
 - * Copyright-Vermerk
 - * Kurzbeschreibung des Inhalts
 - * Änderungsliste mit folgenden Angaben je Änderung:
Datum/Autor/Grund der Änderung (*entfällt bei der Verwendung von RCS*)
 - #include-Anweisungen < . . . > für System-Header-Dateien
 - #include-Anweisungen " . . . " für eigene Header-Dateien
 - #define-Anweisungen (Konstanten und Makros)
 - Definition von Datentypen (`typedef`)
 - Definition globaler Konstanten (`const`)
 - Definition globaler Variablen (`extern`)
 - Definition modullokalen Konstanten (`static const`)
 - Definition modullokalen Variablen (`static`)
 - Deklaration modullokalen Funktionen (`static`)
 - Definition globaler und modullokalen Funktionen
(sortiert nach Aufrufhierarchie, Themen oder alphabetisch)

Werden Programmelemente nicht verwendet, so kann der betreffende Abschnitt entfallen. Ein Muster für den Aufbau einer C-Datei und einer H-Datei ist im Anhang zu finden.

- Alle Funktionen sind **vollständig zu deklarieren** (gemäß ANSI-Standard, nicht gemäß K&R-Standard), nur so ist ein automatischer Compiler-Check der Funktionsaufrufe und ihrer Argumente möglich.
- Die Deklarationen von **globalen (externen) Variablen und Funktionen** sind in eine eigene Header-Datei auszulagern, diese ist in alle Quell-Dateien zu includen.

- Ausschließlich in einer Quell-Datei genutzte **modullokale Variablen und Funktionen** sind darin als `static` zu deklarieren, sie sind *nicht* in eine externe Header-Datei aufzunehmen.
- @ **Keine überlangen Quelltexte** (mehr als 1000 Zeilen) schreiben. Darin geht zuviel Zeit zum Hin- und Herblättern und Suchen verloren.
Lieber das Programm in mehrere kleinere **Unterdateien** zerlegen, die jeweils logisch zusammengehörende Funktionen enthalten. Die Übersetzung muß dann über ein **Makefile** erfolgen. Der Compiler zahlt diesen etwas erhöhten Verwaltungsaufwand mit kürzeren Recompilationszeiten nach Änderungen zurück.
- @ Keine Funktion über mehr als zwei Seiten (≈ 120 Zeilen) schreiben, sondern in kleinere **Unterfunktionen** zerlegen, da kleine Untereinheiten einfach überschaubarer sind.
Das andere Extrem, für ein paar Programmzeilen bereits eine Unterfunktion einzuführen, ist allerdings auch nicht ratsam (sofern sie nicht *sehr häufig* verwendet wird). Übermäßig viele Funktionen mit ihren Aufrufparametern kann sich keiner merken.
- Quelltexte mit der Zeile `/*{ [(* / beginnen und mit der Zeile /* }]) */` enden lassen. Die meisten Editoren kennen einen Befehl zum Springen zwischen korrespondierenden Klammern unter Berücksichtigung der Klammernhierarchie (z.B. im `vi` per `%`). Klammerungsfehler können so über die ganze Datei hinweg sehr einfach lokalisiert werden.
- Sollen die gleichen Quellen unter mehr als einem Betriebssystem verwendet werden, dann nur **ASCII-Zeichen**, aber keine Sonderzeichen (äöüÄÖÜß. . .) oder Grafikzeichen außerhalb `0x20–0x7E` (Blank-Tilde) im Quelltext oder Kommentaren verwenden (auch nicht in Form ihres Oktal- oder Hexadezimalcodes).
Nur die ASCII-Zeichen im Bereich `0x20–0x7E` werden zuverlässig von allen Rechnern und Druckern gleich verstanden (außer IBM-Rechner, die den EBCDIC-Code verwenden).
Die portablen Ersatzdarstellungen `\a` für **Alert(Bell)**, `\b` für **Backspace**, `\f` für **Formfeed**, `\n` für **Newline**, `\r` für **Carriage Return**, `\t` für **Horizontal Tab**, `\v` für **Vertical Tab** und `\0` für **Stringende** in Strings und Zeichenkonstanten sind natürlich erlaubt.
- @ Öfter mal einen **Zwischenstand** des Quelltexts im RCS sichern. Änderungen sind dadurch einfach zurücknehmbar bzw. können mit der vorherigen Version verglichen werden (`Id` und `Log` am Dateianfang/ende nicht vergessen).
- @ Grundsätzlich mit **höchster Warnstufe** (`-Wall` oder `/W4`) übersetzen und die Ursache aller Warnungen beheben. Es darf keine einzige (vermeidbare) Warnung mehr produziert werden, da unverständene Warnungen potentielle Fehler sind.
- Die `main`-Funktion ist die erste Funktion am Dateianfang. Sie besteht nur aus Funktionsaufrufen und spiegelt den **Kontrollfluß** des Programms wider.
- @ Jedes als Kommando aufrufbare Programm muß eine vollständige **Usage-Meldung** (d.h. eine Kurzbeschreibung seiner Funktion sowie der Optionen und Parameter) ausgeben, wenn es

1. Ohne Parameter
2. Mit der Option `-h` oder `--help`
3. Mit fehlerhaften Parametern

aufgerufen wird.

- **Die Usage-Meldung ist als Funktion zu realisieren.** Sie gibt im obigen 3. Fall eine (parametrisierte) Fehlermeldung auf `stderr` aus und bricht das Programm grundsätzlich unter Rückgabe eines geeigneten Fehlercodes $n \neq 0$ ab.

```
void
Usage(int errcode, char* fmt, ...)
{
    va_list argp;

    va_start(argp, fmt);
    if (fmt != NULL && !strcmp(fmt, "")) {
        fprintf(stderr, "\n");
        fprintf(stderr, "error(%d): ", errcode);
        fprintf(stderr, fmt, argp);
        fprintf(stderr, "\n");
    }
    va_end(argp);

    fprintf(stderr, "\n");
    fprintf(stderr, "usage: check {OPTIONS} FILE...\n");
    fprintf(stderr, "\n");
    fprintf(stderr, "    Ueberprueft Dateien FILE... auf Fehler.\n");
    fprintf(stderr, "    Optionen sind:\n");
    fprintf(stderr, "    -o FILE  Ausgabedatei [Std: stdout]\n");
    fprintf(stderr, "    -c      Nur Anzahl Fehler ausgeben\n");
    fprintf(stderr, "    -v      Ausfuehrliche Ablaufmeldungen\n");
    fprintf(stderr, "    -p      Ergebnis ausdrucken\n");
    fprintf(stderr, "    -s      Keine Ablaufmeldungen\n");

    exit(errcode);
}
```

@ Tritt in einem Programm ein Fehler auf, so ist eine entsprechende Fehlermeldung auf `stderr` auszugeben und unter Rückgabe eines geeigneten Fehlercodes $n \neq 0$ abzubrechen.

- **Fehlermeldung und Programmabbruch** sind als (parametrisierte) Funktion mit variabler Anzahl von Argumenten zu realisieren.

```
void Error(int errcode, char* fmt, ...) => "error(ERRCODE): ..."
```

- Analoge Funktionen für **Warnungen** (ohne Programmabbruch), **katastrophale Fehler** (mit Programmabbruch unter eventuellem Datenverlust) und **Debugausgaben** (mit verschiedenen Levels) sind ebenfalls sinnvoll.

```
void Warn(char* fmt, ...)           => "warning: ..."
void Fatal(int errcode, char* fmt, ...) => "fatal(ERRCODE): ..."
void Debug(int lvl, char* fmt, ...)   => "debug(LEVEL): ..."
```

2 Dokumentation

2.1 Allgemein

- @ Ein Programm ohne **Benutzerdokumentation** ist ein ständiges Ärgernis, Minimum ist eine vollständige **Usage-Meldung**.

Kein normaler Anwender kann ohne schriftliche oder mündliche Dokumentation auskommen. Die Zeit, die man in die Erstellung der schriftlichen Dokumentation investiert, spart man bei der mündlichen Dokumentation, *multipliziert* mit der Anzahl der Anwender, wieder ein.

Außerdem kann beim Erstellen der Benutzerdokumentation nochmals überprüft werden, ob das Programm wirklich den Vorgaben entspricht. Die Erstellung der Benutzerdokumentation ist also als **QS-Maßnahme** zu betrachten.

- @ Ein Programm ohne **Programmdokumentation** ist nahezu wertlos. Soll der Ersteller selbst nach Ablauf einiger Wochen oder Monate, oder ein anderer Programmierer etwas daran ändern, kostet die Einarbeitung soviel Zeit, daß sich möglicherweise eher das Neuschreiben lohnt. Deshalb *begleitend zur Programmerstellung* folgende Tätigkeiten durchführen:

- Einen **Ordner** für alle Papierdokumente anlegen, in dem Vorüberlegungen, Entwürfe, Anforderungsdefinition, Angebot, Pflichtenheft, Mails, Protokolle, Ergänzungen, Notizen, eventuell der letzte Programmausdruck (keine Wälder abholzen!), Dokumentation, Sicherungen, ... abgelegt werden.
- Eine **Versionsgeschichte** mitführen, die die wichtigsten Änderungen und Erweiterungen des Programms beschreibt (mit Datum und Autor).
- Eine **Bug-Liste** anlegen, die bereits bekannte aber noch zu behebbende Bugs beschreibt.
- Eine **Todo-Liste** anlegen, die noch notwendige Ergänzungen und Erweiterungen beschreibt.
- Die **Aufrufhierarchie** der Funktionen dokumentieren.
- Den **Zusammenhang** der Datentypen und Klassen dokumentieren.
- **Konstanten, Makros, Datentypen, Variablen und Funktionen** dokumentieren.
- **Programmtechnische Feinheiten** (z.B. „warum so“ bzw. „warum so nicht“ realisiert), Unklarheiten, Workarounds, Heuristiken, ... dokumentieren.
- **Quelltext- und Header-Dateien** dokumentieren (wer includet was).
- **Scripten und Makefiles** dokumentieren (wer macht was).
- Ein **Installations-Skript** erstellen.
- **Test-Skripte** zum automatisierten Testen des Programms erstellen.

- @ Jeder Quelltext beginnt mit einem Dokumentations-Header folgender Form:

```
/*-----
 * Projekt: PROJEKT
```

```

* Kunde:    KUNDE
* Auftrag:  NUMMER
*-----*
* Modul:    NAME
* Version:  NAME
* Autor:    AUTOR
* Datum:    TT.MM.JJ
* Zweck:    TEXT
*-----*
* Aenderungen: (neueste an oberster Stelle)
* TT.MM.JJ VERANTWORTLICHER BESCHREIBUNG
* ...
*-----*/

```

Der Kopf soll kompakt die relevanten Daten des Quelltextes beschreiben, damit beim Ausdruck durch ihn wenig Platz belegt wird.

Wichtig: Im Feld `Datum` bei jeder Änderung *sofort* das Datum aktualisieren!

- @ Wenn mit RCS gearbeitet wird, verwendet man an Stelle dieses Kopfes die Schlüsselwörter `Id` (am Dateianfang) und `Log` (am Dateiende). Eine Beschreibung der Änderungen an einer Quelldatei wird von RCS automatisch beim Einchecken abgefragt und in den Quelltext bei `Log` eingefügt.

```

/*-----*/
/* $Id$ */
/*-----*/
.
.
.
/*-----*/
/* $Log$
/*-----*/

```

Hinweis: Auf `Log` kann aus Platzgründen auch verzichtet werden, die Änderungsliste zu einer eingetragenen Quelldatei `file` erhält man dann über das RCS-Kommando `rlog file`.

- **Kurze und verständliche Kommentare** zur Erläuterung der Aufgabe von Funktionen, der Bedeutung ihrer Argumente und ihres Rückgabewertes im C-Quelltext eintragen. Nicht übertreiben und z.B. offensichtliche Dinge oder Selbstverständlichkeiten nicht beschreiben.
- **Funktionsanfänge** sehen relativ unscheinbar aus. Sie sind daher durch einen Kommentar folgender Form zu kennzeichnen, in dem ihr Zweck sowie die Ein-/Ausgabeparameter und der Rückgabewert beschrieben werden:

```

/*-----*/
/* DESC: Macht ... mit Parametern ... und gibt ... zurueck.
*-----*/
RESULT_TYPE
XyzFunc (ARGUMENTS)
{
    CODE;
}

```

Dank des Schlüsselworts `DESC`: kann das Programm `texdoc` dann bereits einen übersichtlichen Ausdruck des Quellcodes erstellen.

- @ Auch **Quick-and-Dirty**-Programme/Skripten sind zu dokumentieren, nach 3 Monaten weiß man sonst nicht mehr, wozu sie mal gedient haben. Minimum ist eine vollständige **Usage-Meldung**.

2.2 Versionierung

- @ Ein Programm hat eine **eindeutige Versionsnummer W.X.Y.Z**, die bei Usage- oder Fehler-Meldungen ausgegeben wird.
 - Als **Versionsstatus** stehen drei verschiedene Zustände zur Verfügung, nämlich **Exp** (experimental), **Stab** (stable) und **Rel** (released).
 - Diese Versionsnummer und der Versionsstatus sind bei Fehler- oder Usage-Meldungen eines Programms und beim Aufruf mit der Option `-v` bzw. `--version` auszugeben.

```
@(#)PROGRAM V2.5 Exp (03/07/19 08:54) Copyright (C) 2003 T.Birnthaler, OSTC GmbH
```

3 Programmtest

@ Durch eine **Top-Down** Vorgehensweise kann ein Programm bereits während seiner Entstehung und nicht erst nach Abschluß der Programmierarbeiten getestet werden; es ist sozusagen von Beginn der Programmierung an lauffähig.

Dazu sind die noch nicht ausprogrammierten Funktionen einfach mit der Ausgabe „*Ich bin Funktion xyz und tue dies und das*“ zu versehen und mit einem geeigneten Rückgabewert zu beenden.

@ Umfangreiche oder komplizierte Algorithmen und Funktionen zunächst für sich alleine implementieren und testen, bevor sie in ein Programm eingebaut werden. Dies fördert auch die **Modularisierung** des Programms.

@ Bereits vor dem Erstellen eines Programms maßgeschneiderte **Testfälle konstruieren** (Eingabedaten + Ergebnisse) mit denen sich die korrekte Funktion des Programms als Ganzes oder von Teilen davon überprüfen läßt.

Nach größeren Änderungen ist das Programm wieder mit allen diesen Testfällen zu konfrontieren.

@ Die Testfälle müssen auch sämtliche **Grenzsituationen** des Programms überprüfen, um es gegen Programmabsturz bei Fehlbedienung oder fehlerhaften Daten abzusichern. Beispiele sind:

- Keine leere/triviale/falsche Eingabe akzeptieren
- Keine oder falsche Argumente beim Aufruf
- Angabe nicht existierender Dateien
- Verarbeitung sehr großer/sehr vieler Eingabedateien
- Überlange Eingabezeilen
- Datei nicht lesbar/schreibbar, Speicherplatz nicht verfügbar, Datenbank nicht ansprechbar, Rechner nicht erreichbar, ...

@ Am besten erstellt man zum Programmtest ein **Testskript**, das *automatisch* alle Testfälle durchführt und die realen Ergebnisse mit den vorher konstruierten oder mit Ergebnissen vorheriger Testläufe vergleicht. Für Programme mit Oberflächen sind dazu spezielle Tools notwendig.

- Um während des Programmablaufs das Erfülltsein gewisser Bedingungen an bestimmten Programmstellen zu testen, können über das Makro `assert` aus der Header-Datei `assert.h` **Absicherungen** in den Programmtext eingebaut werden. Ist die darin angegebene Bedingung erfüllt, bewirkt die Anweisung nichts. Ist sie nicht erfüllt, wird die Bedingung, der Name des Moduls und die Quelltextzeilennummer ausgegeben und das Programm abgebrochen. Beispiel:

```
for (i = 0; i < MAX && a[i] != NULL; ++i) {
    assert(a[i] < 20000);
    ...
}
```

liefert im Fehlerfalle z.B. folgende Meldung

```
assertion failed: file FILE, line NR.
```

- Ist bei der Übersetzung eines Moduls mit `assert`-Anweisungen die Präprozessorvariable `NDEBUG` (no debug, z.B. durch den Compilerschalter `-DNDEBUG`) definiert, so wird für diese Anweisungen kein Code erzeugt (d.h. es ist als ob diese Anweisungen überhaupt nicht im Quelltext vorhanden wären).
- Für Tests notwendiger Code ist einzurahmen durch:

```
#ifdef TEST
...
#endif /* TEST */
```

Er kann dann durch `#define TEST` oder den Compilerschalter `-DTEST` jederzeit aktiviert werden.

- Tools wie SmartHeap, HeapCheck, BoundsChecker und Insight lokalisieren eine Reihe von Fehlern (insbesondere bei Speicherallokation) automatisch während des Programmlaufs. *Sie sind während der Programmentwicklung periodisch einzusetzen.*
- @ Alle Programme (insbesondere interaktive) werden nicht vom Programmierer selbst, sondern von einer **unabhängigen Person** getestet.

4 Namensvergabe

- @ **Gut gewählte Namen** von Funktionen/Methoden, Datentypen/Klassen, Variablen und Konstanten sind äußerst wichtig für das Verständnis von Programmcode. Dem Compiler sind Namen dagegen egal, für ihn könnten statt Namen auch einfach Nummern verwendet werden. *Der Programmierer muß daher Systematik und Sorgfalt bei der Namensvergabe anwenden.*
- @ Keine Scheu vor **längeren Namen** und **nachträglichem Umbenennen** bei Nichteignung, das macht sowieso der Editor per Suchen + Ersetzen-Kommando (z.B. im `vi` durch `:%s/\<OLD\>/NEW/g`) oder der Stream-Editor `sed` (per `s/\<OLD\>/NEW/g`).
- @ Die Namensvergabe erfolgt nach folgender Systematik:

	Präfix ung. Not.	1. Buchstabe	Folgende Buchstaben	Unter- strich	Beispiel
Konstanten	—	GROSS	GROSS	*	ARRLEN
Makros	—	GROSS	GROSS + klein	*	DEBUG
Datentypen/Klassen	—	GROSS	GROSS + klein	*	File
Variablen	—	klein	klein	*	anzahl
Variablen (ung. Not.)	*	klein	GROSS + klein	—	pString
Funktionen	—	GROSS	GROSS + klein	—	PrintHead

- @ Eine **Mischung von Groß-/Kleinschreibung und Unterstrichen** in einem Namen ist nicht erlaubt. Bei den Namen von Datentypen/Klassen (und Makros) muß man sich für eine von beiden Alternativen entscheiden.
- @ Funktionen **müssen ein Tätigkeits/Eigenschaftswort** enthalten wie z.B.: `Get`, `Init`, `Delete`, ...
- @ Variablen und Konstanten **dürfen kein Tätigkeitswort** enthalten.
 - Makros dürfen Funktionsnamen haben, wenn sie Funktionen vortäuschen.
- @ **Einbuchstabile Variablen** sind nur lokal als Schleifenzähler oder Indices erlaubt.
- @ **Variablen mit angehängter Nummer** als Arrayersatz oder aufgrund von Denkfaulheit sind nicht erlaubt. Statt dessen ein Array einführen oder sinnvolle Namen vergeben.
- @ **Keine kryptischen Abkürzungen** für Namensteile verwenden, nur weil man tippfaul ist (2 statt „to“ und 4 statt „for“ sind erlaubt).
- @ **Gleiche Dinge immer gleich bezeichnen**, Abkürzungen sinnvoll, verständlich und immer gleich durchführen. Negativbeispiel:

```
ges gesch geschft geschaeft      nicht gleichzeitig verwenden!
```

- @ Entweder **nur englische oder nur deutsche Namen** in einem Quelltext verwenden (Empfehlung: englische Namen, da sie kürzer und prägnanter sind).
Dies gilt auch für C-Kommentare und RCS-Kommentare zu Änderungen.

- @ Bei **Änderungen an fremden Quellen** (z.B. allgemeine Bibliotheken) den dort vorgefundenen Stil beibehalten (und ebenfalls die Sprache).
- Bei WINDOWS-Programmen für Variablennamen Präfixe gemäß der **ungarischen Notation** als Typhinweis verwenden (siehe 17 auf Seite 49).

5 Übersicht erhalten

5.1 C-Code

- Keine **goto-Anweisungen** verwenden, sie sind unübersichtlich und gemäß dem Paradigma der *strukturierten Programmierung* auch unnötig.

Ausnahmen: Verlassen verschachtelter Schleifen, Fehlerabbruch und Vermeiden von doppeltem Code in `switch`-Anweisungen.

