Wikibook

GNU-Pascal in Beispielen

Inhalt

GNU-Pascal
in Beispielen

ein Wikibook

Falls sie irgendetwas an diesem Buch finden, was falsch ist, fühlen sie sich frei es zu editieren und so auch anderen zu helfen, GNU-Pascal zu lernen.

Vorwort

Herzlich willkommen zu **GNU-Pascal in Beispielen**. Das vorliegende WikiBook möchte Ihnen einen beispielorientierten Überblick über den GNU-Pascal Compiler geben. Die Idee dahinter ist, nur so viel Text wie nötig zu schreiben und so viele Beispiele wie möglich zu präsentieren.

Da dieses Werk an Einsteiger in GNU-Pascal gerichtet ist, wurde auf eine exakte Darstellung der Regeln dieser Sprache, wie sie in anderen Einführungen in Form von Syntaxdiagrammen geboten wird, verzichtet.

Die Sprache Pascal wurde zu Beginn der 1970er Jahre von N. Wirth entwickelt und ist ein direkter Nachfahre der Sprache Algol60. Das GNU-Pascal-Projekt wurde 1988 von J. Virtanen ins Leben gerufen und wird gegenwärtig von vielen Unterstützern gepflegt.

GNU-Pascal läuft auf einer Vielzahl von Betriebssystemen, darunter Linux, FreeBSD, NetBSD, OpenBSD, Mac OS X, CygWin, DOS, GNU HURD, MinGW32, OS/2 und Solaris und kann zwischen diesen Systemen als Cross-Compiler eingesetzt werden, also beispielsweise unter Linux Programme erzeugen, die unter DOS lauffähig sind.

Alles am GNU-Pascal Compiler ist Freie Software. Das berechtigt jeden Lizenznehmer dazu, das Programm zu jedem erdenklichen Zweck einzusetzen, das Programm wie auch seinen Quellcode zu kopieren und zu verbreiten, das Programm zu modifizieren und die modifizierte Version zu verbreiten.

Heutzutage ist das Schreiben von Software, egal ob es sich dabei um Freie Software oder proprietäre[1] Software handelt, bedroht durch Softwarepatente. Informieren Sie sich bitte über dieses Thema unter *http://patinfo.ffii.org/*.

Den GNU-Pascal Compiler können Sie unter folgender Adresse downloaden: *http://www.gnu-pascal.de/*.

Weitere Informationen über GNU-Pascal bekommen Sie, falls Sie das Programm **info** installiert haben mit **info -f gpc**. Nützlich sind darüber hinaus die *GNU-Pascal Coding Standards*, die in einer deutschsprachigen Übersetzung mit **info -f gpcs-de** zu finden sind. Mailinglisten zum Austausch von Informationen rund um GNU-Pascal können Sie abonnieren, in dem Sie eine Mail an **majordomo@gnu.de** schreiben mit dem *subscribe gpc* für die englischsprachige oder *subscribe gpc-de* für die deutschsprachige Mailingliste.

1 Als *proprietär* bezeichnet man Software, die nicht frei ist.

Einstieg in GNU-Pascal

In diesem Kapitel wird die Struktur eines GNU-Pascal-Programms demonstriert. Darüber hinaus wird gezeigt, wie man mit dem GNU-Pascal Compiler umgeht.

Das erste Programm

Der erste Quelltext sieht wie folgt aus:

Das erste Programm

```
{ Gibt 'Hallo, Welt!' aus }
program Erstes;
begin
  WriteLn ('Hallo, Welt!')
end.
```

Dieser Quelltext muss nur noch in die "Sprache des Computers" übersetzt werden, damit dieser die von uns geschriebenen Anweisungen ausführen kann. Der entsprechende Befehl, der innerhalb einer Shell oder einer entsprechenden grafischen Programmierumgebung ausgeführt werden kann, lautet

```
gpc erstes.pas -o erstes
```

Dabei wird angenommen, dass die Datei, unter welcher der Quelltext gespeichert wurde, *erstes.pas* lautet.

Die Option **-o** bewirkt, dass eine Datei erzeugt wird, welche *erstes* heißt[2]. Diese Datei ist ausführbar. Ausführen kann man sie, indem man auf der Kommandozeile **./erstes** eingibt.

Erklärung

In der ersten Zeile befindet sich ein Kommentar. Kommentare werden mit einer geschweiften Klammer[3] "**{**" eingeleitet, danach darf beliebig viel Text auch über mehrere Zeilen folgen. Dieser Text darf kein weiteres Kommentarzeichen enthalten. Das Ende des Kommentars ist eine geschlossene geschweifte Klammer**}**.

program ist das erste Schlüsselwort. Es teilt dem Compiler mit, dass der nun folgende Quelltext ein Programm bedeutet. Außerdem wird hier der Name des Programms angegeben. Der Name des ausführbaren Programms, welcher durch die Option **-o** bestimmt wurde und der Name neben der **program**-Anweisung müssen nicht übereinstimmen. Im Sinne der Lesbarkeit von Quelltexten ist dies aber von Vorteil.

Zwischen **begin** und **end** befinden sich alle Aktionen, die das Programm ausführen soll. Die einzige Aktion, die unser erstes Programm auszuführen hat, besteht darin, den Text "Hallo, Welt!" auf den Bildschirm zu schreiben. Die passende Anweisung lautet **WriteLn**. Das **Ln** bewirkt einen Zeilenvorschub. Text muss immer zwischen Apostrophe gesetzt werden. Soll der Apostroph selbst ausgegeben werden, so sind zwei Apostrophe zu schreiben. Wie am Ende eines Satzes ein Punkt steht, so steht auch am Ende eines Programms ein Punkt.

---

2 Auf betriebssystemspezifische Unterschiede der erzeugten Dateinamen gehen wir im Anhang A, →Systemspezifische Details ein.

3 Außer den geschweiften Klammern gibt es noch weitere Arten von Kommentaren. Hierzu gehören die einzeiligen Kommentare, die mit "//" eingeleitet werden und bis zum Ende der Zeile reichen, wie auch Kommentare, die durch je zwei Zeichen umschlossen werden (* Kommentar *).))

```
// Kommentar bis zum Ende der Zeile
 (* Kommentar
 ... mehrzeilig *)
```

Das zweite Programm

Der zweite Quelltext sieht wie folgt aus:

Das zweite Programm

```
program Zweites;

begin
  Write ('Dieser Text enthält das ''-Zeichen');
  WriteLn
end.
```

Auch dieses Programm lässt sich übersetzen, indem **gpc zweites.pas -o zweites** eingeben wird um anschließend mit der Eingabe von **./zweites** ausgeführt werden zu können.

Erklärung

Die erste **Write**-Anweisung schreibt in bekannter Weise einen Text auf den Bildschirm. Diesmal wird das Apostroph einmal ausgegeben. Nach der ersten **Write**-Anweisung wird kein Zeilenvorschub durchgeführt. Dieser wird hingegen durch die zweite **WriteLn**-Anweisung bewirkt. Zwischen zwei Anweisungen muss ein Semikolon stehen, wobei **begin** und **end** nicht als Anweisungen gelten, sie dienen lediglich der Gruppierung.

Eingabe

Programme dienen häufig dazu, Eingaben entgegenzunehmen und diese zu verarbeiten. Im folgenden Programm beschäftigen wir uns mit der Eingabe von Daten.

Eingabe

```
{ Liest einen Text ein }
program Eingabe1;

var
  Name: String (10);

begin
  Write ('Bitte geben Sie Ihren Namen ein: ');
  ReadLn (Name);
  WriteLn ('Guten Tag, ', Name)
end.
```

Erklärung

Zwischen **var** und **begin** werden Variablen deklariert. Variablen dienen dazu, Werte, die später nochmal verwendet werden können, zu speichern. Der Name der Variablen lautet **Name**, dann folgt ein Doppelpunkt, um den Namen von seinem Typ zu trennen. Der Typ ist in diesem Fall eine Zeichenkette (**String**). Später werden wir sehen, dass es noch viele weitere Typen gibt und die Möglichkeit besteht, sich eigene Typen zu erzeugen. In unserem Fall wird eine Zeichenkette gespeichert, die maximal 10 Zeichen lang sein kann.

Das Programm fragt nach dem Namen. Wenn dieser eingegeben wurde, indem die Routine **ReadLn** ihre Eingabe gelesen und in der Variablen **Name** gespeichert hat, dann wird "Guten Tag," gefolgt von ihrem Namen ausgegeben. Sie sehen in der letzten **WriteLn**-Anweisung, dass mehrere Zeichenketten durch Kommas getrennt aneinandergehängt werden können. **ReadLn** wartet übrigens so lange auf das Ende Ihrer Eingabe, bis Sie die ENTER-Taste gedrückt haben. Geben Sie mehr als 10 Zeichen ein, so wird die Eingabe nach dem 10. Zeichen abgeschnitten.

Schreibweise von Programmen

Niemand verbietet es uns, das obige Programm folgendermaßen zu schreiben:

Schreibweise

```
progRAM eingabe2;vaR Name:StrIng(10);begin
WrItE('Bitte geben Sie Ihren Namen ein: ');READLN (NaME);
WriteLn ('Guten Tag, ', Name)END.
```

Dieser Stil ist sicher sehr persönlich. Wartungsfreundlich ist ein solcher Quellcode jedoch nicht, vor allem wenn es sich um längere Programme handelt. Aus diesem Grunde wurden die *GNU-Pascal-Coding-Standards* (GPCS) entwickelt, an die man sich bei dem Schreiben von GNU-Pascal-Programmen halten darf. Man sollte sich daran halten, wenn man Programme veröffentlichen möchte und darauf hofft, weitere Programmierer zur Mitarbeit zu bewegen. Im Übrigen ist es gerade für Anfänger sehr hilfreich, sich sofort den richtigen Schreibstil beim Programmieren anzugewöhnen.

Variablen und Typen

Dieses Kapitel gibt einen Überblick über die Verwendung von Variablen und den dazugehörigen Typen.

Variablendeklarationen

Wie im ersten Kapitel angedeutet, werden Variablen innerhalb eines Bereiches deklariert, der mit **var** eingeleitet wird. Mehrere Variablen vom gleichen Typ werden durch Kommas voneinander getrennt aufgeführt.