Achtung: Eine **Sprungmarke (Label)** stellt nur einen Einsprungpunkt dar. Wird sie über die davor stehenden Anweisungen erreicht, so bricht der Programmfluß an dieser Stelle nicht ab, das Label wird einfach ignoriert.

```
...
y = 10;
Labell: /* wird ignoriert */
x = y * y;
...
```

- Den **Komma-Operator** (Sequenz von Anweisungen) nicht verwenden. Er fällt zuwenig auf und wird leicht mit dem Strichpunkt verwechselt.

Ausnahmen: In `for`-Schleifen zur Initialisierung bzw. Weiterschaltung mehrerer Variablen und in Makros, die mehr als einen Befehl enthalten (da sonst bei ihrer Verwendung in `if`-Anweisungen oder Schleifen ohne geschweifte Klammern um den Anweisungsteil nur die erste Anweisung im Körper steht).

```
for (i = 0, k = 0; i < k; ++i, ++k)
    ...

#define SWAP(x, y, tmp) ((tmp) = (x), (x) = (y), (y) = (tmp))
```

- Möglichst wenige **return-Anweisungen** in einer Funktion verwenden. Wenn möglich nur am Anfang einer Funktion (falls sie z.B. falsch aufgerufen wurde) oder am Ende (zum Rücksprung und zur Rückgabe des Funktionswerts), da viele in einer Funktion verstreute `return`-Anweisungen schwer zu überblicken sind bzw. das Fehlen einer `return`-Anweisung leicht übersehen wird.

Zur Vermeidung einer `goto`-Anweisung oder von komplexen Formulierungen darf diese Empfehlung ignoriert werden.

- Bei **Programmfehlern** das Programm (nach einer geeigneten Fehlermeldung auf `stderr`) über `exit(n)` mit einer geeigneten Fehlernummer $n \neq 0$ verlassen. Bei ordnungsgemäßer Ausführung das Programm mit `exit(0)` (oder in `main` durch `return 0`) beenden.

Unter UNIX (und MSDOS/Windows) besteht die Konvention, daß Programme bei erfolgreichem Abschluß den Exitstatus 0 und im Fehlerfall einen Exitstatus $\neq 0$ zurückgeben. In Shell-Skripten oder Batch-Dateien kann darüber eine entsprechende Abfrage und Fehlerbehandlung durchgeführt werden.

- Bei komplexen Schleifen, Bedingungen oder Programmteilen eine Beschreibung der logischen Bedingungen, die *vor und/oder nach dem Durchlaufen* dieser Anweisungen gelten, als Kommentar hinzufügen (**Vorbedingung/Nachbedingung**).

Dies dient der nochmaligen Überprüfung „*Macht der Code das, was er bis zu dieser Programmstelle tun soll*“ oder dem schnelleren Verstehen bei erneuter Beschäftigung mit dem Code. Im hier aufgeführten Beispiel beinhaltet diese Beschreibung natürlich nur die logische Negation der `for`-Schleifenbedingung:

```
for (i = 0; i < MAX && a[i] != NULL; ++i) {
    ...
}
/* hier gilt: i == MAX oder          */
/*          i < MAX und a[i] == NULL */
```

Alternativ kann die Bedingung auch in das Makro `assert()` gesetzt und damit sogar zur Laufzeit überprüft werden:

```
assert(i == MAX || i < MAX && a[i] == NULL);
```

- In komplexen Schleifen die **Schleifeninvariante** suchen und als Kommentar angeben. Dies ist eine logische Bedingung, die während jedem Schleifendurchlauf an dieser Programmstelle erfüllt ist (wohingegen die im vorherigen Punkt angesprochenen Bedingungen vor oder nach dem Verlassen einer Schleife gelten).

```
for (i = 0; i < MAX && a[i] != NULL; ++i) {
    /* Invariante: i < MAX und          */
    /*          fuer alle k aus [0..i]: a[k] != NULL */
    ...
}
```

Die Variable *k* ist dabei keine C-Variable, sondern eine durch den Ausdruck „für alle“ *gebundene* oder *allquantifizierte* Variable, die in der logischen Beschreibung intern verwendet wird.

- `break` und `continue`-Anweisungen in **verschachtelten Schleifen** (`for`-, `do`-, `while`) oder in **verschachtelten switch-Anweisungen** sind besonders sorgfältig zu verwenden und zu dokumentieren, da nur die innerste Schleife abgebrochen und erneut begonnen wird oder nur die innerste `switch`-Anweisung verlassen wird.
- Die **Doppelbedeutung** von `break` zum Abbruch von Schleifen bzw. zum Beenden des `case`-Teils einer `switch`-Anweisung kann bei Verschachtelung von Schleifen und `switch`-Anweisungen zu Verwirrung führen. Entweder sorgfältig dokumentieren, was jeweils gemeint ist oder solche Konstrukte nicht verwenden.
- `case`-Teile von `switch`-Anweisungen, die nicht mit einem `break`- oder `return`-Befehl enden (also auf den nächsten `case`-Teil „**durchfallen**“), sind durch `/*NOBREAK*/` abzuschließen.

- Den `default`-Teil in `switch`-Anweisungen **auf keinen Fall weglassen**, sondern — auch wenn er (vermeintlich) nicht benötigt wird — darin eine Fehlerausgabe oder `assert(0)` eintragen. Wird ein `case`-Fall vergessen, so wird der `default`-Teil erreicht und dann eine Fehlermeldung ausgegeben.
- Nicht mehrere **switch-Anweisungen** verschachteln, da dies sehr unübersichtlich ist.
- Bei **verschachtelten** `else if`-Anweisungen den `else`-Teil nicht weglassen (eventuell `assert(0)` eintragen).
- Der **Rumpf** von `if`-, `else`-, `switch`-, `for`-, `while`- und `do`-Anweisungen soll grundsätzlich in geschweifte Klammern gesetzt werden, auch wenn er nur aus einer einzigen Anweisung besteht. Bei der nächsten Änderung stehen bereits zwei Anweisungen da, oft werden die dann notwendigen geschweiften Klammern vergessen.
- Den **Bedingten Operator** `?:` nur in Makros, bei `printf` und bei `return` verwenden und nicht verschachteln, sondern besser `if-else` verwenden. Die Formulierung über diesen Operator bietet trotz ihrer textuellen Kürze kaum Geschwindigkeitsvorteile (intern wird fast der gleiche Code generiert), sie ist aber meist sehr unübersichtlich.
- Die **Deklaration von Funktionen** aus C-Bibliotheken erfolgt *nicht* manuell im Quellcode sondern durch Includen der jeweiligen System-Include-Datei, in der diese Funktion deklariert ist. Beispiel: `malloc()` wird in der Include-Datei `malloc.h` deklariert, d.h. `#include <malloc.h> statt char* malloc();` angeben.
Achtung: Die Deklarationen von Standardfunktionen können sich bei verschiedenen Compilern unterscheiden (`void* ↔ char*`, `void ↔ int`, `int ↔ long`, `int ↔ t_size`).

5.2 Präprozessor

- In einer `#if ... #endif` Klammer ist nach dem `#endif`-Befehl ein Hinweis auf das korrespondierende `#if` als *Kommentar* anzugeben. Zu jedem `#endif` ist dann sofort das zugehörige `#if` ersichtlich. Die Kommentarklammern sind notwendig, da manche Compiler keinen Text hinter Präprozessorbefehlen akzeptieren. Beispiel:

```
#ifdef MAIN_MODULE
...
#endif /* MAIN_MODULE */
```

- Statt mehrfach verschachtelter `#ifdefs` besser das Kommando `#if` verwenden, hier dürfen beliebig komplexe arithmetische und logische Ausdrücke (ausschließlich auf Basis von Konstanten) angegeben werden:

```
#if defined(TEST) && !defined(__STDC__) || VERSION > 100
...
#endif /* ... */
```

- **Verschachtelte Präprozessorkommandos** sind wie C-Anweisungen (nach dem Zeichen `#`) einzurücken (nur das Zeichen `#` *muß* in der ersten Spalte der Zeile stehen):

```
#ifdef A
#   ifdef B
#       ...
#   endif /* B */
#       ...
#endif /* A */
```

- Wenn bereits während des Präprozessorlaufs eine Fehlermeldung ausgegeben und der Übersetzungsvorgang abgebrochen werden soll, das Kommando `#error` (ANSI-C Standard) mit einem beliebigen Text verwenden:

```
#ifndef MSDOS
#   error Fehler, wird nur von MSDOS unterstuetzt.
#endif
```

- `#includes` **nicht verschachteln**, wenn es sich vermeiden läßt, da dies unübersichtlich ist.

6 Quellcode-Formatierung

6.1 Einrückung und Tabulatoren

@ **Je Verschachtelungsebene um einen Tabulator einrücken**, dies ist schneller und reproduzierbarer als mit Leerzeichen. Außerdem kann bei den meisten Editoren die Tabulatorbreite eingestellt und somit die Bildschirmdarstellung dem persönlichen Geschmack angepaßt werden.

- Der Quelltext ist **nach** einer öffnenden geschweiften Klammer um genau einen Tabulator mehr einzurücken. **Vor** einer schließenden geschweiften Klammer ist er um genau einen Tabulator weniger einzurücken.

```
...
{
    ...
}
...
```

@ Die empfohlene Tabulatorbreite ist **4 Zeichen**, beim Anzeigen im Editor oder Drucken die Tabulatorbreite auf 4 Leerzeichen einstellen.

@ Tabulatoren dürfen **nur am Zeilenanfang** direkt hintereinander stehen und nicht mit Leerzeichen gemischt verwendet werden.

@ Tabulatoren und Leerzeichen dürfen nicht am Zeilenende stehen (überflüssig).

Achtung: In per **Cut-and-Paste** zwischen verschiedenen Fenstern übernommenen Texten sind normalerweise die Tabulatoren durch entsprechend viele Leerzeichen ersetzt. Am Zeilenende können so ebenfalls überflüssige Leerzeichen übertragen werden.

- Mit den obigen Vorgaben haben die C-Kontrollanweisungen folgendes Format:

```
for (...) {           while (BEDINGUNG) {           do {
    ...                ...                ...
}                       } while (BEDINGUNG)

    if (BEDINGUNG) {           switch (AUSDRUCK) {
        ...                ...
    }                       }
```

- Alternativ darf die **öffnende geschweifte Klammer** auch in einer neuen Zeile direkt unter dem ersten Zeichen der Kontrollanweisung stehen. Diese Methode kostet mehr Platz, bietet aber den Vorteil, daß zusammengehörende geschweifte Klammern direkt übereinander stehen. *Nur eine der beiden Methoden darf in einem Quelltext verwendet werden.*

```
for (...)           while (BEDINGUNG)           do
{                   {                             {
    ...                ...                ...
```

```

}           }           } while (BEDINGUNG)

        if (BEDINGUNG)       switch (AUSDRUCK)
        {                     {
            ...                 ...
        }                     }

```

- Geschweifte Klammern und darin eingeschlossenen Code **nicht 2x einrücken**. Negativebeispiele:

```

for (...)           for (...)
{                   {
    ...             ...
}                   }
...                 ...

```

- Bei langen Blöcken kann die schließende Klammer mit einem Kommentar als Hinweis auf den Anweisungsbeginn versehen werden (lange Blöcke sollen allerdings vermieden werden):

```

while (BEDINGUNG) {
    ...
} /* while (BEDINGUNG) */

```

- Bei direkt auf `else` folgendem `if` wird zwischen dem `else` und dem `if` keine geschweifte Klammer eingefügt und analog dem vorhergehenden `if` eingerückt (linkes Beispiel ist okay, rechtes nicht).

```

if (BEDINGUNG) {           if (BEDINGUNG) {
    ...                     ...
}                           }
else if (BEDINGUNG) {     else {
    ...                       if (BEDINGUNG) {
}                             ...
}                             }

```

- Die `cases` und der `default` zu einem `switch` sind einzurücken. Nach einem `case` oder `default` ist ebenfalls einzurücken. Der `default` steht nach allen `cases`.

```

switch (AUSDRUCK)
{
    case x:
        ...
        break;
    ...

    default:
        ...
}

```

6.2 Leerzeichen

- Zwischen `for`, `while`, `if`, `switch` und der nachfolgenden Klammer ist ein Leerzeichen zu setzen, es handelt sich ja nicht um Funktionsaufrufe.

```
FALSCH:      RICHTIG:
for(i = 0; i < argv; ++i) ...   for (i = 0; i < argv; ++i) ...
while(i > 0) ...               while (i > 0) ...
if(i > 1000) ...               if (i > 1000) ...
switch(*cp) ...                switch (*cp) ...
```

- Keine Leerzeichen nach öffnenden und vor schließende runde oder eckige Klammern setzen, sondern stets den Text anschließen lassen.

```
FALSCH:      ( a-b )      if( test_flag ) {      array[ i-1 ]
FALSCH:      (a-b)       if(test_flag){         array[i-1]
RICHTIG:      ( a - b)   if (test_flag) {      array[i - 1]
```

- Kein Leerzeichen vor und ein Leerzeichen nach den Zeichen `,` (Komma) und `;` (Strichpunkt).

```
FALSCH:      Func(a,b,c,d) ;x=1 ;
FALSCH:      Func(a , b , c , d) ; x=1 ;
RICHTIG:      Func(a, b, c, d); x = 1;
```

- Je ein Leerzeichen um die Zuweisungen `=` `+=` `-=` `*=` `/=` `%=` `&=` `|=` `^=` `<<=` `>>=`, die Operatoren `+` `-` `*` `/` `%` `&` `|` `^` `&&` `||` `<<` `>>` und die Vergleiche `<` `<=` `>` `>=` `==` `!=` setzen:

```
FALSCH:      a=b+c      a%=b      i=0      a!=b      a&&b      a>>2
RICHTIG:      a = b + c  a %= b   i = 0    a != b   a && b   a >> 2
```

- Die Operatoren `*` `&` `[]` `.` `->` `()` `!` `~` `++` `--` ohne Leerzeichen dazwischen an die zugehörige Variable/Funktion anschließen.

```
*cp &n arr[n] item.elem ptr->nxt Func() !flag ~bits ++i
```

6.3 Konstanten und Variablen

- Bei mehreren **Konstantendefinitionen** hintereinander die Konstantennamen, Konstantenwerte und Kommentare untereinander ausrichten.

```
#define NUL          '\0'          /* Stringende */
#define INPUT_FILE  "test.dat"    /* Eingabedatei */
#define TEST_SIZE   100           /* Feldgroesse */
...                 ...          ...
```

- Jede **Variable** für sich allein deklarieren (keine Kommas verwenden) und dabei die Datentypen, Variablennamen und Kommentare untereinander ausrichten.

```
int          i;          /* Laufvariable */
char*       cp;         /* Hilfszeiger */
struct MAIN* p_main;    /* Zeiger auf Hauptdatenstruktur */
...         ...         ...
```

6.4 Funktionen

- Bei **Funktionsdeklarationen (Prototypen)** den Rückgabedatentyp, den Funktionsnamen und die Argumentlisten untereinander ausrichten, zu lange Argumentlisten geeignet umbrechen.

```
char*      GetText          (FILE* input);
struct MAIN* FillMainWithDefaults (int argc, char* argv[]);
void      UseTooMuchArguments (int code, char* text, long number,
                               char* desc, struct MAIN* p_main);
...      ...              ...
```

- Bei Funktionsprototypen für jeden Parameter nicht nur den Datentyp, sondern auch einen **aussagekräftigen Parameternamen** angeben.

```
FALSCH: void Func(int, char*);
RICHTIG: void Func(int code, char* message);
```

- Funktionen **ohne Rückgabeparameter** folgendermaßen deklarieren, da sonst als Rückgabebetyp automatisch `int` eingestellt ist.

```
void Function(...);
```

- Funktionen **ohne Argumente** folgendermaßen deklarieren, da sonst als Argumentliste automatisch eine beliebige erlaubt ist (veraltetes K&R-Format).

```
... Function(void);
```

- **Lange Parameter- oder Argumentlisten** oder `if/while`-Bedingungen nach einem Komma umbrechen und bis zur Klammer auf oder um eine Einrückungsstufe einrücken (per Tabulatoren + Leerzeichen!).

```
int
Function(int argument1, int argument2, int argument3, int argument4,
         int argument5, int argument6)
{
    ...
    if (i == 0 && !strcmp(arr[i], "test") ||
        i > 0 && !strcmp(arr[i], "last")) {
        ...
    }
    ...
    fprintf(stderr, "error: Parameter %d und %d verschieden",
            len, cnt);
    ...
}
```

- Hinweis: Durch das Utility *indent* kann ein C-Quelltext auch nachträglich in diese Standardform gebracht werden. Dieses Tool ist sowohl unter MSDOS/Windows als auch unter UNIX verfügbar.

6.5 Kommentare

- Kommentare grundsätzlich **in der gleichen Zeile abschließen**, in der sie beginnen. Man sieht dann sofort, wo ein Kommentar beginnt und endet, das versehentliche Auskommentieren von Quellcode wird erschwert.
- Bei sehr langen Kommentaren können folgende Formen verwendet werden:

```

/*           /*           /*
  Zeile 1      * Zeile 1      ** Zeile 1
  Zeile 1      * Zeile 2      ** Zeile 2
  ...         * ...         ** ...
*/           */           */

```

- C++-Kommentare der Form `// . . .` sind erlaubt.
- **Keine verschachtelten Kommentare** verwenden, sie werden von kaum einem C-Compiler akzeptiert.
- Sollen zu **Debug- oder Testzwecken** Programmteile vorübergehend wirkungslos gemacht werden, diese nicht in Kommentar verwandeln, sondern mit Hilfe des Präprozessors entfernen

```

#if 0
...
#endif /* 0 */

```

oder (wobei natürlich `COMMENTED_OUT` nicht definiert sein darf)

```

#ifdef COMMENTED_OUT
...
#endif /* COMMENTED_OUT */

```

Dies funktioniert auch dann, wenn in den zu entfernenden Zeilen schon Kommentare stehen.

7 Spezielle Programmelemente

7.1 Konstanten und Makros

- **Magische Zahlen, Zeichen und Zeichenketten**, die in einem C-Programm *ein- oder mehrfach* verwendet werden, sind als Konstanten zu definieren. Der Programmtext wird dadurch einfach klarer, und schon bei der ersten Änderung macht sich das bezahlt.

```
#define BLANK      ' '
#define NUL        '\0'          /* NUL != NULL! */
#define INFILE     "input.txt"
#define ARRLEN     256
```

- Konstanten **prophylaktisch in Klammern setzen**, wenn sie durch arithmetische oder logische Operationen entstehen. Beispiele:

```
#define MAXINDEX   (256 - 1)
#define PI_APPROX  (355 / 113)
```

- Konstanten können mit `const` auch als konstante Variable statt als Makro deklariert werden, sie belegen dann genau einmal Platz im Textsegment statt bei jeder Verwendung.

```
const int  MAX_LINE_LEN = 1024 + 1;
const char* INFILE_NAME = "input.txt";
```

- **Keine Scheu vor Makros**, immer wiederkehrende kleine Codestücke können durchaus als Makros formuliert werden. Dabei darauf achten, daß sämtliche Platzhalter im Makrokörper und der ganze Makrokörper **prophylaktisch geklammert** werden.

```
#define ABS(x)     ((x) < 0 ? -(x) : (x))
#define MIN(x,y)  ((x) < (y) ? (x) : (y))
#define MAX(x,y)  ((x) > (y) ? (x) : (y))
```

Durch diese zusätzlichen Klammern werden auch verschachtelte Konstrukte der Form

```
min = MIN(MIN(10 - 2, 5 + 3), MIN(-1, -2))
```

richtig ausgewertet.

- Ein **Nachteil von Konstanten und Makros** ist die Tatsache, daß der *Debugger* sie nicht kennt, dort sieht man immer nur die echten Werte. Verwendet man für **Aufzählungstypen** statt Konstanten den Datentyp `enum` und statt Makros `inline`-Funktionen, so tritt dieses Problem bei sonst gleicher Übersichtlichkeit und Performance nicht auf.
- Im Unterschied zu Funktionsargumenten können Makroargumente **mehrfach ausgewertet** werden und dürfen daher **keine Seiteneffekte** bewirken. Zum Beispiel ist nach der Ausführung von

```
min = MIN(++i, max)
```

i nicht unbedingt nur um 1, sondern eventuell um 2 erhöht worden, da in dem von `MIN` generierten Code der Ausdruck `++i` möglicherweise zweimal ausgewertet wird. Ebenso wird bei der Ausführung von

```
min = MIN(g(), f())
```

eine der beiden Funktionen *g()* oder *h()* zweimal aufgerufen.