Variablen haben, wenn sie auf diese Weise deklariert werden, einen zufälligen Wert. Man sollte sich wirklich nicht darauf verlassen, dass dieser Wert bei Variablen vom Typ Zahl immer Null und bei Zeichenkettenvariablen immer die leere Zeichenkette ist. Tatsächlich handelt es sich dabei um ein Abbild des Computerspeichers. Variablen kann man bei der Deklaration initialisieren, wobei es zwei alternative Schreibweisen gibt:

Programm: Initialisieren

```
program Initialisieren;

var
  Text1: String (10) = 'Hallo,';
  Text2: String (10) Value 'Welt';
  Text3: String (10);

begin
  Text3 := '!';
  WriteLn (Text1, ' ', Text2, Text3)
end.
```

Erklärung

Es werden drei Variablen deklariert, **Text1**, **Text2** und **Text3**, allesamt vom Typ **String (10)**. Die Variable **Text1** wird mit dem Gleichheitszeichen und **Text2** wird mit dem Schlüsselwort **Value** initialisiert. **Text3** wird innerhalb des Programmkörpers ein Wert zugewiesen. Wertzuweisungen erfolgen immer mit **:=**.

Variablennamen

Variablennamen dürfen grundsätzlich alle Kombinationen aus Buchstaben[4] und Ziffern sein, sofern am Anfang des Wortes ein Buchstabe vorkommt und der Name selbst kein reserviertes Wort in Pascal ist[5].

Namen von Objekten, wie Programmen und Variablen, heißen auch "Bezeichner".

- **Gültige Bezeichner:** Name, Bezeichner, Nummer, Anzahl, Siebzehn, i, k1, Maus1, Programm, Beginne
- **Ungültige Bezeichner:** 1.Element, @Name, Müller, begin, program, GNU-Pascal

Der Unterstrich "_" darf auch vorkommen. Der Grund, weswegen an dieser Stelle auf Variablennamen aufmerksam gemacht wird ist folgende Regel: Namen von Bezeichnern sollten aussprechbar sein und eine Aussage haben. Es macht keinen Sinn im Sinne der Lesbarkeit von Programmen, wenn Variablen allesamt Bezeichner der Art k1, k2, k3, ... sind. Bezeichner sollten grundsätzlich mit großem Anfangsbuchstaben geschrieben werden. Bei zusammengesetzten Worten sollte jedes Teilwort mit großem Anfangsbuchstaben beginnen (z.B. **AutoKennzeichen**).

Vordefinierte Typen

Dieser Abschnitt befasst sich mit den in GNU-Pascal vordefinierten Typen.

Ganze Zahlen

Typen für ganze Zahlen werden grundsätzlich unterschieden in solche, die nur einen positiven Wertebereich haben (vorzeichenlos) und solche, deren Wertebereich auch negative Zahlen umfassen kann (vorzeichenbehaftet). Folgende Typen stehen zur Verfügung:

Vorzeichenbehaftet	Vorzeichenlos
ByteInt	ByteCard, Byte
ShortInt	ShortCard, ShortWord
Integer	Cardinal, Word
MedInt	MedCard, MedWord
LongInt	LongCard, LongWord

Falls nicht besondere Gründe dagegen sprechen, sollte immer **Cardinal** oder **Integer** benutzt werden.

4 ä, ü, ö und ß gelten in diesem Zusammenhang nicht als Buchstaben

5 Bei manchen reservierten Worten ist es durchaus erlaubt, diese als Bezeichner zu verwenden. Im Sinne der Lesbarkeit von Programmen sollten Sie auf dieses Feature verzichten.

Rechnen mit ganzen Zahlen

Das folgende Programm demonstriert die Grundrechenarten mit ganzen Zahlen:

Programm: Rechnen

```
program Rechnen;

var
  Zahl1, Zahl2, Ergebnis: Integer;

begin
  Zahl1 := 10;
  Zahl2 := 3;
  Ergebnis := Zahl1 + Zahl2;     { Ergebnis ist 13 }
  Ergebnis := Zahl1 - Zahl2;     { Ergebnis ist 7 }
  Ergebnis := Zahl1 * Zahl2;     { Ergebnis ist 30 }
  Ergebnis := Zahl1 Div Zahl2;   { Ergebnis ist 3 }
  Ergebnis := Zahl1 Mod Zahl2;   { Ergebnis ist 1 }
  WriteLn ('Ergebnis ist nun: ', Ergebnis)
end.
```

Erklärung

Zuerst werden drei Variablen für ganze Zahlen deklariert. Anschließend werden einige Rechenoperationen auf diese Variablen ausgeführt, die Ergebnisse dieser Operationen stehen in den Kommentaren neben den Zuweisungen. Da die Rechnung 10 / 3 ein Ergebnis von 3,333333... liefern würde, dieses Ergebnis aber nicht in eine Variable vom Typ **Integer** passt muss die "geteilt durch"-Operation durch **Div** ersetzt werden. 10 / 3 ist bei ganzen Zahlen 3 Rest 1. Den Rest erhalten wir durch den Operator **Mod**.

Fließkommazahlen

Fließkommazahlen sind solche, die Nachkommastellen haben. Der Operator "geteilt durch" ist so definiert, wie man es naiv erwarten würde. Fließkommazahlen sind immer vorzeichenbehaftet, sie unterscheiden sich untereinander in der Genauigkeit, die mit steigender Größe wächst[6].