- Makros dürfen nicht zum Einführen einer **zweiten Schnittstelle** für Funktionen (für Tippfaule) mißbraucht werden.
- Ein **mehrzeiliges Makro** ist besser in Form einer Funktion zu realisieren (eventuell als `inline`-Funktion).
- Makros sollen *keine casts* enthalten, da der Compiler dadurch eventuell falsche Parameter nicht erkennen kann.
- Die **Länge** eines statischen Feldes oder einer Zeichenkettenvariablen ist eine Konstante und kann durch folgende Makros (zur Compile-Zeit) erhalten werden:

```
#define ARR_SIZE (sizeof(arr) / sizeof(arr[0])) /* ODER */  
#define ARR_SIZE (sizeof(arr) / sizeof(*arr))  
#define STR_SIZE sizeof(str)
```

7.2 Include-Dateien

- Include-Dateien müssen **selbst-genügend** und **idempotent** sein. Das bedeutet:
 - Zu ihrer Übersetzung müssen keine weitere Headerdateien includet werden.
 - Sie können in einer Quellcode-Datei beliebig oft eingelesen werden, ohne Übersetzungsfehler zu verursachen.

Dies wird einfach dadurch erreicht, daß jede Include-Datei alle für sie notwendigen Include-Dateien selbst includet und sich gegen **mehrfaches Einfügen** schützt.

Dies geschieht in jeder Include-Datei durch Abfrage und Definition einer Präprozessorvariablen, die den Namen der Include-Datei trägt (groß geschrieben und ‘_’ statt ‘.’), also z.B. für die Datei `structs.h`

```
#ifndef STRUCTS_H
#   define STRUCTS_H
#   include <...>      /* notwendige weitere Include-Dateien */
#   include "... "    /* notwendige weitere Include-Dateien */
...                  /* Includetext */
#endif /* STRUCTS_H */
```

Wird eine nicht so geschützte Include-Datei mehr als einmal eingelesen, kann es durch doppelte `typedefs` oder `#defines` zu Fehlern kommen.

- Steht eine Include-Datei zwischen doppelten Hochkommas "...", so wird zuerst das **aktuelle Verzeichnis** durchsucht und danach die Verzeichnisse im Include-Pfad (Umgebungsvariable `INCLUDE`) in der darin angegebenen Reihenfolge.
- Ist eine Include-Datei durch Winkelklammern `<...>` eingeschlossen, wird das aktuelle Verzeichnis *nicht* beachtet, sondern sofort die Verzeichnisse im Include-Pfad (Umgebungsvariable `INCLUDE`) in der darin angegebenen Reihenfolge durchsucht.
- Sofern **Standard-Include-Dateien** (in `/usr/include` oder in `/usr/include/sys`) nicht gegen doppeltes Includen geschützt sind, ist dies entweder vom Superuser nachzutragen (schlecht portabel), oder es wird besser folgendermaßen vorgegangen (Beispiel `malloc.h`):
 - In jeder Quelldatei die Anweisung `#include <malloc.h>` umwandeln in die Anweisung `#include "malloc.h"`.
 - Eine Datei `malloc.h` im eigenen Verzeichnis anlegen, die folgende Anweisungen enthält:

```
#ifndef MALLOC_H
#   define MALLOC_H
#   include <malloc.h>
#endif /* MALLOC_H */
```

Beim Übersetzen wird jetzt zuerst die eigene Include-Datei `malloc.h` includet, diese includet nur ein einziges Mal die Standard-Include-Datei `malloc.h`.

- **Nur die unbedingt notwendigen (System-)Header-Dateien** includen, keinen Rundumschlag machen (kostet Übersetzungszeit).
- **Keine absoluten Pfadangaben** bei Include-Dateien verwenden (nicht portabel), sondern die Suchpfade im Makefile oder beim Compiler mit angeben. Relative Pfade wie `sys/time.h` oder `../main.h` sind natürlich erlaubt.
- Unter UNIX sollte der Pfad `/usr/local/include` zum Suchpfad für Includes gehören.
- Include-Dateien dürfen nur Definitionen und Deklarationen, aber **keinen Programmcode** enthalten. Dies ist eine äußerst bedenkliche Art der Wiederverwendung von Programmcode. Derartigen Code stattdessen in eine getrennte Objektdatei oder ein Bibliotheksmodul auslagern und dazulinken.

7.3 Datentypen und Casts

- Bei der Definition von `struct`- oder `union`-Datentypen kann entweder der Typname oder der Name nach dem Schlüsselwort `struct/union` weggelassen werden. Werden beide angegeben, sollen sie verschieden lauten (müssen aber nicht).

```
typedef struct NODE_tag {
    int    data;
    char*  text;
} NODE;
```

Es hat sich eingebürgert, den Namen nach `struct` auf `_tag` enden oder mit `tag` beginnen zu lassen. Zur Variablendeklaration können beide Typnamen `struct XXX_tag` und `XXX` gleichermaßen verwendet werden, sie werden vom Compiler nicht unterschieden.

- Für den **Byte-Sex**, d.h. die Abbildung der Bytes eines elementaren Datentyps auf aufeinanderfolgende Speicheradressen gibt es prinzipiell zwei Möglichkeiten:
 - **little-endian** = niedrige Speicheradressen werden zuerst belegt.
 - **big-endian** = hohe Speicheradressen werden zuerst belegt.

Beim Transport von *binären* Daten zwischen INTEL- (little-endian) und RISC-Maschinen (big-endian) muß daher eine Umsetzung der Bytes erfolgen. Beispiel für eine `long`-Zahl:

	i	i+1	i+2	i+3	Speicheradresse
	+-----+-----+-----+-----+				
0x44332211L =>	0x11	0x22	0x33	0x44	Intel-Prozessor
	+-----+-----+-----+-----+				
0x44332211L =>	0x44	0x33	0x22	0x11	RISC/68000-Prozessor
	+-----+-----+-----+-----+				

- Da vom Datentyp abhängig ist, wieviele Bytes wie vertauscht werden müssen, ist der reine Byte-Datenstrom für sich alleine nicht interpretierbar. Am besten legt man das externe Speicherformat fest, indem z.B. XDR (**external data representation**) verwendet oder die Daten im **ASCII** bzw. **XML-Format** gespeichert werden.
- Am Anfang einer binären Datei soll durch einen Header erkennbar sein, auf welche der beiden Arten sie abgespeichert wurde. Dies kann z.B. durch Ablegen der Zeichenfolge ABCD über eine `long`-Variable (`long x = 0x41424344L = "ABCD" oder "DBCA"`) erfolgen.
- Für viele Prozessoren ist es notwendig, daß elementare Datentypen, die länger als ein Byte sind, an einer durch die Datentyplänge teilbaren Speicheradresse beginnen (**Alignment**). Einige andere Prozessoren (z.B. INTEL) erlauben zwar eine beliebige Ausrichtung der Daten im Speicher, sind aber im obigen Fall schneller. Daher werden von der meisten C-Compilern zwischen den Elementen von Datenstrukturen oder Feldern **Lücken** eingefügt, der `sizeof`-Operator zählt diese Lücken aber immer mit. Beispiel für eine Maschine mit 4-Byte Alignment.

```

typedef struct {
    char  c1; +-----+-----+-----+-----+-----+-----+-----+-----+
    char  c2; | c1 | c2 | c3 | -- |   s1   | -- | -- |
    char  c3; +-----+-----+-----+-----+-----+-----+-----+-----+
    short s1; |           11           | -- | -- | -- | -- |
    long  l1; +-----+-----+-----+-----+-----+-----+-----+-----+
    double d1; |                               d1                               |
} TEST;      +-----+-----+-----+-----+-----+-----+-----+-----+

```

`sizeof(TEST)` liefert 24, obwohl nur $3 \times 1 + 2 + 4 + 8 = 17$ Bytes belegt sind.

- Durch **Umsortieren** der Elemente einer Datenstruktur können solche Löcher häufig eliminiert werden, falls Speicherplatz wirklich ein Problem sein sollte.
- „Wildes“ herum-casten zeugt von schlechtem Programmierstil und kann zu Fehlern führen. Die meisten Umwandlungen führt der Compiler automatisch von selbst durch.
- Im Zeitalter der Funktionsprototypen ist ein **Cast vor der Konstanten** `NULL` nicht mehr nötig *außer* bei Funktionen mit einer variablen Anzahl von Argumenten.
- Unter WINDOWS immer das **large-Modell** (lange Funktionsadressen und lange Zeiger auf Daten) verwenden und die Datenstrukturen maximal packen (**Zp1**).

7.4 Dateizugriff

- C kennt zwei Dateischnittstellen, die erste stammt aus dem UNIX-System, die zweite ist im ANSI-Standard vorgeschrieben und soll vorzugsweise verwendet werden:
 1. **Ganzzahlige Filedeskriptoren**, die mit den Funktionen `creat`, `open`, `lseek`, `read`, `write`, `close`, ... verwaltet werden und eine *ungepufferte* Ein-/Ausgabe realisieren (UNIX).
 2. **Filepointer-Datenstrukturen**, die mit den Funktionen `fopen`, `fseek`, `fread`, `fwrite`, `fclose`, ... verwaltet werden und eine *gepufferte* Ein-/Ausgabe realisieren (ANSI-C).
- Aufgrund der vom Betriebssystem abhängigen **Form von Dateinamen** (Länge, erlaubte Zeichen, Verzeichnis-Separator, Groß-/Kleinschreibung relevant) sollen alle in einem Programm verwendeten festen Dateinamen in einer eigenen Headerdatei stehen.
- Unter MSDOS dürfen **Dateinamen maximal 11 Zeichen lang** sein, max. 8 Zeichen echter Name und max. 3 Zeichen Extension, getrennt durch einen Punkt. Die Extension legt in der Regel den Typ der Datei fest.
- Als kleinster gemeinsamer Nenner der verschiedenen UNIX-Versionen dürfen **Dateinamen maximal 14 Zeichen lang** sein. Im Unterschied zu MSDOS ist mehr als ein Punkt im Dateinamen erlaubt, er zählt als ein Zeichen und darf an beliebiger Stelle stehen.
- Der **Separator zwischen Verzeichnissen** bzw. zwischen Verzeichnis- und Dateiname ist unter MSDOS `\` und unter UNIX `/`. Dies gilt allerdings nur für Kommandozeilenaufrufe, in Funktionsaufrufen kann einheitlich der Schrägstrich (`/`) verwendet werden.
- Das **Optionszeichen** ist unter MSDOS `'/'` und unter UNIX `'-'`.
- Der **Pfadtrenner in Suchpfaden** ist unter MSDOS `';'` und unter UNIX `':'` .
- Wer ganz sauber arbeiten will kann folgende Makrodefinitionen verwenden:
 - `FILEPATHSEP` liefert den Dateipfad-Trenner.
 - `OPTSTART` liefert das für Kommandozeilen-Optionen benutzte Zeichen.
 - `SEARCHPATHSEP` liefert das in Suchpfaden benutzte Trennzeichen.
- **Laufwerke** (`A:`, `B:`, ...) gibt es nur unter MSDOS. Entsprechende Quellcodeteile für alle anderen Rechner auskommentieren bzw. ganz auf die Behandlung von Laufwerken verzichten.
- **Dateinamen klein schreiben**, da sie unter MSDOS sowieso in Großschreibung umgesetzt werden, und unter UNIX Dateinamen standardmäßig klein geschrieben werden. Allerdings werden von den MSDOS-Funktionen `_dos_findfirst`, `_dos_findnext`, ... Dateinamen normalerweise groß geschrieben zurückgeliefert.

- **Zeilen in Textdateien** werden unter UNIX durch das Steuerzeichen `<nl>` (newline = ASCII-Code 10), unter MSDOS durch die zwei Steuerzeichen `<cr>` `<lf>` (carriage return + linefeed = ASCII-Codes 13 + 10) getrennt. Unter MSDOS wird eine Textdatei eventuell auch noch durch ein `Ctrl-Z` (ASCII-Code 26) terminiert.

Um unter MSDOS im üblichen Format abgespeicherte Textdateien mit C-Programmen wie unter UNIX verarbeiten zu können, erfolgt beim Lesen und Schreiben von Textdateien *automatisch* eine Umsetzung. Beim Lesen mit `fgets`, `fscanf`, `fread`, ... wird `<cr>` `<lf>` in `<nl>` umgewandelt und `Ctrl-Z` entfernt, beim Schreiben mit `fputs`, `fprintf`, `fwrite`, ... umgekehrt. Soll diese Umwandlung unter MSDOS unterbleiben, sind die Dateien im **binären Modus** zu öffnen (Default unter UNIX):

```
in_fp = fopen("infile.dat", "rb");
out_fp = fopen("outfile.dat", "wb");
```

- Dieser Unterschied in der internen Speicherung von Texten führt auch bei der **Übertragung von Quelltexten** von MSDOS- auf UNIX-Rechnern zu Schwierigkeiten. Das Zeichen `<cr>` am Zeilenende wird z.B. von `make` oder `awk` nicht akzeptiert, das Zeichen `Ctrl-Z` am Dateiende mag der C-Compiler nicht.

Daher bei der Übernahme von Quelltexten nicht die Umwandlung mit `dos2unix` bzw. `unix2dos` vergessen. Achtung: Umlaute werden von diesen beiden Programmen nicht umgesetzt.

- Die Funktion `fseek` führt auf Textdateien nur dann zu sinnvollen Ergebnissen, wenn die Stelle, auf die positioniert werden soll, vorher über `ftell` oder ähnlich ermittelt wurde (nur die Positionierung auf den Anfang einer Zeile ist sinnvoll). Da Textzeilen in der Regel verschieden lang sind, kann ihr Anfang nicht berechnet werden.
- Zwischen einem Lese- und einem Schreibzugriff (oder umgekehrt) auf einer zum Lesen + Schreiben geöffneten Datei *muß* der Dateizeiger mit `fseek` positioniert werden.
- In einem System mit **gepufferter Ausgabe** kann es sein, daß erst nach dem Aufruf von `fflush(file)` oder `fclose(file)` alle vorherigen Ausgaben auf die Datei `file` auch wirklich in ihr stehen.

7.5 Fließkommazahlen

- Fließkommakonstanten immer mit **Dezimalpunkt oder Exponent** angeben. Dies dient der Klarheit und der Compiler erzeugt bereits zur Übersetzungszeit die entsprechende Fließkommazahl, statt sie jedesmal zur Laufzeit aus einer Ganzzahl zu konvertieren.
- Der **Vergleich von Fließkommazahlen** mit dem Wert 0.0 oder anderen festen Werten, die keine ganze Zahl oder eine Summe von Zweierpotenzen (0.5, 0.25, 0.125, 0.625, ...) darstellen, ist gefährlich, da bei Fließkommaoperationen **Rundungsfehler** auftreten können. Am besten auf einen kleinen Bereich um den abzufragenden Wert testen:

```
#define EPSILON 0.00001  
  
if (fabs(val - 10.5) < EPSILON) ...
```

- **Fließkommaberechnungen** immer mit `double`-Variablen durchführen, `float`-Variablen nur verwenden, falls sehr sehr viele Variablen (Array) benötigt werden, um damit Speicherplatz zu sparen. Die Genauigkeit von `float`-Zahlen beträgt nur 7–8 Stellen nach dem Komma, d.h. eine Zahl > 10 Millionen kann *nicht mehr exakt* dargestellt werden. Außerdem wandelt der Computer in Berechnungen `float`-Zahlen grundsätzlich in das `double`-Format um und bildet erst das Ergebnis wieder auf `float` ab. Diese Datentyp-Umwandlung kostet Rechenzeit.
- Der Computer rechnet nicht — wie die Mathematik — mit **reellen Zahlen**, sondern kennt nur eine Teilmenge der reellen Zahlen. Jede Zahl, die nicht in diese Teilmenge fällt, wird auf eine der darstellbaren Zahlen abgebildet. Dies kann zur Folge haben, daß komplizierte Berechnungsverfahren (z.B. Matrizeninvertierung) durch Rundungsfehler so sehr aufgeschaukelt werden, daß das Endergebnis völlig unbrauchbar ist.
- In der numerischen Mathematik gibt es dennoch zahlreiche Verfahren, die auch auf realen Rechnern mit ihrem beschränkten Zahlenbereich stabil sind und sinnvolle Ergebnisse liefern. Bei Bedarf die jeweiligen Anwender, einen Mathematiker oder Mathematikbücher konsultieren.

7.6 Dynamischer Speicher

- Bei Anforderung von Speicherplatz über `malloc` oder `calloc` immer auf die **Rückgabe von NULL prüfen**, und das Programm mit einer geeigneten Fehlermeldung abbrechen oder eine geeignete Fehlerbehandlung einleiten.

```
if ((cp = malloc(100)) == NULL) {
    printf("error: can't alloc ...\n");
    exit(3);
}
```

- Soll Speicherplatz in anderen Einheiten als `char` angefordert werden, keine absoluten Zahlen angeben, sondern den **sizeof-Operator** (evtl. multipliziert mit der gewünschten Anzahl an Einheiten) verwenden. `sizeof` ist keine Funktion, sondern ein Operator, und wird daher *bereits zur Übersetzungszeit ausgewertet*.

```
if ((ip = malloc(10 * sizeof(int))) == NULL) {
    printf("error: can't alloc ...\n");
    exit(3);
}
```

- Im Gegensatz zu `calloc` wird der angeforderte Speicherplatz bei `malloc` (und `realloc`) **nicht mit 0-Bytes initialisiert**, d.h. er kann beliebige Bitmuster enthalten.
- Durch `realloc` wird eventuell der **Speicherbereich verschoben**, dessen Größe verändert werden soll. D.h. sämtliche Zeiger, die in diesen Bereich zeigen, sind anschließend ungültig.
- `free(p)` gibt den für `p` reservierten Speicherplatz frei, belegt aber `p` *nicht* mit dem `NULL`-Zeiger. Weitere Schreiboperationen auf `p` zerstören dann die Datenstrukturen der Speicherverwaltung. Deshalb beim Freigeben von Speicherplatz den zugehörigen Zeiger sofort anschließend mit `NULL` belegen oder folgendes Makro zum Freigeben verwenden:

```
#define FREE(ptr) ((ptr) && free(ptr), (ptr) = NULL)
```

- Die Suche nach fälschlicherweise freigegebenem oder versehentlich nicht freigegebenem dynamischen Speicher erleichtern Tools wie SmartHeap, HeapCheck, BoundsChecker oder Insight.
- Unter MSDOS kann der allokierte Speicherbereich **maximal 65530 Bytes** groß sein, unter UNIX **maximal 4 Mrd Bytes**
- In grafischen Oberflächen kann auch sinnvoll sein, `malloc` auf eine Funktion abzubilden, die nicht reservierbarem Speicher dies in einer Messagebox meldet oder eine allgemeine Fehlerbehandlungsfunktion `SetNoMemHandler()` zu definieren.

8 Häufige Fehler

- Der zu einer Syntax-Fehlermeldung gehörende Fehler kann **sehr weit vor** dem eigentlichen Fehler aufgetreten sein (*aber nie danach*). Dies tritt z.B. bei vergessenen Strichpunkten, geschweiften, eckigen oder runden Klammern auf/zu oder Anführungszeichen häufig auf.

Achtung: Ein Fehler in einer Header-Datei kann sich erst in der nächsten Header-Datei oder der sie inkludierenden Quellcode-Datei auswirken.

- Der **Lieblingsfehler** aller Anfänger und Fortgeschrittenen (= statt == beim Vergleich).

```
if (x = 10)
    ...
```

Viele Compiler geben hier eine Warnung aus, wenn der Warning-Level hoch genug gesetzt ist. Man kann sich auch angewöhnen, beim Vergleich mit Konstanten immer die Konstante zuerst zu schreiben, auch hier meckert der Compiler sofort.

```
if (10 = x)
    ...
```

- Der zweitliebste Fehler aller Fortgeschrittenen: Fehlende Klammer um Zuweisung, d.h. *fp* bekommt den Wert 1 oder 0 als Pointer, da der Vorrang von == höher ist als der von =.

```
if (fp = fopen("FILE", "r") == NULL)
    ...
```

Auch in dieser Variation bekannt.

```
if (cp = malloc(100) == NULL)
    ...
```

- Der dritte Fehler auf der ewigen Bestenliste: Der Index schießt übers Ziel hinaus, d.h. es wird ein Arrayelement `a[MAXLEN]` angesprochen, obwohl nur die Elemente `a[0]` bis `a[MAXLEN - 1]` existieren.

```
int a[MAXLEN];

for (i = 0; i <= MAXLEN; ++i)
    a[i] = ...
```

- Obigen Fehler nennt man auch **Zaunpfosten-** (fencepost) oder **um-eins-daneben-** (off-by-one) Fehler. Er ist zu vermeiden, wenn man den zu durchlaufenden Bereich einer Schleife durch **asymmetrische Grenzen** beschreibt. Die Anzahl der Schleifendurchläufe ergibt sich dann aus der Differenz der beiden Grenzen.

```
for (i = 123; i < 456; ++i) /* 456 - 123 = 333 Durchläufe */
for (i = 234; i > 19; --i) /* 234 - 19 = 215 Durchläufe */
for (i = min; i < max; ++i) /* max - min          Durchläufe */
```

Im letzten Fall wird der Schleifenrumpf gar nicht durchlaufen, wenn $max == min$ ist!