Folgende Fließkommatypen stehen zur Verfügung

- Single,
- ShortReal,
- Real,
- Double,
- LongReal,
- Extended

Das folgende Beispiel implementiert einen einfachen Euro-Nach-DM Umrechner:

6 Falls Zahlen mit nahezu beliebiger Genauigkeit benötigt werden, so kann die Unit "GMP" eingebunden werden. Auf Units und Module wird in einem gesonderten Kapitel eingegangen

Programm: EuroRechner

```
program EuroRechner;

var
  EuroNachDM: Real Value 1.95583;
  Betrag: Real;

begin
  Write ('Wie viel Geld haben Sie in der Tasche? ');
  ReadLn (Betrag);
  WriteLn ('Sie hätten jetzt ', Betrag * EuroNachDM, ' DM')
end.
```

Erklärung

Es werden zwei Fließkommazahlen deklariert, wobei eine mit dem Umrechnungskurs von Euro nach D-Mark initialisiert[7] wird. Ein einzugebender Euro-Betrag wird innerhalb der **WriteLn**-Anweisung nach DM umgerechnet. Ein Programmlauf offenbart allerdings eine Schwierigkeit:

```
Wie viel Geld haben Sie in der Tasche 12.77
Sie hätten jetzt  2.497594910000000e+01 DM
```

Es sind zu viele Nachkommastellen und die Zahl selbst wird in einer schlecht lesbaren Notation angezeigt. Es handelt sich dabei um den Betrag $2.4976 * 10^1$DM. Wünschenswert wäre eine Anzeige, die uns zwei Nachkommastellen gerundet ausgibt:

Programm: Eurorechner 2

```
Program EuroRechner2;

var
  EuroNachDM: Real Value 1.95583;
  Betrag: Real;

begin
  Write ('Wie viel Geld haben Sie in der Tasche ');
  ReadLn (Betrag);
  WriteLn ('Sie hätten jetzt ', Betrag * EuroNachDM : 0 : 2, ' DM')
end.
```

Erklärung

Die gewünschte Ausgabe erhalten wir durch Formatangaben in der **WriteLn**-Anweisung:

```
Wie viel Geld haben Sie in der Tasche 12.77
Sie hätten jetzt 24.98 DM
```

Die Zahl nach dem ersten Doppelpunkt bedeutet die gesamte Anzahl von Stellen, einschließlich des Punktes und der Nachkommastellen. Gibt man hier eine "0" ein, so bedeutet das eine durch die Zahl selbst und die Anzahl der Nachkommastellen bestimmte Anzahl von Stellen. Egal wie groß die Zahl ist, sie wird immer korrekt dargestellt. Die Zahl nach dem zweiten Doppelpunkt bedeutet die Anzahl der Nachkommastellen. Anstelle konkreter Zahlen dürfen hier auch Variablen vom ganzzahligen Typ eingesetzt werden.

Manchmal ist es nötig, eine Fließkommazahl in eine ganze Zahl zu überführen. Hierzu stehen zwei Funktionen zur Verfügung:

7 Diese Variable sollte besser eine Konstante sein. Mehr über Konstanten im →Kapitel Konstanten.

Programm: Runden

```
program Runden;

begin
  WriteLn ('2.7 gerundet      = ', Round (2.7));
  WriteLn ('2.7 abgeschnitten = ', Trunc (2.7))
end.
```

Erklärung

Ist der Abstand einer Zahl zur nächsten ganzen Zahl kleiner oder gleich 0.5, so rundet die Funktion **Round** auf, sonst ab. **Trunc** hingegen entfernt die Nachkommastellen, rundet also immer ab.

Zeichenketten

In Kapitel „Einstieg in GNU Pascal“ wurde bereits ein einführendes Beispiel zum Umgang mit Strings gezeigt, so dass wir hier darauf verzichten werden und uns den fortgeschrittenen String-Techniken widmen können.

Programm: StringTest

```
program StringTest;

var
  Text: String (1000);
  TextLaenge: Integer;

begin
  Write ('Bitte geben Sie einen Text ein: ');
  ReadLn (Text);
  Text := 'Eingabe: ' + Text;
  TextLaenge := Length (Text);
  WriteLn (', Text,  ist ', TextLaenge, ' Zeichen lang.' )
end.
```

Erklärung

Ein String wird immer mit einer maximalen Länge deklariert, der so genannten "Kapazität". Gibt ein Anwender einen Text ein, so wird diese Zeichenfolge an die Zeichenkette "Eingabe:" gehängt. Dies geschieht mit dem +-Zeichen. Die gesamte Länge des Textes wird durch die Funktion **Length** ermittelt. Leider ist es nicht möglich, beliebig lange Zeichenketten aufzunehmen. Der größte Wert, den die Kapazität annehmen kann ist allerdings so groß wie die größte Zahl vom Typ **Cardinal**, so dass sie vermutlich immer genug Platz zur Verfügung haben.

Eine wesentlich elegantere Methode zum Erzeugen eines Strings aus verschiedenen Teilen besteht in der Verwendung einer auf diesen Zweck spezialisierten Funktion:

Programm: StringTest2

```
program StringTest2;

var
  GesamtText, Eingabe: String (1000);
  TextLaenge: Integer;

begin
  Write ('Bitte geben Sie einen Text ein: ');
  ReadLn (Eingabe);
  TextLaenge := Length (Eingabe);
  WriteStr (GesamtText, Eingabe : TextLaenge + 3, TextLaenge : 5);
  WriteLn (GesamtText)
end.
```