- Bei jeder Schleife sind die beiden **Grenzfälle** Beginn und Ende besonders sorgfältig auf Korrektheit zu überprüfen, da hier am häufigsten Fehler gemacht werden.
- Die Angabe von **zuwenig Argumenten** bei `printf/scanf` führt zu seltsamen Ausgaben bzw. Fehlern (zuviele Argumente sind hingegen normalerweise unschädlich).

```
printf("%d %s", i);
scanf("%d %s", &i);
```

- Die Angabe von **Variablen statt ihrer Adressen** bei `scanf` führt ebenfalls zu seltsamen Fehlern.

```
scanf("%d", i); /* statt &i */
```

- Ebenfalls sehr beliebte Fehler sind **vergessene breaks** im `case`-Teil von `switch`-Anweisungen. Das Programm läuft dann einfach in den nächsten `case`-Teil hinein, dieses Standardverhalten ist jedoch nur sehr selten erwünscht.
- Ein weiterer häufiger Fehler ist ein **zusätzlicher ; (Strichpunkt)** nach `if`-, `for`- und `while`-Anweisungen und in **Makro-Definitionen**. Im ersten Fall ist der eigentliche Schleifenrumpf leer und der vorgesehene Schleifenrumpf wird immer genau einmal ausgeführt.

```
for (i = 0; i < MAX; ++i);
    ...
```

Im zweiten Fall entstehen bei der Verwendung des Makros „*unsichtbare*“ *Strichpunkte* im Quelltext, die zu äußerst seltsamen Fehlermeldungen des Compilers führen können.

```
#define ANZ 100;
```

- Soll der **Rumpf einer Schleife leer** bleiben, weil die ganze Arbeit schon in der Schleifenbedingung und in der Schleifenweitschaltung geleistet wird, so ist dies mit einem auf eine eigene Zeile abgesetzten und eingerückten `continue` oder dem Kommentar `/*EMPTY*/` deutlich zu machen.

```
for (i = 0; i < MAX && a[i] != NULL; ++i)
    continue;
```

- Beim Zugriff auf ein Element eines **mehrdimensionalen Feldes** wird in C eine gänzlich andere Syntax verwendet als in PASCAL. Trotzdem wird die PASCAL-Schreibweise von C-Compilern in der Regel klaglos akzeptiert, da der Komma-Operator und der Zugriff auf Teile eines Feldes in C erlaubt sind.

```
int a[5][3]; /* 2-dimensionales Array */
int x;

x = a[2,1]; /* FALSCH: PASCAL-Syntax, entspricht x = a[1] */
x = a[2][1]; /* RICHTIG: C-Syntax */
```

- Vorsicht bei Funktionen der C-Bibliothek, die **Zeichen zurückliefern** (`getchar`, `fgetc`, ...), diese haben als Ergebnistyp `int` und nicht `char`. Die Zuweisung ihres Ergebnisses an eine `char`-Variable führt dazu, daß als negative Zahlen codierte Fehlerwerte wie `EOF`, `ERR`, ... in einen anderen Wert konvertiert werden, und daher ein Vergleich mit diesen Werten nie wahr ergeben kann.
- Vorsicht bei **unsigned Variablen**; folgendes Programmstück ergibt eine Endlosschleife, da `unsigned` Variablen keinen negativen Wert annehmen können, sondern 0 – 1 den größtmöglichen positiven Wert ergibt.

```
unsigned int i;
{
    for (i = 100; i >= 0; --i)
        ...
}
```

- **Reine unsigned-Ausdrücke** ergeben keinen negativen Wert. Z.B. ergibt

```
sizeof(char) - sizeof(int)
```

eine positiven Wert, obwohl `char` kleiner als `int` ist, da `sizeof` einen `unsigned`-Wert liefert, und Operationen ausschließlich zwischen `unsigned`-Werten wieder einen `unsigned`-Wert ergeben.

- **Nicht initialisierte Zeiger**, bereits freigegebene Zeiger, Nullzeiger oder Zeiger, die zufällig auf irgendeinen Teil des Speichers zeigen, führen zu äußerst seltsamen und oft nicht reproduzierbaren Fehlern, die rechner- und compilerabhängig sind. Häufig werden dabei die Verwaltungsdaten der dynamische Speicherverwaltung zerstört.

UNIX-Rechner erkennen lesende und schreibende Zugriffe auf derartige Speicherbereiche außerhalb des eigenen Programms und brechen sie mit **Memory Fault** ab. MSDOS/Windows-Rechner dagegen akzeptieren sie meist klaglos und stürzen erst etwas später oder gar nicht ab.

- **Zeiger und Arrays** sind sehr ähnlich, denn Arraynamen werden intern (z.B. bei der Übergabe als Argument an eine Funktion) in einen Zeiger auf das erste Element des Arrays umgewandelt. Deshalb erfolgt die Parameterübergabe **by reference**, und nicht wie bei den skalaren Objekten **by value**.

Dies macht Zeiger und Array allerdings nicht beliebig austauschbar. Gibt es z.B. im Quellcode eines Modules die Definition `char a[6]`, darf in einem anderen Modul nicht `extern char *a` an Stelle von `extern char[] a` verwendet werden.

- **Nichtstatische lokale Variablen** liegen auf dem **Stack**, der beim Verlassen der Funktion wieder freigegeben bzw. beim nächsten Aufruf einer Funktion wieder für andere Variablen verwendet wird. Deshalb beachten:

- Lokale Variablen müssen **explizit initialisiert** werden, weil ihr Anfangswert im Gegensatz zu globalen Variablen undefiniert ist.
- Lokale Variablen einer Funktion, deren Adresse als Ergebnis zurückgegeben wird, sind als `static` zu deklarieren, da sonst der Zeiger nach dem Verlassen der Funktion ins Nirwana zeigt. *Derartige Fehler sind sehr schwer zu finden.*
- Das Überschreiben einer lokalen Variablen über ihre physikalische Größe hinaus zerstört den Stack. *Derartige Fehler sind sehr schwer zu finden.*
- Der Wert einer lokalen Variablen geht mit dem Verlassen der Funktion verloren und steht deshalb nicht mehr für die Weiterbearbeitung beim nächsten Funktionsaufruf zur Verfügung.
- Der Speicherbereich von lokalen statischen Variablen kann nicht freigegeben werden.
- Große lokale Arrays können unter MS-DOS leicht den Stack zum Überlauf bringen. Um dies zu vermeiden, lokale Arrays immer dynamisch allokalieren.

```
if (cp == NULL)
    cp = malloc(100);
```

- Programmtexte, die durch

```
#ifdef COMMENTED_OUT
...
#endif /* COMMENTED_OUT */
```

vorübergehend wirkungslos gemacht werden sollen, müssen im Gegensatz zu den herkömmlichen C-Kommentaren (`/* */`) gültigen C-Code enthalten.

- Manche Rechner, insbesondere RISC-Maschinen, lassen die Speicherung von elementaren Datentypen wie `short`, `int`, `long`, `float`, ... nur auf Speicherpositionen zu, die ein ganzzahliges Vielfaches dieser Datengrößen als Startadresse haben. Wird dies (wie im folgenden Programmstück) nicht beachtet, tritt ein **Bus** oder **Alignment Error** auf.

```
typedef struct {
    int x;
} DATA;

DATA p[10];
DATA *q;

p->x = 100;
printf("p->x = %d\n", p->x);

q = (DATA*) ((char*)p + 1); /* Fehler! */

q->x = 100; /* BUS ERROR */
printf("q->x = %d\n", q->x);
```

- **Stringfunktionen** auf RISC-Rechnern akzeptieren keine `NULL`-Zeiger als Argumente, auch wenn nur lesend darauf zugegriffen wird. Zum Beispiel führt folgender Aufruf zum Rechnerabsturz.

```
p = strcat("abc", NULL); /* ueberhaupt kein String! */
```

erlaubt ist dagegen

```
p = strcat("abc", ""); /* String der Laenge 0 */
```

- Nur Dateien schließen, die auch geöffnet wurden. Nur Speicherplatz freigeben, der auch allokiert wurde. Dabei wird jeweils Verwaltungsinformation in den Speicher geschrieben, bei ungültigen Zeigern also der Inhalt irgendwelcher Speicherzellen zerstört. Absichern durch

```
if (fp) fclose(fp);
if (mem) free(mem);
```

- **Puffer** werden häufig zu klein dimensioniert bzw. nicht auf Überlauf abgeprüft. Der *Internet-Worm* ist genau durch einen solchen Fehler in der C-Bibliotheksfunktion `gets()` möglich gewesen.
- Das Abprüfen von zwei Bedingungen in der Form

```
if (min < x < max) ...
```

ist logisch falsch, syntaktisch aber zulässig, da in C ein logischer Ausdruck die Zahl 0 oder 1 liefert. Richtig ist in diesem Fall natürlich die Abfrage

```
if (min < x && x < max) ...
```

- **Seiteneffekte** eines Ausdrucks sind alle Veränderungen im Speicher, die während der Auswertung eines Ausdrucks stattfinden. Beispielsweise sind die Seiteneffekte des Ausdrucks

```
j = *x++ + g() + h();
```

die Zuweisung an j , an x sowie alle Seiteneffekte, die bei der Auswertung der Funktionen $g()$ und $h()$ auftreten. Die **Reihenfolge von Seiteneffekten ist nicht festgelegt**, daher Vorsicht vor Ausdrücken, in denen die Auswertungsreihenfolge nicht festliegt (compilerabhängig) und gleichzeitig Seiteneffekte stattfinden. In

```
x = 2;
j = ++x * --x;
```

erhält j je nach Compiler den Wert $3 * 1$, $3 * 2$ oder $2 * 1$. In

```
int g() { return ++x; }
int h() { return --x; }

x = 2;
j = g() * h();
```

erhält j je nach Compiler den Wert $3 * 2$ (erst wird $g()$ und dann $h()$ aufgerufen) oder $2 * 1$ (umgekehrt). Auch die harmlos und effizient erscheinenden Zuweisungen

```
arr[i++] = i;
arr[i] = ++i;
```

arbeitet nicht korrekt, da die Auswertung des rechten Ausdrucks vor oder nach der Auswertung des Indexausdrucks stattfinden kann.

- Die Anweisung

```
printf(str, "%s %d", str, i)
```

zum Anhängen von Text an einen String funktioniert bei vielen C-Compilern, aber nicht bei allen. Die Anweisung

```
printf(str, "%s %s", "TEST: ", str)
```

funktioniert bei kaum einem C-Compiler, da der bisherige Inhalt von *str* möglicherweise schon überschrieben ist, wenn er benötigt wird.

- `break` verläßt nur die **direkt umgebende Schleife**, mehrfach verschachtelte `for`-, `do`- und `while`-Schleifen müssen über eines der folgenden Konstrukte verlassen werden.

```
/* 1. Konstrukt: Wirkt geheimnisvoll */
for (i = 0; i < MAX1; ++i) {
    for (j = 0; j < MAX2; ++j) {
        ...
        if (beenden == TRUE) { /* Abbruch? */
            i = MAX1;
            j = MAX2;
        }
    }
}
/* Hier soll es nach dem Abbruch weitergehen */
```

```
/* 2. Konstrukt: Erlaubtes goto */
for (i = 0; i < MAX1; ++i) {
    for (j = 0; j < MAX2; ++j) {
        ...
        if (beenden == TRUE) /* Abbruch? */
            goto Weiter;
    }
}
/* Hier soll es nach dem Abbruch weitergehen */
Weiter:
```

```
/* 3. Konstrukt: Kostet etwas mehr Zeit */
ende = FALSE;
for (i = 0; !ende && i < MAX1; ++i) {
    for (j = 0; !ende && j < MAX2; ++j) {
        ...
    }
}
```

```
        if (beenden == TRUE) /* Abbruch? */
            ende = TRUE;
    }
}
/* Hier soll es nach dem Abbruch weitergehen */
```

Welches der drei Konstrukte verwendet wird, hängt von der Problemstellung ab, jeweils das natürlichste und übersichtlichste ist zu verwenden.

9 Optimierung

- **Optimierung im Kleinen** ist nicht sinnvoll, der Compiler ist darin besser. Zum Beispiel kann auf **Programmiertricks** wie Shiften statt mit einem Vielfachen von 2 zu multiplizieren oder ähnliches verzichtet werden, die meisten Compiler optimieren die Auswertung von Ausdrücken bereits von sich aus ganz gut. Z.B. wird schon sehr guter Code für folgende C-Standard-Operationen erzeugt:

```
*cp++
(a < b) ? a : b
++i
anz += 10
```

- **Funktionsaufrufe und Parameterübergaben** kosten gegenüber dem Einlesen, Verarbeiten oder Schreiben von Daten praktisch keine Zeit, man darf also nicht anfangen, die Anzahl der Funktionen oder Funktionsaufrufe zu reduzieren. Der Verlust an Übersichtlichkeit wiegt die paar Mikrosekunden Gewinn nicht auf.
Eventuell können extrem häufig aufgerufene „kleine“ Funktionen als Makro oder `inline`-Funktion (C++) geschrieben werden.
- Über **spezielle Maßnahmen zur Optimierung** darf erst nachgedacht werden, wenn ein Programm stabil und fehlerfrei läuft, da Optimierung eine sehr fehleranfällige Tätigkeit ist. Oft hilft bereits das Merken des letzten Arguments und seines zugehörigen Ergebnisses einer oft aufgerufenen Funktion, um die Performance zu verbessern.
- **Weitere Optimierungsmöglichkeiten** sind:
 - SQL-Statements tunen (Index, Join, temporäre Tabelle)
 - Ausgangsdaten verdichten oder besser organisieren
 - Geeignete Datenstrukturen
 - Intelligenterer Algorithmen (Quick Sort, Binäre Suche, Hashen)
 - Daten nur einmal lesen und im Speicher halten
 - Schnellere Hardware (Rechner, Platten, Speicher)
 - Spezialhardware (Software in Hardware)
 - Parallelisieren auf Mehrprozessormaschine
 - Threads (statt Prozesse)
- Eine naheliegende Optimierung, nämlich **gepuffertes Lesen/Schreiben** von/auf Dateien, macht das Betriebssystem bereits selber (sofern man die ANSI-C-Funktionen verwendet). Darum muss man sich also auch nicht mehr kümmern.

10 Sonstiges

- Keinen **überflüssigen Code** oder **überflüssige Variablen** stehen lassen, Programme werden dadurch extrem unübersichtlich. Soll alter Code aufgehoben werden, ist dies am besten durch ein Archivierungstool wie RCS zu erreichen.
- Die in den **man-Pages** zu C-Funktionen oder UNIX-Kommandos enthaltenen Verweise (SEE ALSO) auf andere Funktionen oder Kommandos beachten. Oft kann hierüber eine Funktion mit genau der gewünschten Funktionalität ermittelt werden.
- In einem Quellcode oder einem Projekt **mehrfach benötigten Code nicht kopieren** (und leicht verändern), sondern ihn stattdessen in eine (parametrisierte) Funktion packen, die dann mit entsprechenden Argumenten aufgerufen wird.
- Keine Funktionen der C-Bibliothek **nachprogrammieren**.
- Keine **Variablenitis**, es ist sehr unübersichtlich und fehleranfällig, mehrere Variablen einzuführen, die letzten Endes den gleichen Sachverhalt darstellen (z.B. eine Variable `cnt` und eine weitere `cnt_minus_one`, die den um eins verminderten Wert enthält oder eine Indexvariable und gleichzeitig eine Zeigervariable auf das entsprechende Arrayelement). Negativbeispiel:

```
char text[80];
int found;
int cnt;

found = cnt = 0;
while (fgets(text, 80-1, stdin) != EOF) {
    found = 1;
    text_array[cnt++] = strdup(text);
}

if (found)
    printf("%d Zeilen gefunden\n", cnt);
else
    printf("Keine Zeilen gefunden\n");
```

Die Variable `found` kann durch `cnt` vollkommen ersetzt werden:

```
int cnt;

cnt = 0;
while (fgets(text, 80-1, stdin) != EOF) {
    text_array[cnt++] = strdup(text);
}

if (cnt > 0)
    printf("%d Zeilen gefunden\n", cnt);
else
    printf("Keine Zeilen gefunden\n");
```

11 Vorschläge für Funktions- und Variablennamen

11.1 Generelle Konventionen

Eine vollständige Auflistung aller generellen Namenskonventionen ist in Abschnitt 4 auf Seite 9 zu finden, hier noch einmal die wichtigsten:

- Funktionsnamen werden in Groß/Kleinschreibung ohne Unterstrich geschrieben, ihre Strukturierung erfolgt durch Großbuchstaben und sie enthalten ein **Tätigkeits/Eigenschaftswort**.
- Variablennamen werden in Kleinschreibung mit Unterstrich geschrieben, ihre Strukturierung erfolgt durch Unterstriche und sie enthalten **kein Tätigkeits/Eigenschaftswort**.
- Alternativ kann unter Verwendung der **Ungarischen Notation** für Variablen auch ein kleingeschriebener Typpräfix am Variablenanfang und danach der Variablenname in Groß/Kleinschreibung ohne Unterstriche stehen (MicroSoft-Stil).
- **Einbuchstabile Variablen** sind nur lokal als Schleifenzähler erlaubt.
- **Abkürzungen** müssen sinnvoll, verständlich und immer gleich erfolgen (2=to und 4=for sind erlaubt). Sie haben durch
 - Abschneiden des Wortendes und/oder
 - Weglassen der Vokale oder stummer/doppelter Konsonantenzu erfolgen.

11.2 Vorschläge für Tätigkeitswörter von Funktionen

Tätigkeitswort direkt mit dem Objekt- oder Datentypnamen zum konkreten Funktionsnamen verketteten (ohne Unterstrich dazwischen), die einzelnen Namensteile dabei mit großem Anfangsbuchstaben und den Rest klein schreiben. Z.B. `DumpLine()`, `SaveText()`, `GenHeader()`, `InitVars()`, `GetOptions()`, `OpenFiles()`, ...

- Abort, Add, Alloc(ate), Allow, Append, Apply, Assert
- Backup, Begin, Build
- Calc(ulate), Call, Cancel, Change, Ch(ek), Cl(ea)r, Close, Coll(ect), Combine, C(o)mp(are), Connect, Continue, C(ou)nt, Copy, Crea(te), Cut
- Dec(rement), D(e)b(u)g, Del(ete), Destroy, Die, Div(ide), Dump, Dup(licate)
- End, Enter, Erase, Err(or), Exch(ange), Exec(ute), Exit, Expand
- Fill, Find, First, Forbid, Free
- Gen(erate), Get
- Hash, Hide

- Inc(rement), Init(ialize), Ins(ert)
- Join
- Keep, Kill
- Last, Leave, Link, Load, Lock
- Make, Map, Mark, Match, Merge, M(es)s(a)g(e), Mul(tiply)
- Name, Next
- Open, Order
- Parse, Pop, Prev(ious), Print, Pull, Push, Put
- Query, Quote
- Read, Recall, Receive, Recover, Refresh, Release, Rem(ove), Renew, Rep(eat), Replace, Report, Resize, Retrieve, Ret(urn)
- Save, Say, Scale, Scan, Search, Seek, Sel(ect), Send, Set, Size, Skip, Solve, S(o)rt, Store, Subst(itute), Suspend, Sub(tract)
- Take, T(e)st, Trace, Trans(fer/form/late), Traverse
- Undel(ete), Unlink, Upd(ate)
- Validate, Verify
- Wait, Warn(ing), W(ei)ght, Write
- eXec(ute)
- Y(ie)ld
- Zero, Zip

11.3 Vorschläge für Eigenschaftswörter von Funktionen

Die ursprünglich in FORTRAN verwendeten Abkürzungen `EQ`, `NE`, `LT`, `LE`, `GT` und `GE` werden als **Standardkürzel für die Vergleichsart** allgemein verstanden. (z.B. mit dem Datentyp zu einem Funktionsnamen wie `StringEQ` zusammensetzen). `Has` und `Is` entsprechen der **Enthaltensein-/Vererbungsrelation** zwischen zwei Objekten.

- Contains,
- Diff(erent),
- EQ(ual)
- G(reater)E(qual), G(reater)T(han)
- Has

- ls
- L(ess)E(qual), L(ess)T(han), Like
- N(ot)E(qual)

11.4 Zu vermeidende Funktionsnamen

- !!!

11.5 Vorschläge für Variablenamen (oder Teilen davon)

Getrennt durch `_` (Unterstrich) mit anderen Namen oder untereinander zu konkreten Variablenamen verknüpfen und diese vollständig klein schreiben, z.B. `main_struct`, `prev_elem`, `dbg_flag`, `err_flag`, `op_stack`, `line_buffer`, `line_cnt`, `call_stack`, ... (oder bei Verwendung der Ungarischen Notation z.B. `mainStruct`, `pPrevElem`, `nDbgFlag`, `nErrFlag`, `aOpStack`, `pLineBuffer`, `nLineCnt`, `callStack`).