Erklärung

Die Prozedur[8] **WriteStr** erlaubt es, den Text beim Aneinanderhängen zu formatieren. Hierbei wird auch gezeigt, wie Variablen als Formatangabe benutzt werden können. Der Text der Eingabe wird so formatiert, als habe er drei Zeichen mehr als er hat[9]. Diese Zeichen werden mit Leerstellen aufgefüllt. Die Ausgabe der Textlänge wird mit einer Gesamtlänge von 5 Zeichen angegeben. Somit wird auch hier die niederwertigste Ziffer 5 Stellen eingerückt. Folgende zwei Programmläufe verdeutlichen das:

```
Bitte geben Sie einen Text ein: hallo welt
  hallo welt   10
Bitte geben Sie einen Text ein: Tag!
  Tag!    4
```

Enthält ein String eine Zahl, so kann es nötig sein diese in eine Zahl vom Typ **Integer** oder **Real** umzuwandeln. Gründe dafür könnten sein, dass mit dieser Zahl weitergerechnet werden soll oder eine Umwandlung bessere Darstellungsmöglichkeiten mit Hilfe von Formatangaben, wie wir sie oben besprochen haben, bietet

Programm StringNachZahl

```
program StringNachZahl;

var
  Zahlentext: String (20) = '1.234';
  Zahl: Real;
  Fehler: Integer;

begin
  Val (Zahlentext, Zahl, Fehler);
  WriteLn (Zahl : 0 : 2, '  Fehlermeldung= ', Fehler)
end.
```

Erklärung

Die Prozedur **Val** wandelt einen String in eine Fließkommazahl oder eine ganze Zahl, je nachdem, von welchem Typ das Argument ist. Tritt ein Fehler auf, weil der umzuwandelnde String keine Zahl repräsentiert, so wird die

8 Auf Details zu Prozeduren, Funktionen und Routinen gehen wir in einem späteren Kapitel ein. Bisher reicht die Definition, dass all diese Bezeichnungen für Ausdrücke stehen, die etwas bewirken.

9 Diese Art der Formatierung kennen Sie bereits aus dem Abschnitt über Fließkommazahlen

Variable **Fehler** auf einen Wert gesetzt[10] der verschieden von **0** ist.

Der Typ **Char** dient dazu, genau ein Zeichen aufzunehmen. Zeichen müssen dabei nicht notwendigerweise druckbar sein sondern können auch Steueraufgaben übernehmen[11]. Zeichen werden genauso gelesen und ausgegeben wie Strings, daher verzichten wir hier auf ein einführendes Beispiel. Zu jedem Zeichen gehört eine Repräsentation als Zahl im Bereich von 0 bis 255[12], die man ermitteln kann:

Programm: Zahlencodes

```
program Zahlencodes;

var
  Buchstabe: Char;

begin
  Write ('Geben Sie einen Buchstaben ein: ');
  ReadLn (Buchstabe);
  WriteLn ('Zahlencode von ', Buchstabe, ' : ', Ord (Buchstabe));
  WriteLn ('Nächster Buchstabe: ', Chr (Ord (Buchstabe) + 1))
end.
```

Erklärung

Die Funktion **Ord** gibt zu jedem Buchstaben den dazugehörigen Zahlenwert zurück. **Chr** hingegen liefert den Buchstaben zu einem bestimmten Zahlenwert. Diese Zahlenwerte lassen sich addieren, damit liefert **Chr (Ord (Buchstabe) + 1)** den nächsten Buchstaben zurück.

Der Typ Boolean

Boolsche Werte gibt man üblicherweise nicht ein, man erhält sie aus bestimmten Abfragen. So liefert die Aussage $3<4$ den Wert "Wahr". In GNU-Pascal wird dies repräsentiert durch **True**, das Gegenteil davon ist **False**. Nebenbei gilt "'*False*"' < "'*True*"'.

Programm: BoolTest

```
program BoolTest;

var
  Wahrheit: Boolean;

begin
  Wahrheit := 3 < 4;
  WriteLn (Wahrheit)
end.
```

Zu Boolschen Werten gehören die Operatoren **not**, **and**, **or** und **xor** mit folgenden Regeln:

A	B	not A	A and B	A or B	A xor B

10 Diese Art der Übergabe bezeichnet man als "Call-By-Reference". Mehr darüber im →Kapitel Call by Reference
11 Beispielsweise übernehmen die Tasten `Tabulator`, `Backspace` und `Enter` nur Steueraufgaben.
12 Gebräuchlich sind dabei die so genannten ASCII- oder ANSI-Codes.

True	True	False	True	True	False
True	False	False	False	True	True
False	True	True	False	True	True
False	False	True	False	False	False

Aufzählungstypen

Aufzählungstypen dienen dazu, eine bestimmte Wertemenge exakt festzulegen. Das könnten Spielkartenfarben sein oder Automarken.

Programm: Aufzaehlung1

```
program Aufzaehlung1;

var
  Farbe: (Rot, Gelb, Blau);

begin
  Farbe := Rot;
  WriteLn (Farbe = Rot, ' ', Farbe = Gelb, ' ', Farbe = Blau);
  Farbe := Succ (Farbe);
  WriteLn (Farbe = Rot, ' ', Farbe = Gelb, ' ', Farbe = Blau);
  Farbe := Pred (Blau);
  WriteLn (Farbe = Rot, ' ', Farbe = Gelb, ' ', Farbe = Blau);
  WriteLn ('Ord: ', Ord (Gelb));
end.
```

Erklärung

Die möglichen Werte werden bei der Deklaration in Klammern gesetzt. Ab da kann **Farbe** nur die Werte **Rot**, **Gelb** oder **Blau** annehmen. Die Funktion **Succ** gibt den Nachfolger und **Pred** den Vorgänger eines bestimmten Elementes an. Die Reihenfolge der Elemente der Aufzählung werden bei der Deklaration festgelegt. So ist **Blau** der Nachfolger von **Gelb**. Die Funktion **Ord** liefert die zahlenmäßige Entsprechung des Elementes. Der Vorgänger von **Rot** und der Nachfolger von **Blau** sind nicht definiert.

Unterbereichstypen

Unterbereichstypen schränken einen möglichen Wertebereich ein. So ist der Typ **Byte** ein Unterbereich vom Typ **Cardinal**. Unterbereichstypen werden gebildet, indem die untere und die obere Grenze der Werte angegeben werden. Folgendes Beispiel verdeutlicht das:

Programm: Unterbereich

```
program Unterbereich;

var
  KleinBuchstaben: 'a'..'z';
  Ziffern: 0..9;

begin
  Ziffern := 5;
  WriteLn (Ziffern, '  ', Succ (Ziffern))
end.
```

Erklärung

KleinBuchstaben und **Ziffern** sind zwei Unterbereichstypen. Der Zahlenwert der unteren Grenze muss kleiner sein als der Zahlenwert der oberen Grenze. Die Funktionen **Succ** und **Pred** sind auch hier sinnvoll anwendbar.

Arrays

Arrays[13] dienen zur Aufnahme einer Menge gleichartiger Werte. Die Menge der Werte wird bei der Deklaration in Form eines Typs angegeben, ebenso der Typ, der vom Array gespeichert wird.

Programm: Array1

```
program Array1;

var
  MeinArray: array [1..4] of Integer;

begin
  MeinArray[1] := 10;
  MeinArray[2] := 11;
  MeinArray[3] := 12;
  MeinArray[4] := 13
end.
```

Erklärung

Das Array **MeinArray** kann vier Werte aufnehmen, allesamt vom Typ **Integer**. Die Definition der Grenzen (**1..4**) erfolgt über einen Unterbereichstypen, der in eckige Klammern gesetzt wird. **MeinArray[1]** ist das erste Element des Arrays, die **1** hierbei ist der Index, der selbst zum Unterbereichstypen **1..4** passen muss. Das Array ist somit durchnummeriert, die einzelnen Elemente können genauso wie andere Variablen initialisiert werden.

Eine andere Art, Arrays zu deklarieren zeigt folgendes Beispiel:

13 Andere Bezeichnungen für Array sind "Vektor" und "Feld"

Programm: Array2

```
program Array2;

var
  BoolArray: array [Boolean] of String (4);

begin
  BoolArray[False] := 'Nein';
  BoolArray[True]  := 'Ja';
  WriteLn ('Ist 4 kleiner als 2? ', BoolArray[4 < 2])
end.
```

Erklärung

Bei der Deklaration von BoolArray wird kein Unterbereichstyp, sondern ein aufzählbarer Typ verwendet, der **False** und **True** als mögliche Indizes anbietet. Der Inhalt des Arrays sind zwei Strings der Kapazität 4. Da 4 < 2 falsch ist, schreibt die **WriteLn**-Anweisung den String, den **BoolAray[False]** enthält.

Eine Initialisierung des Arrays während der Deklaration sieht wie folgt aus:

Programm: Array3

```
program Array3;

var
  BoolArray: array [Boolean] of String (1) = (, '0');
  Zufall: Integer;

begin
  Zufall := Random (20);
  WriteLn ('Zufallszahl := ', BoolArray[Zufall < 10], Zufall)
end.
```

Erklärung

Die beiden möglichen Werte des Arrays werden innerhalb der Klammer in aufsteigender Reihenfolge der Indizes initialisiert, also zuerst **BoolArray[False]**. Initialisiert wird dieses Array mit den beiden Werten "leerer String" und "0". Die Funktion **Random** liefert eine Zufallszahl zwischen 0 und 19 (20 − 1). Die **WriteLn**-Anweisung schreibt alle Zufallszahlen zweistellig aus, wobei Zufallszahlen, die kleiner als 10 sind, um eine führende "0" (also **BoolArray[True]**) ergänzt werden. Falls die Zufallszahl größer oder gleich 10 ist, so wird der leere String ergänzt, der sich nicht bemerkbar macht.

Mehrdimensionale Arrays, wie sie bei Spielen oder in der Mathematik auftauchen, werden von GNU-Pascal ebenfalls unterstützt. Im folgenden zeigen wir, wie ein 3 * 3 Felder großes TicTacToe[14]-Spielfeld initialisiert wird:

14 Ein Spiel, bei dem es darum geht, drei gleiche Steine in eine Reihe zu legen wobei nacheinander gesetzt wird.

Programm: Array4

```
program Array4;

var
  TicTacToe: array [1..3, 1..3] of (Schwarz, Weiss, Frei);

begin
  { Spielfeld initialisieren }
  TicTacToe[1, 1] := Frei;
  TicTacToe[1, 2] := Frei;
  TicTacToe[1, 3] := Frei;
  TicTacToe[2, 1] := Frei;
  TicTacToe[2, 2] := Frei;
  TicTacToe[2, 3] := Frei;
  TicTacToe[3, 1] := Frei;
  TicTacToe[3, 2] := Frei;
  TicTacToe[3, 3] := Frei
end.
```

Erklärung

Mehrdimensionale Arrays werden deklariert, indem die einzelnen Dimensionen durch Kommas getrennt angegeben werden. Bei der Initialisierung verfährt man genauso.