- after, amount, arr(ay), av(era)g(e)
- back(ward), before, beg(in), bot(tom), buf(fer)
- ch(ar), child, cl(as)s, code, connection, const(ant), c(o)py, c(ou)nt
- dec(rement), diff(erence), down, dest(ination)
- edge, elem(ent), end, err(or)
- factor, feat(ure), f(ie)ld, file, first, fl(a)g, f(or)m(a)t, forward
- gr(ou)p
- hash
- inc(rement), item
- leaf, l(a)st, l(e)ft, len(gth), l(e)v(e)l, line, l(i)st
- main, master, mat(rix), max(imum), member, min(imum)
- name, new, n(e)xt, node, n(umbe)r
- obj(ect), old, op(erand), op(erator)
- parent, p(oin)t(er), pred(ecessor), prev(ious), prod(uct)
- queue
- rec(ord), r(i)g(h)t, root
- save, sign, slave, s(ou)rc(e), stack, start, stop, str(ing), struc(ture), succ(essor), sum
- top, t(e)mp(orary), tree

- up, us(e)r
- var(iable), vec(tor), vers(ion)
- warn(ing)

11.6 Zu vermeidende Variablennamen

- **Einbuchstabile Variable** nur lokal als Schleifenzähler verwenden.
- Folgende Namen nicht verwenden, da sie nichts aussagen:
 - counter, zaehler, eingabe, ausgabe. . .
 - flag, flag2, . . .
 - help, helper, help2, . . .
 - save, save2, . . .
 - tmp, tmp2, . . .
 - var, var1, . . .
 - v, v2, . . . (analog für alle anderen Buchstaben)
 - ii, iii, i2, ii3, . . . (analog für alle anderen Buchstaben)

12 Erweiterung von C

- Folgende Makros können definiert werden, um **fehlende C-Kontrollstrukturen** zu ergänzen bzw. eine einfachere und eindeutige Formulierung zu erreichen:

```
#define repeat      do
#define until(x)   while (!(x))
#define loop       for (;;)          /* oder while (1) */
#define elsif      else if
#define unless(x)  if (!(x))
```

Damit lassen sich REPEAT-UNTIL-Schleifen, Endlosschleifen mit mehreren Abbruchstellen, Mehrfach-Verzweigungen und negative Bedingungen übersichtlicher programmieren:

```
x = start;
repeat {
    ...
    x = x->next;
} until (x == NULL);

loop {
    ...
    if (ende == TRUE)
        break; /* loop */
    ...
}

if (x == 0)
    printf("Fall a\n");
elseif (x == 1)
    printf("Fall b\n");
else
    printf("Fall c\n");

unless (cnt == 0)
    sum /= cnt;
```

- Die Einführung einer **eigenen Syntax** (z.B. ähnlich PASCAL) durch Makros ist nicht gestattet. Negativbeispiel:

```
#define STRING char*
#define IF      if(
#define THEN    ){
#define ELSE    }else{
#define FI      };
#define WHILE   while(
#define DO      ){
#define OD      };
#define INT     int
#define BEGIN   {
#define END     }
#define EQ      ==
#define RETURN  return
```

```
...
INT compare (STRING s1, STRING s2)
BEGIN
    WHILE *s1++ EQ *s2
    DO IF *s2++ EQ 0
        THEN RETURN 0
        FI
    OD
RETURN *--s1 - *s2
END
```

In dieser Syntax ist (leider!) die **Bourne-Shell** geschrieben. Änderungen daran sind aus diesen (und einigen anderen) Gründen äußerst mühselig.

13 Der Datentyp Boolean

- Vergleiche mit `<` `<=` `>` `>=` `==` `!=` und logische Ausdrücke mit `&&` `||` `!` liefern immer die Werte 1 (für „Wahr“) bzw. 0 (für „Falsch“) zurück.
- Vom C-Compiler wird jeder Wert $\neq 0$ als „Wahr“ interpretiert, der Wert 0 als „Falsch“.
- Der Datentyp `Boolean` wird von WINDOWS und vielen anderen Programmpaketen folgendermaßen festgelegt:

```
typedef BOOL int;
#define TRUE 1
#define FALSE 0
```

Statt `BOOL`, `TRUE` und `FALSE` werden auch `Bool/bool`, `True/true` und `False/false` verwendet. Da in C allerdings jeder Wert $\neq 0$ logisch wahr ist, darf ein Vergleich

```
if (flag == TRUE) ...
```

nur bei solchen Variablen erfolgen, die sicher nur mit dem Wert `TRUE` oder `FALSE` belegt sind.

- Die Formulierungen

```
if (flag) ...
if (!flag) ...
```

sind zu verwenden, um auf **Wahrheit** oder **Falschheit** abzuprüfen.

- Im ANSI99-C-Standard wird für diesen Zweck der Datentyp `_Boolean` eingeführt.

14 Rekursive Datentypen

- Die **Schwierigkeit bei der Definition** eines rekursiven Datentyps liegt darin, daß während seiner Definition bereits auf ihn Bezug genommen wird. Seine Größe (der von einer Variablen seines Typs eingenommene Speicherplatz) liegt aber zu diesem Zeitpunkt noch nicht fest. **Als Ausweg bieten sich Zeiger an.** Da sie immer die gleiche Größe haben, können sie bereits verwendet werden, wenn der Datentyp, auf den sie zeigen, noch nicht (fertig) definiert ist.
- Das folgende Codestück zeigt die Definition eines rekursiven Datentyps `NODE` mit Hilfe einer sogenannten **Vorwärtsdeklaration**. Diese führt nur den Namen des Datentyps ein, legt aber seine Struktur noch nicht fest. Anschließend können bereits Zeiger auf diesen Datentyp vereinbart werden, obwohl er noch nicht vollständig definiert ist.

```
typedef struct NODE;      /* Vorwaertsdeklaration */

typedef struct {
    int data;
    NODE* next;          /* Zeiger auf NODE */
} NODE;
```

- Alternativ können die beiden Definitionen auch zu einer Definition zusammengezogen werden.

```
typedef struct NODE_tag {
    int data;
    struct NODE_tag* next;
} NODE;
```

- Bei wechselseitig rekursiven Datentypen kann nur noch mit Vorwärtsdeklaration gearbeitet werden.

```
typedef struct PREV;     /* Vorwaertsdeklaration */
typedef struct SUCC;     /* Vorwaertsdeklaration */

typedef struct {
    int data;
    SUCC* succ;
} PREV;

typedef struct {
    int data;
    PREV* prev;
} SUCC;
```

15 Variable Argumentlisten

- In C sind variable Argumentlisten möglich. Ein Beispiel für die Programmierung von Zugriffen auf variable Argumentlisten ist die folgende Funktion `Error()`. Sie erwartet analog `printf()` einen Formatstring und eine dazu passende Anzahl von Argumenten, um eine Fehlermeldung auszudrucken und das Programm anschließend abzubrechen.

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

void
Error(int err_code, char *format, ...)
{
    va_list argp;

    fprintf(stderr, "error(%d): ", err_code);
    va_start(argp, format);
    vfprintf(stderr, format, argp);
    va_end(argp);
    fprintf(stderr, "\n");

    exit(err_code);
}
```

- Beispiele für ihren Aufruf sind

```
Error(1, "Datei '%s' nicht gefunden", file_name);
Error(13, "Laenge %d > max. Laenge %d", length, max_length);
```

- Bei der Übergabe eines **Null-Zeigers**, einer **Fließkommazahl** oder einer `long`-Variablen in variablen Argumentlisten ist `(void*)NULL` statt `NULL`, `10.0` statt `10` und `0L` statt `0` zu schreiben, sonst wird nur eine Zahl vom zu kleinen Datentyp `int` übergeben.

```
FALSCH: Error(13, "%f %ld\n", 456, 789, NULL);
RICHTIG: Error(13, "%f %ld\n", 456.0, 789L, (void*)NULL);
```

- Die **Anzahl an übergebenen Argumenten** und ihre Datentypen sind in der aufgerufenen Funktion leider nicht ermittelbar. Entweder wird ein Verfahren analog `printf()` verwendet oder als erster Wert die Anzahl der Argumente mit übergeben oder an das Ende der Argumentliste ein besonderer **Sentinel-Wert** (Wächter) wie z.B. `NULL` oder `EOF` gestellt, der abgeprüft werden kann.

```
Error(13, "1.Arg", "2.Arg", "3.Arg", (void*)NULL);
```

- Die K&R-Headerdatei `vararg.h` ist nicht mehr zu verwenden, sie wird durch `stdarg.h` vollständig ersetzt.

16 Suchen und Sortieren

- qsort, lsearch, bsearch, Vergleichs-Funktion, Funktions-Zeiger

!!!

17 Datentypen und ungarische Notation

- In der ungarischen Notation wird jedem Variablennamen ein **kleingeschriebener Präfix** vorangestellt, der den Datentyp der Variablen beschreibt. Ist kein Präfix vorhanden, so handelt es sich meistens um einen `INT`-Wert. Die in der Tabelle aufgelisteten Datentypen treten in `WINDOWS` an die Stelle der C-Standard-Datentypen `char`, `int`, `long`, ... Sie sind bei der Programmierung von `WINDOWS`-Anwendungen zu verwenden.

Präfix	Datentyp	Größe	Vorz.	Beschreibung
–	VOID	–	–	Kein Wert!
b	BOOL	16 Bit	signed	TRUE(= 1 bzw. $\neq 0$) oder FALSE(= 0)
c	CHAR	8 Bit	signed	–128...127
n	INT	16 Bit	signed	–32768...32767
l	LONG	32 Bit	signed	–2147483648...2147483647
by	BYTE	8 Bit	unsigned	0...255
w	WORD	16 Bit	unsigned	0...65535
f	WORD	16 Bit	unsigned	Bitfeld
dw	DWORD	32 Bit	unsigned	0...4294967295
–	FLOAT	32 Bit	signed	Fließkommazahl
–	DOUBLE	64 Bit	signed	Fließkommazahl
h	HANDLE	16 Bit	unsigned	Handle eines <code>WINDOWS</code> -Objekts (auch <code>HMENU</code> , <code>HWND</code> , <code>HCURSOR</code> , ...)
fn	NEARPROC/FARPROC			Funktion
sz	STR		–	C-String (string zero)
p	P..	16 Bit	–	NEAR-Zeiger (pointer)
lp	LP..	32 Bit	–	FAR-Zeiger (long pointer)

- Leider sind die in der Literatur angegebenen **Präfixe nicht einheitlich**, siehe auch Charles Petzold, *Programmierung unter Microsoft Windows 3.1*, Microsoft Press und Marcellus Buchheit, *Windows-Programmierbuch*, Sybex Verlag.

18 Vorrangtabelle

Ein **vollständiges Verständnis der Vorrangregeln** der C-Operatoren ist äußerst wichtig. Bei falscher Interpretation von Ausdrücken sind sonst die herrlichsten Fehler möglich. Zum schnellen Nachsehen kann z.B. die folgende Liste am Bildschirm angebracht werden.

Typ	Prio	Operatoren	Assoziativität
Einstellig	15	[] . -> ()	links nach rechts
	14	! ~ ++ -- & * (TYP) sizeof + -	rechts nach links
Arithmetisch	13	* / %	links nach rechts
	12	+ -	links nach rechts
Bitshift	11	<< >>	links nach rechts
Vergleich	10	< <= > >=	links nach rechts
	9	== !=	links nach rechts
Bitweise	8	&	links nach rechts
	7	^	links nach rechts
	6		links nach rechts
Logisch	5	&&	links nach rechts
	4		links nach rechts
Bedingung	3	?:	rechts nach links
Zuweisung	2	= += -= *= /= %= &= ^= = <<= >>=	rechts nach links
Sequenz	1	,	links nach rechts

- Jedem Operator ist eine **Priorität** zugeordnet, in einem Ausdruck eingesetzte Operatoren **unterschiedlicher Priorität** werden nach abfallender Priorität ausgeführt.
- Benachbarte Operatoren **gleicher Priorität** werden gemäß ihrer **Assoziativität** von links nach rechts oder umgekehrt ausgeführt.
- Trotz der 45 Operatoren und 15 Vorränge ist die Tabelle **leicht zu behalten**. Die Vorränge und die Assoziativitäten wurden nämlich (fast) so gewählt, wie man es intuitiv erwarten würde, bis auf 2 Ausnahmen: Die Bit-Operatoren | & ^ und die Bit-Shift-Operatoren << >>. **Tipp**: wenn man diese **immer klammert**, dann passiert auch hier nichts.
- Beispiele für die Auswertungsreihenfolge gemäß Priorität:

```

a * b + c / d      entspricht      (a * b) + (c / d)
a < b && c > d      entspricht      (a < b) && (c > d)
*a[1]              entspricht      *(a[1])
*a++              entspricht      *(a++)
&a->b              entspricht      &(a->b)
*--a              entspricht      *(--a)

```

- **Assoziativität von links nach rechts** heißt: Sind hintereinanderstehende Ausdrücke über Operatoren der gleichen Priorität verknüpft (ohne explizite Klammerung), so wird mit der Auswertung beim am weitesten links stehenden Operator begonnen:

```

a + b - c + d      entspricht      ((a + b) - c) + d
a->b.c->d          entspricht      ((a->b).c)->d
a[1][2]           entspricht      (a[1])[2]

```

- Entsprechend gilt für die **Assoziativität von rechts nach links**:

```
a = b = c = d      entspricht      a = (b = (c = d))
a ? b : c ? d : e  entspricht      (a ? b : (c ? d : e))
**a               entspricht      *(**a)
```

19 Ungewöhnliche Operatoren

Ungewöhnlich gegenüber anderen Programmiersprachen sind folgende Operatoren:

- Der **logische Negations-Operator** `!` invertiert einen Booleschen Wert. Der **bitweise Invertierungs-Operator** `~` invertiert alle Bits eines Wertes:

```
a = !b;           /* wahr -> falsch, falsch -> wahr */
a = ~b;           /* 10001110 -> 01110001 */
```

- Die **Inkrement/Dekrement-Operatoren** `++` `--` haben eine **Präfix-** und eine **Postfix-Form**. Werden beide Operatoren nur zusammen mit einer Variablen verwendet, besteht kein Unterschied zwischen ihnen:

```
++i;             /* i = i + 1 */
i++;            /* i = i + 1 */
--i;            /* i = i - 1 */
i--;            /* i = i - 1 */
```

Werden die beiden Operatoren in einem **Ausdruck** verwendet, dann unterscheiden sich die Wirkungen der beiden Varianten:

```
a = ++i;        /* i = i + 1;   a = i      */
a = i++;        /* a = i;       i = i + 1 */
a = --i;        /* i = i - 1;   a = i      */
a = i--;        /* a = i;       i = i - 1 */
```

- Der **sizeof-Operator** `sizeof` ermittelt den Platzbedarf eines Datentyps oder einer Variablen in Byte:

```
int var;
i = sizeof(int);
i = sizeof(var);
```

- Der (explizite) **Cast-Operator** `(TYP)` konvertiert einen Datentyp in einen anderen:

```
float f;
int i;
f = (int) i;           /* int -> float */

char *cp;
int *ip;
cp = (char*) ip;      /* char* -> int* */
```

- Der **Modulo-Operator** `%` ist nur auf Ganzzahlen anwendbar und errechnet den ganzzahligen **Divisionsrest**:

```
17 % 5      /* ergibt 2 wg. 17 : 5 = 3 Rest 2 */
```

- **Kleiner/Größer-Vergleiche** haben einen **höheren Vorrang** als die **Gleich/Ungleich-Vergleiche**. Dies erlaubt folgende Konstruktion (deren Bedeutung sich nicht unbedingt leicht erschließt):

```
a <= b == b <= c      /* (a <= b && b <= c) || (a > b && b > c) */
```

- Der **Bedingte Ausdruck** `?:` ist ein dreiwertiger Operator, er wertet den 2. oder 3. Ausdruck abhängig vom Wert des 1. Ausdrucks aus.

```
max = (a > b) ? ++a : ++b; /* entweder a oder b inkrementiert */
```

- Die 10 Operatoren `+ - * / % & ^ | << >>` sind als **verkürzte Operation** mit der **Zuweisung** kombinierbar:

```
i += 1;      /* i = i + 1 */
i -= 2;      /* i = i - 2 */
i *= 3;      /* i = i * 3 */
i /= 4;      /* i = i / 4 */
i %= 5;      /* i = i % 5 */
i &= 6;      /* i = i & 6 */
i ^= 7;      /* i = i ^ 7 */
i |= 8;      /* i = i | 8 */
i <<= 9;     /* i = i << 9 */
i >>= 10;    /* i = i >> 10 */
```

- Das **Komma** `,` ist ein **schwaches Semikolon**, das — im Gegensatz zu diesem — in einem Ausdruck erlaubt ist. Er sorgt dafür, dass erst der linke Ausdruck und dann der rechte ausgewertet wird. Ergebnis des ganzen Ausdrucks ist der Wert des rechten Ausdrucks. Da der Komma-Operator den niedrigsten Vorrang hat, werden alle anderen Operatoren vor ihm ausgewertet, solange nicht geeignet geklammert wird.

```
i = 10.2;    /* i = 10 (Dezimalpunkt!) */
i = 10,2;    /* i = 10 */
i = (10,2);  /* i = 2 */
j = (i = 10,2); /* i = 10, j = 2 */
j = (i = (10,2)); /* i = 2, j = 2 */
```

Typischer Anwendungsfall des Komma-Operators sind `for`-Schleifen mit zwei oder mehr Laufvariablen oder Makros, die mehrere Anweisungen zusammenfassen.

```
for (i = 0, j = 0; i > 0 && j > 0; ++i, ++j) ...
#define INCBOTH(i,j) ++i, ++j
```

- Die **Bit-Operatoren** `& | ~` nicht mit den **Logischen/Booleschen Operatoren** `&& || !` verwechseln. Erstere arbeiten auf den einzelnen Bits der Werte, letztere arbeiten nur mit Wahrheitswerten nach folgender Logik:
 - **Falsch/False** = Wert 0
 - **Wahr/True** = Alle anderen Werte (ungleich 0).

20 Operatoren mit problematischer Priorität

- Die **Shift-Operatoren** `<<` und `>>` können zwar als Multiplikation mit 2 und Division durch 2 betrachtet werden, haben aber einen geringeren Vorrang als die Additionsoperatoren `+` und `-`. Beim Rechnen mit geschifteten Werten also klammern:

```
a << 4 + b >> 8      entspricht      (a << (4 + b)) >> 8
```

- Ebenso sind die **Bit-Operatoren** `&`, `|` und `^` von geringerem Vorrang als die Vergleichsoperatoren. Beim Vergleich von isolierten Bits also klammern:

```
x & 0x11 > 0        entspricht      x & (0x11 > 0)
```

- Bei der Verwendung eines Zeiger `ps` auf eine Datenstruktur muss man beim **Komponentenzugriff** ebenfalls aufpassen. `*ps.elem` liefert wegen dem Vorrang `.` vor `*` nicht das richtige Ergebnis. Entweder man klammert `(*ps).elem` oder man verwendet den **Pfeiloperator** `ps->elem`.

21 Operatoren mit fixer Auswertungsreihenfolge

- Die Berechnung eines über die **Logik-Operatoren** `&&` und `||` verknüpften logischen Ausdrucks erfolgt schrittweise gemäß Vorrang und von links nach rechts und wird sofort abgebrochen, wenn das Endergebnis *wahr* oder *falsch* feststeht (**verkürzte Auswertung, shortcut/short circuit evaluation**). D.h. folgende `for`-Schleife ist korrekt formuliert (es findet kein Zugriff auf das nicht existierende Arrayelement `arr[MAXLEN]` statt):

```
char* arr[MAXLEN];

for (i = 0; i < MAXLEN && arr[i] != NULL; ++i)
    arr[i] = ...;
```

- Beim **Bedingten Ausdruck** `?:` wird zuerst der Vergleichsausdruck und dann *genau einer* der beiden Wertausdrücke berechnet:

```
max = (a > b) ? ++a : ++b; /* entweder a oder b inkrementiert */
```

Dies entspricht der etwas längeren Formulierung:

```
if (a > b)
    ++a;
else
    ++b;
```

- Ein **bedingter Ausdruck** kann — im Gegensatz zu einer `if`-Abfrage — auch innerhalb eines anderen Ausdrucks oder als Funktionsargument verwendet werden:

```

cp = (x % 2 == 0) ? "gerade" : "ungerade";

printf("Zeiger cp zeigt auf %s\n, (cp == NULL) ? "(null)" : cp);

```

Das Klammern der Bedingung in diesen Beispielen ist zwar nicht notwendig, erhöht aber die Übersichtlichkeit.

- Nur die Operatoren `&&`, `||`, `?:` und `,` (Komma) sowie die Zuweisungen `=` `+=` `-=` `*=` `/=` `%=` `&=` `^=` `|=` `<<=` `>>=` garantieren eine **feste Auswertungsreihenfolge** ihrer Operanden (in den folgenden Beispielen ist die garantierte Auswertungsreihenfolge durch 1 bzw. 2 angedeutet).

```

1 && 2
1 || 2
1 ? 2a : 2b
1, 2
2 = 1
2 *= 1

```

Bei allen anderen Operatoren ist die Auswertungsreihenfolge **compilerabhängig** . Insbesondere die Aufrufreihenfolge von als Operanden verwendeten Funktionen und die Auswertungsreihenfolge von Inkrement-/Dekrementoperationen ist compilerabhängig.

```

f() + g()          /* Aufruf-Reihenfolge von f und g unbestimmt */
++i + i--         /* Auswertung-Reihenfolge von ++i und i-- unbestimmt */

```

22 Auswertungs-Reihenfolge beeinflussen

- Durch Aufspalten eines Ausdrucks in **Unterausdrücke mit Zwischenvariablen** kann eine bestimmte Auswertungsreihenfolge erzwungen werden. Ob z.B. im folgenden Beispiel zuerst `g()` und dann `h()` ausgewertet wird, bevor ihre beiden Ergebnisse addiert werden, oder umgekehrt, ist compilerabhängig.

```

y = g(x) + h(x);

```

Im folgenden Beispiel wird bei gleichem Ergebnis auf jeden Fall zuerst `g()` und dann `h()` ausgewertet.

```

y = g(x);
y += h(x);

```

- **Tipp:** Im Zweifelsfall komplexe Ausdrücke lieber in Teilausdrücke zerlegen oder lieber eine Klammer zuviel als zuwenig verwenden, um den gewünschten Vorrang auszu-drücken. Allzu große Klammerngebirge sind allerdings auch nicht mehr überschaubar.

23 C-Deklarationen

Folgende Deklarationen sind auf Anhieb verständlich.

```
int i;
char* p;
```

i ist eine Variable vom Typ `int`, *p* ist ein Zeiger auf eine Variable vom Typ `char`. Die beiden folgenden Deklarationen sind ebenfalls noch verständlich.

```
int* ia[3];
int (*ib)[3];
```

ia ist eine Feld von 3 Zeigern auf Variablen vom Typ `int`, *ib* ist ein Zeiger auf ein `int`-Feld der Länge 3. Deklarationen dieser Form rufen allerdings bereits Unsicherheit hervor, es fehlt das richtige Verständnis, man hat sie meist auswendig gelernt.

Bei noch komplexeren Deklarationen ist oft Raten der einzige Ausweg. Das ist schade, da die Theorie der Deklarationen eigentlich sehr einfach und überschaubar ist. Deklarationen beruhen auf der **Hierarchie der C-Operatoren**, d.h. es gilt für die Auswertungsreihenfolge.

Operator	Bedeutung	Priorität	Auswertung
(...)	Klammerung	höchste	Reihenfolge ändern
()	Funktion	mittlere	links nach rechts
[]	Array	mittlere	links nach rechts
*	Zeiger	niedrigste	rechts nach links

23.1 Lesen von C-Deklarationen

In der hierdurch definierten Reihenfolge sind die Teile einer Deklaration gemäß folgenden Regeln zu lesen.

1. **Attribute** sind [], () und *, also Feld, Funktion und Zeiger.
2. Stellen Sie den Bezeichner der Deklaration fest. Sagen Sie „**Bezeichner ist**“, wobei Bezeichner der Name der Variablen ist.
3. Schauen Sie rechts vom Bezeichner nach () oder [] (können auch fehlen):
 - Sagen Sie „**ein Feld von**“, wenn Sie [] sehen.
 - Sagen Sie „**ein Feld von x**“, wenn Sie [x] sehen.
 - Sagen Sie „**ein x-mal-y Feld von**“, wenn Sie [x][y] sehen.
 - Sagen Sie „**ein x-mal-y mal ... Feld von**“, wenn Sie [x][y][...] sehen.
 - Sagen Sie „**Funktionen mit Rückgabewert**“, wenn Sie () sehen und das zuletzt gefundene Attribut [] war.
 - Sagen Sie „**eine Funktion mit Rückgabewert**“, wenn Sie () sehen.
4. Schauen Sie auf die linke Seite des Bezeichners (nur Sterne sind interessant, können auch fehlen, Klammerung beachten):
 - Sagen Sie „**Zeiger auf**“ für jeden *, wenn das zuletzt gefundene Attribut [] war.
 - Sonst sagen Sie „**ein Zeiger auf**“, wenn Sie * sehen.
5. War der bisher gelesene Ausdruck eingeklammert, betrachten Sie ihn als abgearbeitet, ignorieren die Klammern und fahren bei 3. fort.

6. Es muß ein terminierendes Attribut der Form `char`, `short`, `int`, `long`, `float`, `double`, `struct`, `union`, ... oder ein selbstdefinierter Typ, evtl. versehen mit einem der Modifizierer `signed`, `unsigned`, `static`, `auto`, `register`, `extern` übrigbleiben.
- Sagen Sie „eine Struktur vom Typ `y`“ für `struct y`.
 - Sagen Sie „eine Union vom Typ `y`“ für `union y`.
 - Sagen Sie „eine Aufzählung vom Typ `y`“ für `enum y`.
 - Sagen Sie „ein Typ“ für den/die noch vorhandenen Typ(en)

Beispiele

```
int i;
  ↑i ist
↑ein Integer
```

```
int *i;
  ↑i ist
  ↑ein Zeiger auf
↑Integer
```

```
int *i[3];
  ↑i ist
  ↑ein Feld von 3
  ↑Zeigern auf
↑Integer
```

```
int (*i)[3];
  ↑i ist
  ↑ein Zeiger auf
  ↑ein Feld von 3
↑Integern
```

```
int *i();
  ↑i ist
  ↑eine Funktion mit Rückgabewert
  ↑Zeiger auf
↑Integer
```

```
int (*i)();
  ↑i ist
  ↑ein Zeiger auf
  ↑eine Funktion mit Rückgabewert
↑Integer
```

```
int **i;
  ↑i ist
  ↑ein Zeiger auf
  ↑einen Zeiger auf
↑Integer
```

```

int *(*i)();
    ↑i ist
    ↑ein Zeiger auf
    ↑eine Funktion mit Rückgabewert
    ↑Zeiger auf
    ↑Integer

int *(*i[])();
    ↑i ist
    ↑ein Feld von
    ↑Zeigern auf
    ↑Funktionen mit Rückgabewert
    ↑Zeiger auf
    ↑Integer

int *(*(*i)())[10];
    ↑i ist
    ↑ein Zeiger auf
    ↑eine Funktion mit Rückgabewert
    ↑Zeiger auf
    ↑ein Feld von 10
    ↑Zeigern auf
    ↑Integer

int *(*i[5])[10];
    ↑i ist
    ↑ein Feld von 5
    ↑Zeigern auf
    ↑ein Feld von 10
    ↑Zeigern auf
    ↑Integer

```

23.2 Schreiben von C-Deklarationen

Analog zu den Regeln für das Lesen von Deklarationen lassen sich folgende Regeln für das Schreiben von Deklarationen angeben. Hat man eine umgangssprachliche Beschreibung der Deklaration, läßt sich diese damit in ihre syntaktische Form umsetzen.

1. Schreiben Sie „**Bezeichner**“, wobei Bezeichner der Name der Variablen ist.
2. Merken Sie sich immer, ob das letzte Attribut, das Sie geschrieben haben, ein „*“ war.
3. Schreiben Sie „*“ zur Linken dessen, was Sie bisher geschrieben haben, solange Sie in der Beschreibung **Zeiger auf** sehen.
4. Klammern Sie das bisher geschriebene, wenn Sie zuletzt einen „*“ geschrieben haben.
5. Schreiben Sie „[x]“ zur Rechten dessen, was Sie bisher geschrieben haben, wenn Sie **Feld von x** sehen,
 - schreiben Sie „[x] [y]“ rechts, wenn Sie **ein x-mal-y Feld von** sehen,

23.3 Einschränkungen

- Es gibt **keine Felder von Funktionen**, d.h die Deklaration `int a[5]()` ist ungültig. Es gibt aber Felder von Zeigern auf Funktionen, d.h. die Deklaration `int (*a[5])()` ist gültig.
- **Eine Funktion kann kein Feld zurückliefern**, d.h die Deklaration `int a()[]` ist ungültig. Eine Funktion kann aber einen Zeiger auf ein Feld zurückliefern, d.h. die Deklaration `int (*a())[]` ist gültig.
- **Eine Funktion kann keine andere Funktion als Ergebnis liefern**, d.h die Deklaration `int a()()` ist ungültig. Eine Funktion kann aber einen Zeiger auf eine andere Funktion zurückliefern, d.h. die Deklaration `int (*a())()` ist gültig.

23.4 Typdefinitionen und Casts

- Bei einer Typdefinition ist statt dem Variablennamen der **Typname** einzusetzen und vor den ganzen Ausdruck `typedef` zu schreiben. Beispiel

```
int* f(); /* Funktion f mit Rueckgabewert Zeiger auf int */

typedef int* FUNC_TYPE(); /* Analoge Typdefinition */
FUNC_TYPE f; /* 1:1 obige Deklaration */
```

- Für einen **cast (Typumwandlung)** ist die Deklaration einer entsprechenden Variablen (unter Weglassen des Variablennamens und in Klammern gesetzt), vor die umzuwandelnde Variable zu setzen. Beispiel

```
(* (void(*)()) 0xFFFF0)();
```

Eine Funktion (ohne Rückgabewert und ohne Argumente), deren Startadresse an der Speicherstelle `0xFFFF0` liegt, soll aufgerufen werden. Dazu ist der Integerwert `0xFFFF0` in einen Zeiger auf eine Funktion ohne Rückgabewert und ohne Argumente umzuwandeln. Das erledigt der `cast`-Ausdruck `(void(*)())`. Zum Aufruf ist anschließend noch der Zeiger zu dereferenzieren (`*`) und der Funktionsaufruf auszulösen (`()`). Eine identische, aber übersichtlichere Formulierung des Aufrufs läßt sich durch Einführen eines eigenen Typs `FPTR` erreichen

```
typedef void (*FPTR)(); /* Typdefinition */
(* (FPTR) 0xFFFF0)(); /* Aufruf */
```

- Analog sind in ANSI-C bei der Angabe von Argumenttypen in Funktionsdeklarationen für jedes einzelne Argument seine Typdeklaration (wahlweise mit oder ohne Variablenname) anzugeben. Beispiel

```
void (*signal(int, void(*) (int)))(int);
```

Die Funktion `signal` hat zwei Argumente, eines vom Typ `int` und das andere vom Typ Zeiger auf eine Funktion mit Argument `int` und Rückgabewert `void`. Sie erwartet als Argument einen Integerwert und gibt keinen Wert (`void`) zurück. Eine identische, aber übersichtlichere Deklaration läßt sich durch Einführen eines eigenen Typs `SIGNAL_HANDLER` erreichen.

```
typedef void (*SIGNAL_HANDLER)(int);
SIGNAL_HANDLER signal(int, SIGNAL_HANDLER);
```

24 RCS

Programmentwicklung und Dokumentationserstellung erfolgt in der Regel schrittweise über einen längeren Zeitraum hinweg, wobei trotz ständiger Weiterentwicklung alte Versionen jederzeit verfügbar sein sollen. In großen Projekten ist dies eine komplexe Aufgabe, insbesondere wenn mehrere Personen an der Erstellung und Weiterentwicklung beteiligt sind.

Das RCS (**Revision Control System**) dient ganz allgemein zur Verwaltung beliebiger Dateien (insbesondere Quellcode von Programmen), die einer **Entwicklungsgeschichte** unterliegen. Eine übliche Entwicklungsgeschichte eines Programmquelltextes ist z.B.:

- Erste Version 1.0
- Wiederholte Erweiterungen und Fehlerkorrekturen
- Erste (Test-)Version 1.18
- Wiederholte Erweiterungen und Fehlerkorrekturen
- Erste ausgelieferte (Produktions-)Version 1.25
- ...
- Letzte ausgelieferte (Produktions-)Version 3.14
- ...

Die **Nullversion** 1.0 ist zwar lauffähig, aber noch unvollständig und aller Wahrscheinlichkeit nach stark fehlerbehaftet. Die erste ausgelieferte Version 1.25 entspricht noch nicht den Erwartungen der Anwender. Auch die letztendlich ausgelieferte Version 3.14 wird während ihrer Anwendung wahrscheinlich Fehler zeigen oder aufgrund von Anwenderwünschen erweitert werden müssen.

24.1 Übersicht

Zur Koordination und Verwaltung solcher Entwicklungsgeschichten bietet RCS folgende Fähigkeiten:

- **Speichern und Wiederherstellen vielfacher Versionen** einer Datei. RCS sichert alle alten Versionen auf platzsparende Weise. Alte Versionen können über **Kennungen** der Form
 - Versionsnummer
 - Symbolischer Name
 - Datum
 - Autor
 - Status

wiederhergestellt werden.

- Erhaltung der kompletten **Historie von Veränderungen**. RCS protokolliert alle Veränderungen automatisch mit. Außer dem Text jeder Version speichert RCS
 - Autor
 - Zeit und Datum des letzten Eincheckens
 - Eine die Veränderung zusammenfassende Notiz
- **Lösen von Konflikten** beim Zugriff. Wenn zwei Programmierer die selbe Version einer Datei bearbeiten wollen, macht RCS darauf aufmerksam und verhindert, daß eine Bearbeitung die andere versehentlich zerstört. Im Extremfall werden sogar gleichzeitig Änderungen an verschiedenen Stellen einer Datei ermöglicht.
- Erstellen eines **Baums aller Versionen**. RCS kann für jedes Modul getrennte Entwicklungslinien unterhalten. Es speichert eine Baumstruktur, die die Abstammungen und Beziehungen der Versionen darstellt.
- **Verschmelzen von Versionen**. Zwei unterschiedliche Entwicklungslinien eines Moduls können verschmolzen werden. Falls die Veränderungen der zu verschmelzenden Versionen die gleichen Stellen eines Codes betreffen, macht RCS den Anwender darauf aufmerksam.
- **Überblick** über ausgelieferte Versionen und Konfigurierungen. Es ist möglich, Versionen symbolische Namen zu geben und sie als
 - Ausgeliefert
 - Stabil
 - Experimentellzu kennzeichnen.
- **Identifizieren jeder Version** durch Name, Versionsnummer, Zeitpunkt der Erstellung, Autor, etc. Die Identifikation ist wie ein Stempel, der an einer geeigneten Stelle des Textes einer Version eingefügt werden kann.
- **Minimierung sekundärer Speicherung**. RCS benötigt wenig zusätzlichen Speicherplatz für die einzelnen Versionen (nur die Unterschiede). Falls dazwischen liegende Versionen gelöscht werden, werden die entsprechenden Deltas entsprechend komprimiert.

RCS ist *nicht* dafür gedacht, jede kleine Änderung einzeln zu protokollieren, sondern einzelne Entwicklungsschritte aufzuzeichnen. Dies ergibt sich schon aus dem Schema der Quellcodeverwaltung unter RCS:

1. Eine erste **Nullversion** des Quellcodes (und sei es in Form von leeren Dateien) eines Programms liegt vor, z.B. `filename.c`.
2. Diese Quellen werden in einem (allen Entwicklern zugänglichen) **Projekt-Verzeichnis** abgelegt.

3. In diesem Verzeichnis wird ein **RCS-Unterverzeichnis** gleichen Namens (RCS groß) angelegt.
4. Die am Projekt beteiligten Entwickler erzeugen **symbolische Links** von ihrem jeweiligen lokalen Projekt-Arbeitsverzeichnis auf das gemeinsame Projekt-RCS-Verzeichnis.
5. Um eine Quelldatei `filename.c` an RCS zu übergeben, verwendet man den Check-in Befehl **ci filename.c**. RCS erzeugt eine Versionsdatei `filename.c,v`, legt `filename.c` darin als Version 1.1 ab, und löscht `filename.c`. Außerdem wird nach einer Beschreibung gefragt, die eine Übersicht über die Datei darstellt.
6. Zum Bearbeiten einer Quelldatei muß diese mit dem Check-out Befehl **co [-l] filename.c** aus ihrer Versionsdatei **ausgecheckt** werden. D.h. ihre aktuelle Version wird zur Verfügung gestellt und die Versionsdatei für die anderen Entwickler **gesperrt**. Eine Quelldatei kann auch so ausgecheckt werden, daß sie nicht verändert werden darf. Dies funktioniert sogar, während die Datei gesperrt ist.
7. Nach der Bearbeitung der Quelldatei wird sie wieder in ihre Versionsdatei **eingchecked**. Dies kann nur derjenige Entwickler, der sie auch ausgecheckt hat. Ihre Versionsnummer wird erhöht, das Datum und der Bearbeiter gemerkt und ein **Kommentar** für den Grund der Änderung abgefragt, der dieser Version als Information hinzugefügt wird. Anschließend ist die Versionsdatei wieder für andere Entwickler freigegeben.
8. Über ihre **Versionsnummer** kann jede der in einer Versionsdatei vorhandenen Versionen einer Quelltextdatei wieder erhalten werden.

24.2 Beschreibung

- Die Form der von RCS verwendeten Versionsnummer ist (je nachdem, ob eine **lineare** oder **hierarchische** Folge von Versionen verwaltet wird):

```
Release.Level                                ODER
Release.Level.Branch.Sequence
```

- Die **erste Version** einer Quelldatei bekommt die Versionsnummer **1.1**. Jedes Aus- und wieder Einchecken führt zu einer Erhöhung des **Level** um 1. Wahlweise kann auch (bei größeren Änderungen oder dem Abschluß eines Entwicklungsschrittes) der **Release** erhöht und der Level wieder auf 1 zurückgesetzt werden.
- Die beiden letzten Stellen **Branch** und **Sequence** dienen als Reserve und können z.B. verwendet werden, wenn an einer eigentlich abgeschlossenen älteren Version kleine Änderungen vorzunehmen sind, ohne diese bis zur neuesten Version durchzuziehen.
- RCS kennt folgende Kommandos:

Kommando	Bedeutung
ci	Speichert neue Version in einer RCS-Datei
co	Entnimmt eine Version aus einer RCS-Datei
ident	Sucht nach dem Text: <i>\$/d...\$</i> und gibt ihn aus
rscs	Stellt neue RCS-Dateien her oder ändert ihre Attribute
rscs clean	Löscht ausgecheckte Dateien, die nicht verändert wurden
rscs diff	Vergleicht RCS-Versionen
rscs freeze	Vergibt einen symbolischen Versionsnamen für RCS-Dateien, die eine gültige Konfiguration bilden
rscs merge	Vereinigt die Veränderungen zwischen zwei Versionen
rlog	Gibt Informationen über RCS-Dateien aus

- Durch **Schlüsselwörter** (begrenzt von zwei Dollarzeichen) können in den Quelldateien **variable Texte** eingetragen werden, die beim Check-out durch aktuelle Werte ersetzt werden. Dadurch können wichtige Informationen zur Version (Quellname, Versionsnummer, Bearbeiter, Datum, ...) *automatisch* in den Quelltext eingetragen werden.

Keyword	Bedeutung
<i>\$Source\$</i>	Name der RCS-Datei mit Pfad
<i>\$RCSfile\$</i>	Name der RCS-Datei ohne Pfad
<i>\$Revision\$</i>	Versionsnummer
<i>\$Date\$</i>	Datum und Zeit (GMT) des Eincheckens
<i>\$Author\$</i>	Login-Name des eincheckenden Anwenders
<i>\$State\$</i>	Zustand einer Quelldatei
<i>\$Locker\$</i>	Login-Name des Anwenders, der die Version sperrt
<i>\$Header\$</i>	Standard-Kopf = alle o.g. Keywords außer <i>\$RCSfile\$</i> (Source Revision Date Author State Locker)
<i>\$Id\$</i>	Standard-Kopf = alle o.g. Keywords außer <i>\$Source\$</i> (RCSfile Revision Date Author State Locker)
<i>\$Log\$</i>	Check-in Kommentar

- *\$State\$* wird durch die Option *-s* von **rscs** oder **ci** definiert. Es stehen drei verschiedene Zustände zur Verfügung, nämlich **Exp** (experimental), **Stab** (stable) und **Rel** (released).
- Die beiden Schlüsselwörter *\$Id\$* und *\$Log\$* müssen in jeder Quelldatei am Dateianfang bzw. am Dateiende vorhanden sein (als Kommentar).

24.3 Beispiel

Zum Abschluß dieser Beschreibung von RCS werden die wichtigsten Kommandos in einem Musterbeispiel zusammenhängend verwendet. Im Prinzip können bei jedem Kommando mehrere Dateien angegeben werden, die Aufrufe verwenden der Einfachheit halber eine Datei.

- Bei allen RCS-Kommandos kann der Schalter *-rN.M* angegeben werden, wenn nicht die letzte Version, sondern die Version *N.M* angesprochen werden soll.
- Zu verwalten sei die Quelldatei `neuro.c` eines Programms namens *neuro*. Erster Schritt ist das Anlegen des **zentralen Projektverzeichnis** `neuro` und eines RCS-Verzeichnisses darin (Gesamtpfad `/src/neuro/RCS`):

```
cd /src
mkdir neuro
cd neuro
mkdir RCS
```

- Dann ist im **persönlichen Projekt-Verzeichnis** ein symbolischer Link auf dieses zentrale RCS-Verzeichins zu erstellen:

```
cd ~user/src/neuro
ln -s /src/neuro/RCS .
```

- Durch den Check-in Befehl wird aus `neuro.c` die RCS-Datei `neuro.c,v` mit der Versionsnummer 1.1 erzeugt, `neuro.c` gelöscht und eine Beschreibung des Programms abgefragt.