Auch Strings kann man als Arrays auffassen. Das folgende Programm demonstriert, wie der erste Buchstabe ausgewertet wird:

Programm: StrArray

```
program StrArray;

var
  Wort: String (100);

begin
  Write ('Bitte geben Sie ein Wort ein: ');
  ReadLn (Wort);
  WriteLn ('Der erste Buchstabe des Wortes , Wort,  lautet: ', Wort[1])
end.
```

Erklärung

Die Deklaration des Strings kennen Sie bereits. Neu ist, dass **Wort** auch gleichzeitig ein eindimensionales Array darstellen kann, welches aus Zeichen besteht. Mit **Wort[1]** wird auf das erste Zeichen des Strings zugegriffen. Um auf das letzte Zeichen zuzugreifen, müssten wir **Wort[Length (Wort)]** schreiben. Später im Buch erfahren Sie, wie man alle Zeichen eines Wortes durchläuft.

Mengen

Mengen fassen Elemente vom gleichen Typ zusammen. Eine Menge ist mindestens leer und hat eine abzählbare Anzahl von Elementen. Als Basistypen kommen alle in diesem Kapitel beschriebenen, Typen in Frage außer den Fließkommazahlen.

Programm: Menge1

```
program Menge1;

var
  MeineMenge: set of Char;

begin
  MeineMenge := ['a'..'z', 'B', 'C'];
  WriteLn ('Anzahl der Elemente in der Menge= ', Card (MeineMenge));
  WriteLn ('Ist A in MeineMenge enthalten? ', 'A' in MeineMenge);
  WriteLn ('Ist a in MeineMenge enthalten? ', 'a' in MeineMenge)
end.
```

Erklärung

Eine Menge wird deklariert, indem der Basistyp angegeben wird, auf den diese Menge aufbaut. In unserem Beispiel ist es eine Menge verschiedener Zeichen. Die Menge wird initialisiert, indem alle Elemente angegeben werde, sei es implizit durch einen Bereich '**a'..'z'**, explizit durch die Angabe der einzelnen Elemente oder gemischt wie in obigem Beispiel. Wichtig ist, dass die Elemente in eckigen Klammern angegeben werden. Die Anzahl der Elemente in **MeineMenge** wird mit der Funktion **Card** ermittelt. Für unser Beispiel liefert sie 28 (Anzahl der Kleinbuchstaben plus zwei Großbuchstaben). Für die Menge **MeineMenge := ['A', 'B', 'A', 'C', 'A']** würde **Card** nur **3** liefern, da genau drei verschiedene Elemente in der Menge verfügbar sind. Bei leeren Mengen gibt **Card 0** zurück. Ob ein Zeichen in der Menge enthalten ist oder nicht liefert uns der Operator **in**. Da '**A'** nicht in **MeineMenge** enthalten ist, liefert die erste Abfrage **False**, die Zweite **True**.

Mengenoperatoren

Im folgenden werden die Operatoren, die auf Mengen angewendet werden können, vorgestellt:

- +, Vereinigung, $A \cup B$: Dieser Operator bildet die Vereinigung beider Mengen. Beispiel:

```
['a', 'b'] + ['c', 'd'] = ['a', 'b', 'c', 'd']
```

- *, Durchschnitt, $A \cap B$: Bildet den Durchschnitt beider Mengen, also eine Menge mit nur den Elementen, die in beiden Mengen vorkommen. Beispiel:

```
['a', 'b'] * ['b', 'c'] = ['b']
```

- ><, Symmetrische Differenz $A \times B$: Bildet eine Menge, die aus nicht gemeinsamen Elementen besteht. Beispiel:

```
['a', 'b'] >< ['b', 'c'] = ['a', 'c']
```

- -, Differenz, $A \backslash B$: Bildet eine Menge, die aus Elementen besteht welche nicht in der zweiten Menge vorkommen.Beispiel:

```
['a', 'b', 'c', 'd'] - ['b', 'c'] = ['a', 'd']
```

- <, echte Teilmenge, $A \subset B$: Ist **True**, wenn A eine echte Teilmenge von B ist, also alle Elemente aus A in B enthalten sind aber nicht umgekehrt. Beispiel:

```
['a', 'b'] < ['a', 'b', 'c']
```

- <=, Teilmenge, $A \subseteq B$: Ist **True**, wenn A eine Teilmenge von B ist, also alle Elemente aus A in B enhalten sind oder die Mengen sogar gleich sind. Beispiel:

```
['a', 'b'] <= ['a', 'b', 'c']
```

- =, Gleichheit, $A = B$: Ist **True**, wenn A dieselben Elemente enthält wie B. Beispiel:

```
['a', 'b'] = ['a', 'b']
```

- <>, Ungleich, $A \neq B$: Ist **True**, wenn *A* verschieden von *B* ist. Beispiel:

```
['a', 'b'] <> ['a', 'b', 'c']
```

- **in**, Enthalten, $x \in A$: Ist **True**, wenn *x* in *A* enthalten ist. *x* muss dabei dem Basistyp der Menge entsprechen. Beispiel:

```
'a' in ['a', 'b']
```

Im Kapitel über Konntrollstrukturen wird demonstriert, wie eine Menge zur Anzeige aller ihrer Elemente durchlaufen werden kann.