```
ci neuro.c
```

- Um aus der letzten Version von `neuro.c,v` wieder die Arbeitsdatei `neuro.c` herzustellen, verwendet man den Check-out Befehl:

```
co neuro.c
```

- Will man die Datei `neuro.c` ändern, muß beim Auschecken die **Sperre (Lock)** mit der Option `-l` eingeschalten werden:

```
co -l neuro.c
```

- Die Unterschiede zwischen der ausgecheckten Datei und der letzten eingeeckten Version können angezeigt werden durch:

```
rcsdiff neuro.c
```

- Nach dem Abschluß der Änderungen wird `neuro.c` wieder in RCS eingeecheckt:

```
ci neuro.c
```

- Die Versionsnummer hat sich dadurch auf 1.2 erhöht. Der Zeitpunkt, der Bearbeiter und die Änderungen sind in der Versionsdatei festgehalten, ein Kommentar zur Änderung wird interaktiv abgefragt (*mit einem einzelnen Punkt abzuschließen*), die Datei `neuro.c` wird anschließend gelöscht.

- Falls **ci** die Meldung

```
ci error: no lock set by USERNAME
```

ausgibt, wurde versucht eine Datei einzuchecken, obwohl sie beim Auschecken nicht gesperrt worden war. *In diesem Falle nicht mit einer Sperre auschecken, da dies die neuen Änderungen überschreiben würde.* Stattdessen, zunächst

```
rcs -l neuro.c
```

anwenden, um die letzte Version zu locken. Allerdings funktioniert dies nur, wenn nicht schon jemand anderes zuvorgekommen ist (vorher mit **rcsdiff** vergleichen, ob jemand anderes Änderungen gemacht hat).

- Ein neuer Release 2.1 (der Level wird immer auf 1 zurückgesetzt) kann erzwungen werden (sofern die jüngste Version eine kleinere Versionsnummer hat) durch:

```
ci -r2 neuro.c                ODER  
ci -r2.1 neuro.c
```

- Ein Branch kann angelegt werden, z.B. zu der Version 1.2 (die neue Version erhält die Nummer 1.2.1.1), durch:

```
ci -r1.2.1 neuro.c
```

- Soll das Programm `neuro` erzeugt werden, ist die Quelldatei aus RCS zu extrahieren und anschließend zu übersetzen:

```
co neuro.c  
cc neuro.c -o neuro
```

25 make und Makefiles

Mit *make* (*nmake* = „new“ *make* unter MSDOS¹) und einer zugehörigen **Makedatei** (Standard ist *makefile* oder *Makefile*) kann der zur Erstellung von Programmen notwendige Compilier- und Linkablauf automatisiert werden. Insbesondere kann damit erreicht werden, daß nach Änderungen nur die geänderten und alle davon abhängigen Programmteile neu übersetzt werden. Der Verwaltungsaufwand sowie die Compilations- und Linkzeiten bleiben dadurch minimal. Die in der folgenden Beschreibung angegebenen Beispiele gelten für den MSC-Compiler unter MSDOS. Unter UNIX bestehen keine prinzipiellen Unterschiede, aber die Programmnamen und Suffixe lauten dort anders.

25.1 Beschreibung

- Eine **Makedatei** besteht aus
 - Abhängigkeitsregeln
 - Suffixregeln
 - Aktionen
 - Makrodefinitionen
 - Kommentaren
 - Leerzeilen
- Die **Abhängigkeitsregeln** beschreiben, welche (Ziel)Dateien von welchen (Quell)Dateien abhängen. Auf eine Abhängigkeitsregel folgen, *jeweils mit einem Tabulator eingerückt* (daran werden sie erkannt), beliebig viele Aktionen (bis zur nächsten Abhängigkeitsregel oder Leerzeile).

```
Ziel: Quelle ...
    Aktion      # Tabulator davor!
    ...
```

- Eine Abhängigkeitsregel besagt, daß die (Ziel)Dateien jüngeren Datums sein müssen als die (Quell)Dateien. Verglichen wird jeweils das Datum der letzten Änderung. Ist eine der Quelldateien jünger, müssen die Zieldateien neu erzeugt werden. Es werden dann alle auf eine Abhängigkeitsregel folgenden **Aktionen** (Kommandos) ausgeführt. Diese bewirken die Neuerstellung des Ziels aus den Quellen auf eine geeignete Weise (müssen das aber nicht unbedingt).

```
xyz.exe: xyz.c xyz.h
    cl xyz.c
```

Im diesem Beispiel hängt das Programm (die Datei) *xyz.exe* von den Dateien *xyz.c* und *xyz.h* ab. Ist eine der beiden Dateien *xyz.c* oder *xyz.h* neueren Datums als *xyz.exe*, so wird *xyz.c* mit *cl* übersetzt und das Programm *xyz.exe* dadurch neu erzeugt (*Achtung: vor cl muß ein Tabulator stehen*).

¹Von Microsoft wurde zunächst zusammen mit dem MSC-Compiler ein *make* ausgeliefert, das nicht die üblichen Eigenschaften besaß. Später wurde ein dem Standard besser entsprechendes *make* mitgeliefert, das (leider) den Namen *nmake* bekam, um das alte *make* gegebenenfalls noch parallel verwenden zu können.

- Oft existiert eine **allgemeine Regel (Suffixregel)**, nach der Dateien eines bestimmten Typs von Dateien eines anderen Typs abhängen. Der Typ einer Datei wird dabei durch ihren Suffix festgelegt (.c, .obj, .exe, ...). Beispielsweise werden normalerweise alle C-Quellcode-Dateien mit dem gleichen Kommando in eine Objektdatei übersetzt. Die Suffixregel zum Erstellen von Objektdateien aus C-Quellcode-Dateien kann z.B. folgendermaßen lauten:

```
.c.obj:
    cl -AL -G2 -Zi -Od $<
```

Sie besagt, daß Objektdateien, für die keine explizite Abhängigkeitsregel existiert, mit dem angegebenen *cl*-Kommando aus den gleichnamigen C-Quellcode-Dateien erzeugt werden. Das Makro *\$<* wird dabei durch den jeweiligen Namen der C-Quellcode-Datei ersetzt.

- Da jede Quell(Datei) in einer Abhängigkeitsregel selbst wieder Ziel(Datei) einer anderen Abhängigkeitsregel sein kann, wird durch ein Makefile ein **Abhängigkeitsbaum** definiert. *make* baut diesen Baum auf und arbeitet ihn von den Blättern zur Wurzel hin ab.
- Folgende **Standardziele** müssen in jedem Makefile vorhanden sein und die beschriebenen Aktionen auslösen:

Ziel	Bedeutung
all	Erstes Ziel zum Erzeugen aller Programme und Dateien
clean	Löschen aller Zwischen- und Enddateien
new	Neuerstellen aller Programm und Dateien (<i>clean</i> + <i>all</i>)
depend	Generieren und Eintragen der Abhängigkeiten der C-Quellcode-dateien von den Headerdateien im Makefile
install	Installation der erzeugten Programme und Dateien
backup	Sichern der seit der letzten Sicherung geänderten Quelldateien
save	Sichern aller Quelldateien
test	Testen aller Programme

- Innerhalb eines Makefiles können beliebige **Makros** definiert und verwendet werden (analog zu C). Sie dienen vor allem zur Erhöhung der Übersichtlichkeit und Portierbarkeit. Die Definition eines Makros erfolgt durch

```
MACRO=beliebiger Text
```

Der Text bis zum Zeilenende wird dabei dem Makro *MACRO* zugewiesen (Leerzeichen vor dem Zeichen = werden ignoriert, solche danach und vor dem Zeilenende nicht). Die Verwendung eines Makros erfolgt durch Einklammern des Namens und Voranstellen eines Dollarzeichens (einbuchstabile Makros müssen nicht unbedingt eingeklammert werden):

```
$(MACRO)
```

Anstelle des Ausdrucks *\$(MACRO)* wird dann der ihm zugewiesene Text eingesetzt. Im folgenden Beispiel werden die Makros *CC*, *CFLAGS*, *MAIN* und *OBJS* definiert und anschließend in einer Abhängigkeitsregel und den zugehörigen Aktionen verwendet.

```

CC      = cl
CFLAGS = -AL -G2 -Zi -Od

MAIN = main.exe
OBJS = main.obj a.obj b.obj c.obj

$(MAIN): $(OBJS)
        $(CC) $(CFLAGS) $(OBJS)

```

- Folgende **Standardmakros** müssen in Makefiles anstelle der echten Betriebssystem-Kommandos verwendet werden, um sie portabel zu halten:

Makro	MSDOS	UNIX	Bedeutung
CC	<i>cl</i>	<i>gcc</i>	C-Compiler
LD	<i>link</i>	<i>ld</i>	Linker
AR	<i>lib</i>	<i>ar</i>	Bibliotheksverwalter
AS	<i>masm</i>	<i>as</i>	Assembler
LEX	<i>lex</i>	<i>flex</i>	Scannergenerator
YACC	<i>yacc</i>	<i>bison</i>	Parsergenerator
MAKE	<i>nmake</i>	<i>make</i>	<i>make</i> selbst (rekursiver Aufruf)
CFLAGS			Flags für C-Compiler
LDFLAGS			Flags für Linker
ARFLAGS			Flags für Bibliotheksverwalter
ASFLAGS			Flags für Assembler
LFLAGS			Flags für Scannergenerator
YFLAGS			Flags für Parsergenerator
MAKEFLAGS			Flags für <i>make</i> (rekursiver Aufruf)
RCSFLAGS			Flags für <i>rcs</i> allgemein
RM	<i>del</i>	<i>rm</i>	Löschbefehl
ECHO	<i>echo</i>	<i>echo</i>	Ausgabebefehl
EXE	<i>.exe</i>	–	Suffix ausführbarer Programme
OBJ	<i>.obj</i>	<i>.o</i>	Suffix von Objektdateien
LIB	<i>.lib</i>	<i>.a</i>	Suffix von Bibliotheken
HDR			Headerdateien
SRC			C-Quellcode-Dateien
INC			Includeverzeichnisse
OBJS			Objektdateien
LIBS			Bibliotheken
INST			Installationsverzeichnis
LDFILE	<i>*.lnk</i>	–	Linkdatei

- Weiterhin gibt es **vier spezielle Makros**, die jeweils automatisch in Abhängigkeit von der aktuellen Regel definiert sind. Diese Makros sind vor allem in Suffixregeln nützlich.

Makro	Bedeutung	Wo verwendbar
<code>\$\$</code>	Der Name des Ziels	Beide Regelarten
<code>\$\$?</code>	Alle Quellen, die jünger als das Ziel sind	Spezielle Regel
<code>\$\$<</code>	Der Name der Quelle, die die Aktion auslöste	Suffixregel
<code>\$\$*</code>	Analog, aber ohne Suffix (<code>.c</code> fällt weg)	Suffixregel

Folgendes Beispiel ist eine Suffixregel für die Übersetzung von C-Quellcode-Dateien in Objektdateien, ihre Aufnahme in eine Bibliothek, und das Löschen der Objektdateien angegeben.

```
.c.lib:
$(CC) -c $<
$(AR) $@ $*$$(OBJ)
$(RM) $*$$(OBJ)
```

- **Kommentare** können an beliebiger Stelle durch '#' eingeleitet werden, der danach folgende Text bis zum Zeilenende wird von *make* ignoriert.

```
# Dies ist ein Kommentar
```

- **Leerzeilen** sind überall in einem Makefile erlaubt (außer zwischen Abhängigkeitsregel und den zugehörigen Aktionen). Sie sind zwischen den einzelnen Regeln sowie zur Einteilung der Makrodefinitionen in Gruppen einzufügen.

```
$(CC) a.c      # Ein weiterer Kommentar
```

- Durch ein **Backslash** '\' *direkt vor* dem Zeilenende kann eine logische Zeile auf der nächsten physikalische Zeile fortgesetzt werden.

```
OBJS = a.obj b.obj c.obj \
      d.obj e.obj f.obj
```

25.2 Ablauf

- Die nach einer Abhängigkeitsregel aufgelisteten Aktionen werden Zeile für Zeile an den **Standardkommandoprozessor** (*command.com* unter MSDOS bzw. *sh* unter UNIX) weitergegeben. Da für jede Zeile ein eigener Kommandoprozessor mit eigenem Environment aufgerufen wird, kann im Umgebungsbereich abzulegende Information nicht von einem Kommando an das nächste weitergegeben werden (*set*-Befehle werden wieder vergessen).
- Während unter UNIX Kommandozeilen (fast) beliebig lang sein können, dürfen sie unter MSDOS maximal 126 Zeichen lang sein. Daher muß bei längeren Kommandozeilen wie sie z.B. für das Linken notwendig sind, auf eine Ausweichkonstruktion mit einer externen Link-Datei zurückgegriffen werden (siehe folgendes Beispiel). Makros und Abhängigkeitsregeln werden dagegen von *make* direkt verarbeitet und dürfen auch unter MSDOS (fast) beliebig lang sein.
- Der **Aufruf von *make*** lautet:

```
make [-f file] [optionen] [makro-definitionen] [ziele]
```

Wird *make* ohne Argumente aufgerufen, sucht es eine Datei namens `makefile` oder `Makefile` im aktuellen Verzeichnis. Die erste spezielle Regel darin und alle davon abhängigen Regeln werden ausgeführt. Soll eine andere Makedatei verwendet werden, kann sie über die Option `-f` angegeben werden. Sollen andere Ziele erzeugt werden, sind diese als Argument anzugeben. Zusätzliche Makrodefinitionen können in der Form `MACRO=text` angegeben werden. Sie überschreiben eine eventuell im Makefile vorkommende Definition des gleiche Makros.

- Als Optionen sind möglich

Option	Name	Bedeutung
-d	debug	Ablaufinformation und Datum ausgeben
-e	envvar	Umgebungsvariablen überschreiben Makrodefinitionen
-f file	file	Als Makefile file verwenden
-i	ignore	Bei Fehlern in Aktionen nicht abbrechen
-n	nothing	Aktionen ausgeben, aber nicht ausführen
-p	print	Vordefinierte Makros und Suffixregeln ausgeben
-s	silent	Aktionen bei Ausführung nicht ausgeben

- Ein Makelauf wird normalerweise abgebrochen, wenn eine Aktion einer Regel zu einem Fehler führt. Durch Voranstellen eines Minuszeichens '-' direkt vor der Aktion kann dies für eine einzelne Aktion verhindert werden (bei Angabe des Flags -i führt ein Fehler grundsätzlich nicht zum Abbruch des Makelaufs):

```
.c.lib:
$(CC) -c $<
$(LIB) $@ $*$$(OBJ)
-$(RM) $*$$(OBJ)
```

- Kann ein Ziel aus irgendeinem Grund nicht erzeugt werden (eine Aktion führt zu einem Fehler), so wird seine veraltete Form (sofern vorhanden) gelöscht. Um dies für ausgewählte Ziele zu verhindern, können nach dem Befehl **.PRECIOUS** alle Dateien angegeben werden, die nicht gelöscht werden dürfen.

```
.PRECIOUS mylib.lib ...
```

- In einem Makefile sollen aus Gründen der Portabilität und Übersichtlichkeit alle Kommandos und Dateinamen klein, sowie alle Makronamen groß geschrieben werden. Statt einen Kommandonamen mehrmals direkt zu verwenden, ist es aus dem gleichen Grund besser, ein Makro zu definieren und dieses zu verwenden.

25.3 Beispiel

Als Musterbeispiel für ein übersichtliches und portables Makefile sei folgendes — für ein Programm namens MARSIM erstelltes — besprochen.

```
#-----
# Modul: Makefile MARSIM
# Autor: AUTOR
# Datum: TT.MM.JJ
# Stand: TT.MM.JJ
# Zweck: .....
#-----
# Aenderungen:
#-----

# Programmnamen und Suffixe unter MSDOS # UNIX
CC = cl # cc
```

```

LD = link # ld
RM = del # rm
ECHO = echo # echo
EXE = .exe # leer!
OBJ = .obj # .o

# Flags und Includeverzeichnisse fuer Compiler
CFLAGS = -AL -G2 -Od -Zi -W3 #large model, 286-code, no optimize, codeview-info
#CFLAGS = -AL -W4 #large model, warn all
#CFLAGS = -AL -G2 -Gs -Ox #large model, 286-code, no stack check, optimize
INC = -I. -Itoolbox -Iarrhandl

# Flags und Bibliotheken fuer Linker
LDFLAGS = /NOI /SE:0x90
#LDFLAGS = /NOI /SE:0x90 /CO # Codeview-Info
LIBS = toolbox\mar_ct_1.lib arrhandl\l_ary.lib c:\c600\lib\llibce.lib

# Hauptprogramm + seine Header-, Quellcode- und Objektdateien
MAIN = marsim$(EXE)
HDR = *.h
SRC = mfunk.c mcopyunt.c mgestalt.c mhost.c marsim.c \
      mladen.c mperload.c msprache.c mtextdat.c \
      mbearbei.c mglobals.c mupdate.c mauswahl.c marfo.c
OBJ1 = mfunk$(OBJ) mcopyunt$(OBJ) mgestalt$(OBJ) mhost$(OBJ) marsim$(OBJ)
OBJ2 = mladen$(OBJ) mperload$(OBJ) msprache$(OBJ) mtextdat$(OBJ)
OBJ3 = mbearbei$(OBJ) mglobals$(OBJ) mupdate$(OBJ) mauswahl$(OBJ) marfo$(OBJ)
OBJS = $(OBJ1) $(OBJ2) $(OBJ3)

# Linkdatei und Installationsverzeichnis unter MSDOS
LDFILE = marsim.lnk
INSTDIR = c:\bin

# Zum Sichern unter MSDOS
BACKUP = XCOPY
BFLAGS = /m
DEST = b:\

#-----
# Allgemeine Uebersetzungsregel fuer C-Programme
.c$(OBJ):
    $(CC) $(CFLAGS) $(INC) -c $*.c

#-----
# Erste Regel, wird beim Aufruf von make ohne Argumente ausgefuehrt
all: $(MAIN)

# Loescht alle Zwischen- und Enddateien (fuer komplette Neuerstellung)
clean:
    $(RM) $(MAIN)
    $(RM) *$(OBJ)

# Programm MARSIM vollstaendig neu erstellen
new: clean all

# Programm installieren
install:
    $(CP) $(MAIN) $(INSTDIR)

```

```

# Geaenderte Quellcode-Dateien auf Diskette sichern
backup:
    $(BACKUP) makefile $(DEST) $(BFLAGS)
    $(BACKUP) *.c      $(DEST) $(BFLAGS)
    $(BACKUP) *.h      $(DEST) $(BFLAGS)
    $(BACKUP) *.asc    $(DEST) $(BFLAGS)
    $(BACKUP) *.mar    $(DEST) $(BFLAGS)
    $(BACKUP) *.ury    $(DEST) $(BFLAGS)

# Alle Quellcode-Dateien auf Diskette sichern
save:
    $(BACKUP) makefile $(DEST)
    $(BACKUP) *.c      $(DEST)
    $(BACKUP) *.h      $(DEST)
    $(BACKUP) *.asc    $(DEST)
    $(BACKUP) *.mar    $(DEST)
    $(BACKUP) *.ury    $(DEST)

# Programm MARSIM erstellen (Trick mit ECHO wg. beschraenkter Zeilenlaenge)
$(MAIN): $(OBJS) $(HDR)
# $(LD) $(OBJS), $@, $(LIBS), $(LDFLAGS); ### wg. MSDOS Kommando <= 126 Zeichen
$(ECHO) $(OBJ1) + > $(LDFILE)
$(ECHO) $(OBJ2) + >> $(LDFILE)
$(ECHO) $(OBJ3) >> $(LDFILE)
$(ECHO) $@ >> $(LDFILE)
$(ECHO) nul >> $(LDFILE)
$(ECHO) $(LIBS) >> $(LDFILE)
$(ECHO) $(LDFLAGS) >> $(LDFILE)
$(ECHO) nul >> $(LDFILE)
$(LD) @$ $(LDFILE)
$(RM) $(LDFILE)

```

- In diesem Makefile werden zunächst die Namen der verwendeten Kommandos und Suffixe als Makros definiert. Soll das gleiche Makefile unter UNIX ablaufen, sind die danebenstehenden Namen als Werte einzusetzen.
- Es folgen Makros für die C-Compilerflags und Verzeichnisse, in denen sich die zum Übersetzen notwendigen Include-Dateien befinden. Analog werden die Linkerflags und die Namen der zum Binden benötigten Bibliotheken als Makros definiert. Alternative Belegungen der Compiler- und Linkerflags für Test- und Debugzwecke sind als Kommentar angegeben, sie können so jederzeit aktiviert werden.
- Die nächsten Makros werden zum Festlegen der Abhängigkeiten verwendet: `MAIN` ist der Name des zu erzeugenden Programms `MARSIM`, `HDR` enthält die Namen aller darin verwendeten Headerdateien. Wie man an diesem Makro sieht, können Dateien auch über die Wildcards `*` und `?` definiert werden. `SRC` enthält die Namen aller Quellcode-Dateien, aus denen `MARSIM` besteht. Durch zwei Backslashes vor dem Zeilenende wird dabei die logische Zeile auf drei physikalische Zeilen verteilt. Die Definition des Makros `OBJ` für die Objektdateinamen wird absichtlich über drei Untermakros `OBJ1-3` durchgeführt. Jedes der Untermakros ist deutlich kürzer als 126 Zeichen, d.h. unter MSDOS können diese Makros auch in einer Aktion verwendet werden.

- Die weiteren Makros sind MSDOS-spezifisch, sie legen die Namen der Linkdatei `LDFILE` und die Namen der für die Installation bzw. Sicherung notwendigen Programme und Pfade fest.
- Danach folgt die Suffixregel zum Übersetzen von C-Quellcode-Dateien in Objektdateien. Ihre einzige Aktion ist allgemeingültig fast ausschließlich aus Makros zusammengesetzt. Änderungen sind daher nur an einer Stelle, nämlich den Makros, vorzunehmen. Diese Suffixregel wird auf jede Objektdatei angewendet, die aufgrund einer anderen Regel benötigt wird, für die aber keine spezielle Regel existiert.
- Die erste spezielle Regel mit dem Ziel `all` legt fest, daß das Programm `marsim.exe` zu erzeugen ist, da die Datei `all` (hoffentlich) nicht existiert. Aktionen werden in dieser Regel nicht durchgeführt, insbesondere wird auch keine Datei `all` erzeugt. D.h. beim nächsten Aufruf von `make` wird wieder `marsim.exe` erzeugt, sofern seit dem letzten Makelauf eine Änderung an einer seiner Quelldateien stattfand. Diese Regel wird standardmäßig ausgeführt, wenn `make` im Verzeichnis des Makefiles ohne Argumente aufgerufen wird.
- Die nächste Regel mit dem Ziel `clean` hat keine Abhängigkeiten. Da eine Datei `clean` nicht existiert, werden ihre Aktionen ausgeführt und damit alle Zwischendateien gelöscht. Soll z.B. das Programm MARSIM mit den Compilerflags `-AS ...` übersetzt werden, so ist zunächst **make clean** und dann `make CFLAGS="-AS ..."` aufzurufen.
- Die Regel `new` ruft zuerst `clean` und dann `all` auf, d.h. das Programm wird vollständig neu erzeugt.
- Die Regel `install` verschiebt nach dem Abschluß der Programmentwicklung die erzeugten Programme und Dateien in Verzeichnisse, die im Suchpfad liegen. Dies ist besonders unter UNIX wichtig, um allen Anwendern den Aufruf zu ermöglichen.
- Die Regel `backup` sichert seit der letzten Sicherung veränderte Quelldateien auf Diskette, die Regel `save` sichert sämtliche Quelldateien auf Diskette.
- Die letzte Regel ist für die Erzeugung des Programms MARSIM verantwortlich. MARSIM muß dann neu erzeugt werden, wenn sich eine der Header- oder Objektdateien geändert hat. Da es für keine der Objektdateien eine spezielle Regel gibt, aber eine Suffixregel für sie existiert, wird die Suffixregel auf sie angewendet. D.h. jede Objektdatei muß neuer als ihre zugehörige C-Quellcode-Datei sein. Ist sie das nicht, wird sie durch die Aktion der Suffixregel erzeugt. Für die Headerdateien gibt es keine Abhängigkeitsregeln, sie werden aus keiner anderen Datei generiert, d.h., es genügt, daß sie vorhanden sind.
- Als Aktion der letzten Regel genügt im Prinzip ein einziger Linkbefehl zum Erzeugen von MARSIM. Da dieser Befehl nach dem Einsetzen der Makrotexte aber länger als 126 Zeichen ist, wird das Kommando über **ECHO**-Befehle in der *Linkdatei* `marsim.lnk` abgelegt und dem Linker dieses File übergeben (Kennzeichen ist ein `@` vor dem Linkdateinamen). Abschließend wird die Linkdatei wieder entfernt.

26 makedepend

Ein Problem bei der C-Programmierung und Übersetzung der Quelldateien mittels *make* sind die **Abhängigkeiten** der Quellcode-Dateien von den Headerdateien. Bei Änderung einer Headerdatei müssen alle Quellcode-Dateien, die sie includen, neu übersetzt werden. Häufig werden allerdings aufgrund der Vielzahl von Abhängigkeiten zwischen *.c*- und *.h*-Dateien die sie beschreibenden Regeln nicht in ein Makefile eingetragen.

Bei Verwendung des Programms *makedepend* ist das auch gar nicht notwendig. Es sucht aus allen Quellcode-Dateien eines Verzeichnisses die darin enthaltenen *include*-Anweisungen (nur "...", nicht <...>) heraus, generiert die passenden Abhängigkeitsregeln und trägt sie zusätzlich im Makefile ein.

Statt durch ein eigenes Programm ist *makedepend* (ganz im Sinne von UNIX) durch eine über Pipes verbundene Kombination der UNIX-Tools *sed*, *grep*, *sort* und *awk* realisiert. Das zugehörige Skript läßt sich direkt im Makefile unter dem Ziel *depend* eintragen, es ist im folgenden Beispiel-Makefile enthalten. (Achtung: in den eckigen Klammern steht jeweils Blank + Tabulator, vor den eingerückten Anweisungen nach *depend*: steht ein Tabulator):

```
# Programmnamen
GREP=grep
SORT=sort
SED=sed
AWK=gawk
MV=mv
RM=rm

# Makefile-Name
MAKEFILE=makefile

# Quelldateien
SRC=version.c \
  cpp.c declare.c error.c experror.c stmerror.c expr1.c expr2.c expr3.c \
  gram.y init.c keyword.c main.c memalloc.c nametbl.c \
  optabop.c strconst.c syntab.c optim.c initial.c \
  toks.l typesys.c stm.c constru.c exprouc.c hash.c declout.c stmout.c \
  output.c inline.c glbinit.c

depend: $(SRC)
# hier beginnt das makedepend-Skript !!!
@$(GREP) '^# END MAKE depend' $(MAKEFILE) > /dev/null
$(SED) -n '1,/^# BEGIN MAKE depend/ p' $(MAKEFILE) > depend
$(GREP) '^#[ \t]*include[ \t]*[ \t]*"' /dev/null $(SRC) | \
$(SED) 's/:[ \t]*#[ \t]*include[ \t]*[ \t]*"/:;/s/"*//;s/\. [ylc]:/\.o:/' | \
$(SORT) -u | \
$(AWK) -F: '\
BEGIN {\
  n = 0;\
  obj = "";\
  printf "\n";\
}\
$$1 != obj {\
  if (n >=1 ) printf "\n";\
  printf "%s:", $$1;\
  obj = $$1;\
  n = 0;\
}
```

```

        fill = 15 - length($$1);\
    }\
    {\
        while (--fill >= 0) printf " ";\
        if (n >= 3)\
        {\
            printf "\\n\t\t%s", $$2;\
            n = 1;\
        }\
        else\
        {\
            printf "%s", $$2;\
            ++n;\
        }\
        fill = 16 - length($$2);\
    }\
END    {\
    printf "\n\n";\
}' | pr -t -i8 >> depend
$(SED) -n '/^# END MAKE depend/, $$ p' $(MAKEFILE) >> depend
$(MV) $(MAKEFILE) $(MAKEFILE)-
$(MV) depend $(MAKEFILE)
# hier endet das makedepend-Skript !!!

# BEGIN MAKE depend - do not delete this line
# END MAKE depend - do not delete this line

```

Zum Eintragen der Abhängigkeiten in ein Makefile ist das *makedepend*-Skript einschließlich der beiden Textzeilen `#BEGIN MAKE depend ...` und `#END MAKE depend ...` in das Makefile einzutragen und *make depend* aufzurufen. Danach sind zwischen den beiden Textzeilen die gewünschten Abhängigkeitsregeln zwischen Quellcode-Dateien und Headerdateien eingetragen.

Als Einschränkungen des durch das Skript realisierten *makedepend* gilt: Abhängigkeiten von *mehrfach verschachtelten* Headerdateien werden nicht erkannt. D.h. liest eine Headerdatei wieder eine andere Headerdatei ein, so wird nur die Abhängigkeit der Quellcode-Datei von der ersten, aber nicht von der zweiten Headerdatei im Makefile eingetragen.

Sind unter MSDOS Versionen der vier obengenannten UNIX-Tools *Sed*, *Grep*, *Sort*, *Gawk* verfügbar, so ist *makedepend* auch dort realisierbar. Allerdings ist aufgrund der maximal erlaubten Kommandozeilenlänge von 126 Zeichen die Verwendung von Zwischendateien statt Pipes notwendig. Das folgende MSDOS Makefile entspricht dem vorherigen UNIX Makefile-Beispiel.

```

# Programmnamen
GREP=\mks\bin\grep
SORT=\mks\bin\sort
SED=\mks\bin\sed
AWK=\mks\bin\awk
MV=ren
RM=del

# Makefile-Name
MAKEFILE=makefile

# Quelldateien, aufgeteilt wegen max. Kommandozeilenlaenge <= 126

```

```

SRC1=version.c
SRC2=cpp.c declare.c error.c experror.c sterror.c expr1.c expr2.c expr3.c
SRC3=gram.y init.c keyword.c main.c memalloc.c nametbl.c
SRC4=optabop.c strconst.c sytab.c optim.c initial.c
SRC5=toks.l typesys.c stm.c constru.c exprout.c hash.c declout.c stmout.c
SRC6=output.c inline.c glbinit.c
SRC=$(SRC1) $(SRC2) $(SRC3) $(SRC4) $(SRC5) $(SRC6)

# hier beginnt das makedepend-Skript !!!
depend: $(SRC)
$(SED) -n '1,/^# BEGIN MAKE depend/ p' $(MAKEFILE) >depend
$(GREP) '^#[ ]*include[ ][' ]*" '$(SRC1) >xxx
$(GREP) '^#[ ]*include[ ][' ]*" '$(SRC2) >>xxx
$(GREP) '^#[ ]*include[ ][' ]*" '$(SRC3) >>xxx
$(GREP) '^#[ ]*include[ ][' ]*" '$(SRC4) >>xxx
$(GREP) '^#[ ]*include[ ][' ]*" '$(SRC5) >>xxx
$(GREP) '^#[ ]*include[ ][' ]*" '$(SRC6) >>xxx
$(SED) 's:[ ]*#[ ]*include[ ][' ]*" /:;/s/"./;/s/\.[ylc]:/.o:/' <xxx >yy
$(SORT) -u <yyy >zzz
$(AWK) -F: -f makedep.awk <zzz >>depend
$(SED) -n '/^# END MAKE depend/, $$ p' $(MAKEFILE) >> depend
$(RM) $(MAKEFILE).old
$(MV) $(MAKEFILE) $(MAKEFILE).old
$(MV) depend $(MAKEFILE)
$(RM) xxx
$(RM) yyy
$(RM) zzz

# hier endet das makedepend-Skript !!!

# BEGIN MAKE depend - do not delete this line
# END MAKE depend - do not delete this line

```

In `makedep.awk` muß folgende `awk`-Skript stehen:

```

BEGIN {
    n = 0;
    obj = "";
    printf "\n";
}
$1 != obj {
    if (n >= 1) printf "\n";
    printf "%s:", $1;
    obj = $1;
    n = 0;
    fill = 15 - length($1);
}
{
    while (--fill >= 0) printf " ";
    if (n >= 3)
    {
        printf "\\n\t\t%s", $2;
        n = 1;
    }
    else
    {
        printf "%s", $2;
        ++n;
    }
}

```

```
    fill = 16 - length($2);  
  }  
END {  
  printf "\n\n";  
}
```

27 texdoc

Das Programm *texdoc* unter UNIX unterstützt die Dokumentation von C-Quelltexten, C-Headerdateien und SQL-Quelltexten. Dazu sind in den Quellen innerhalb von Kommentaren bestimmte **Schlüsselwörter** und darauf folgend **L^AT_EX-Beschreibungstext** anzugeben. Dieser Beschreibungstext wird anhand der Schlüsselwörter erkannt und zusammen mit dem nachfolgenden Quellcode von L^AT_EX in ansprechender Form gesetzt. Es gibt 6 Schlüsselwörter:

- HEAD : Leitet den Kopf der Programmdokumentation ein.
- END : Beendet den Kopf der Programmdokumentation.
- GROUP : Leitet einen Abschnitt (Überschrift) ein.
- TEXT : Leitet einen Beschreibungstext ein, auf den *kein* Quellcode folgt.
- DESC : Leitet einen Beschreibungstext zu nachfolgenden Includes, Konstanten, Makros, Präprozessoranweisungen, Datentypdefinitionen, Variablendefinitionen oder einer nachfolgenden Funktion ein.
- KEY : Leitet eine Liste mit Suchbegriffen ein, die den Code beschreiben.

Zwischen Schlüsselwort und Doppelpunkt kann eine Zahl als **Levelnummer** angegeben werden (keine Zahl entspricht der Levelnummer 0). Beim Aufruf von *texdoc* können der minimale bzw. maximale Level angegeben werden, ab dem bzw. bis zu dem der Beschreibungstext in die Beschreibung aufgenommen werden soll. Man kann also zum Beispiel in Level 0 die allgemeine, für alle Entwickler notwendige Beschreibung angeben, und als Level 1 die nur für den Ersteller wichtige hinzufügen. Die Usage-Meldung von *texdoc* lautet:

```
usage: texdoc {OPTIONS} FILE...
```

```
Dokumentation zu C/SQL-Quellen FILE... ausdrucken. Optionen:
```

```
-l      Beschreibung mit LaTeX setzen, DVI-File ausgeben
-p      Beschreibung mit LaTeX setzen, DVI-File ausdrucken

-cd     Prozedurcode in Beschreibung uebernehmen
-mf     Makros mit Argumenten als Funktionen darstellen
-nd     #ifdef DEBUG ... #endif-Abschnitte unterdruecken
-nt     #ifdef TEST ... #endif-Abschnitte unterdruecken
-na     assert()-Aufrufe unterdruecken
-pg     Fuer jede Datei eine neue Seite beginnen
-toc    Inhaltsverzeichnis erzeugen
-fl     Alphabetisch sortierte Funktionenliste erzeugen
-ofl    Nur alphabetisch sortierte Funktionenliste erzeugen
-idx    Index erzeugen

-f <FILE>|<FUNC> Liste/Name auszugebender Funktionen [Std: Alle]
-b<N>   Dokumentation ab Level <N> extrahieren [Std: 0]
-<N>    Dokumentation bis Level <N> extrahieren [Std: 0]
-r<N>   Die letzten <N> Aenderungen mit ausgeben [ohne <N> alle]
-t<N>   Tabulatoren auf <N> Leerzeichen expandieren [Std: 4]
-nl     Zeilenvorschuebe in Kommentaren uebernehmen
-bc     Prozedurcode verschoenern
-s      Keine Fortschrittsmeldungen ausgeben
```

28 Sonstige Tools

Zur Programmentwicklung mit C unter UNIX oder MSDOS sind neben dem Compiler und Linker eine Reihe weiterer Programme von Nutzen: *(n)make*, *rcc*, *what*, *grep*, *sed*, *(g/n)awk*, *lex*, *yacc*, ... Die Tools *rcc* und *make* sind bereits beschrieben worden, zu den anderen Tools sind die Manualseiten oder Literatur verfügbar.

Neben diesen vom Betriebssystem zur Verfügung gestellten Tools sind noch einige Eigenentwicklungen vorhanden, die im folgenden kurz vorgestellt werden. Die Dokumentation und der ausführbare Code für diese Tools sind beim Autor erhältlich.

Die meisten der Programme geben beim Aufruf mit einer unbekanntenen Option (`-?` oder `-h`) eine Usage-Meldung aus.

- Der **interaktive Compileraufsatz** *mcc* wird wie der normale C-Compiler aufgerufen. Beispiel:

```
mcc -AL -Ox -G2 xyz.c
```

Er ruft seinerseits zur Übersetzung der angegebenen Quelltexte den C-Compiler mit den gleichen Argumenten auf. Treten beim Übersetzen Syntaxfehler auf, so wird die fehlerbehaftete Datei *zusammen mit* den eingefügten Fehlermeldungen in den Editor *vi* (oder über die Umgebungsvariable `EDITOR` einstellbar auch einen beliebigen anderen Editor) geladen und kann sofort verbessert werden. Fehlermeldungen werden darin durch

```
>>>>
```

eingeleitet. Die letzte Fehlermeldung ist durch

```
(last error)
```

gekennzeichnet. Im *vi* kann mit `n/N` zum nächsten/vorherigen Fehler gesprungen werden. Nach dem Verlassen des Editors (mit `:wq` oder `ZZ`) werden die Fehlermeldungen wieder entfernt, und der Übersetzungslauf beginnt von vorne (sofern Änderungen an der Datei vorgenommen wurden).

Unter MSDOS können leider aus Speicherplatzgründen mit dem MSC-Compiler nur kleinere Quelldateien übersetzt werden.

- Das Programm *indent* führt ein **Pretty-Printing** von C-Quelltexten durch. Über die Profile-Datei `indent.pro` können dabei eine Vielzahl von Formatierungsparametern eingestellt werden. Beispiel:

```
indent xzy.c
```

Unter UNIX ist auch ein **C-Beautifler** namens *cb* verfügbar, der allerdings nicht die vielfältigen Einstellmöglichkeiten von *indent* besitzt.

- Das Programm *tabs* (unter UNIX auch `pr -e|-i`) ersetzt je nach Aufruf Tabulatoren durch eine entsprechende Anzahl von Leerzeichen oder Folgen von Leerzeichen durch Tabulatoren.

- Zum Sichern von Dateien und Verzeichnissen (auch rekursiv) steht das Tool *udf* (update files) unter MSDOS und UNIX zur Verfügung.
- Zum Ausdrucken mit Datum, Uhrzeit, Dateinamen und Seitennumerierung kann unter UNIX das Tool *prsrc* verwendet werden.
- *lint* sucht nach Merkmalen im C-Quelltext, die auf Fehler oder Portabilitätsprobleme hindeuten bzw. Speicherplatzverschwendung anzeigen. Außerdem wird die Handhabung von Datentypen strenger geprüft als von den Compilern.
- Zum Umsetzen von SQL-Anweisungen in C-Quellen steht unter UNIX das Tool *sqlfmt* zur Verfügung. Die zugehörige Beschreibung ist unter dem Namen `sqlfmt.tex` im Verzeichnis `~vpanel/src/scripts/RCS` abgelegt.
- Die Tools *make* und *makedepend* sind sehr nützliche Werkzeuge, um große Programme zu verwalten, die aus mehreren Modulen bestehen. Die Beschreibung dieser Tools steht als eigenständiges Dokument zur Verfügung.
- Projektspezifische Libraries sind zu beachten.
- *gentext* dient zur Verwaltung von sprachabhängigen Texten.
- Für DB-Zugriffe stehen die Bibliotheken *gsq/* und *dbxx* zu Verfügung. Das sind Sammlungen von C-Funktionen und Makros, die es erlauben, daß Anwendungen mit einem SQL-Server in Verbindung treten. Enthalten sind Funktionen, die SQL-Kommandos zum SQL-Server schicken, und die Ergebnisse solcher Kommandos verarbeiten. Weitere Funktionen handhaben Fehlerbedingungen, führen Datenkonvertierungen durch und stellen eine Reihe von Informationen über Wechselwirkungen mit einem SQL-Server bereit.