Konstanten

Bezeichner, denen eindeutige Werte zugewiesen wurden, die sich zur Laufzeit des Programms nicht ändern nennt man Konstanten. Die eigentliche Bedeutung von Konstanten liegt zumeist in der übersichtlichen Gestaltung von Quelltexten. Auch werden Programme durch die Verwendung von Konstanten wartungsfreundlicher. Dazu stellen wir uns vor, wir schrieben ein Programm, welches unseren Rentenversicherungsbeitrag (Ein bestimmter Prozentsatz des Bruttolohns) berechnete. Ändert sich eines Tages der Beitragssatz, so brauchen wir nur an genau einer Stelle das Programm zu ändern. Bei der Wahl der Namen von Konstanten halten wir uns an die in Abschnitt Variablen und Typen-Variablennamen erläuterten Regeln. Im Gegensatz zu Schreibweisen anderer Programmiersprachen sollten Konstanten nicht großgeschrieben werden.

Konstanten im Eigenbau

Konstanten können wir leicht selbst erzeugen:

Programm: Konstanten1

```
program Konstanten1;

const
  Antwort = 42;

begin
  WriteLn ('Die Antwort auf die Frage lautet: ', Antwort);
end.
```

Erklärung

Im **const**-Bereich des Programms werden Konstanten vereinbart. Ihnen wird direkt ein Wert zugewiesen, der während der Laufzeit des Programms nicht verändert werden kann. Konstanten können auch berechnet werden, wie das folgende Beispiel zeigt:

Programm: Konstanten2

```
program Konstanten2;

const
  SeitenLaenge = 10;
  Volumen = Seitenlaenge ** 3;
  Koerper = 'Würfel Eis';

begin
  WriteLn ('Ein ', Koerper, ' mit der Seitenlänge ', Seitenlaenge,
    'dm hat ein Volumen von ', Volumen : 0 : 2, 'L.');
end.
```

Erklärung

Konstanten sind nicht beschränkt auf ganze Zahlen, sondern können von nahezu jedem Basistyp sein. In unserem Beispiel haben wir eine ganzzahlige Konstante **SeitenLaenge**, eine Fließkommakonstante **Volumen** und eine Stringkonstante **Koerper**. Der Operator ****** berechnet 10^3, was genau dem Volumen des Körpers entspricht. An dieser Stelle lässt sich zeigen, dass Konstanten Rechenzeit sparen können: Konstanten brauchen höchstens einmal berechnet werden und können beliebig oft wiederverwertet werden.

Vordefinierte Konstanten

Vordefinierte Konstanten sind häufig nur in bestimmten Kontexten[15] sinnvoll. Hier wird eine kleine Auswahl vordefinierter Konstanten vorgestellt, die recht interessant sein könnten:

Programm: Konstanten3

```
program Konstanten3;

begin
  WriteLn ('MinReal: ', MinReal);
  WriteLn ('MaxReal: ', MaxReal);
  WriteLn ('Nummer des letzten Zeichens: ', Ord (MaxChar));
  WriteLn ('MaxInt: ', MaxInt);
  WriteLn ('Pi: ', Pi);
end.
```

Erklärung

MinReal und **MaxReal** sind die kleinste und größte Zahl, die eine Variable vom Typ **Real** annehmen kann. **MaxChar** ist das letzte Zeichen, welches vom Typ Char ist. **MaxInt** ist die größte Zahl, die eine Variable vom Typ **Integer** annehmen kann. **Pi** ist diejenige Zahl, die das Verhältnis von Umfang zu Durchmesser eines Kreises angibt.

15 Ein Beispiel dafür findet sich im →Kapitel Kommandozeilenparameter.

Kontrollstrukturen

Bislang haben wir die Programmierung als eine lineare Abfolge von Befehlen kennengelernt. Das Programm fing oben an und hörte unten auf, dazwischen war, mehr oder weniger unabhängig von der Benutzereingabe, der komplette Programmfluss statisch vorgegeben. Kontrollstrukturen sorgen dafür, dass je nach Gegebenheit ein anderer Code ausgeführt wird oder ein bestimmter Abschnitt des Programms nach bestimmten Regeln wiederholt wird.

if-then

Die **if**-Abfrage dient dazu, auf Eingaben, Zwischenergebnisse oder "Benutzerfehler" zu reagieren. Das folgende Beispiel ist ein modifiziertes "StringNachZahl"-Programm[16] mit Fehlerabfrage:

Programm: StringNachZahl2

```
program StringNachZahl2;

var
  Zahlentext: String (20);
  Zahl: Integer;
  Fehler: Integer;

begin
  Write ('Geben Sie eine ganze Zahl ein: ');
  ReadLn (Zahlentext);
  Val (Zahlentext, Zahl, Fehler);
  if Fehler = 0 then
    WriteLn ('Hervorragend!, Ihre Zahl lautet: ', Zahl);
  if Fehler <> 0 then
    begin
      WriteLn ('Fehler! Sie sollten doch eine ganze Zahl');
      WriteLn ('eingeben und nicht , Zahlentext, !')
    end
end.
```

Erklärung

Das Programm erwartet als Eingabe eine ganze Zahl, die es als String entgegennimmt und mit Hilfe von **Val** in eine ganze Zahl umwandelt. Wenn der Wert von Fehler bei dieser Umwandlung **0** ist, so wird die dazugehörige Anweisung **WriteLn ('Hervorragend...');** ausgeführt. Den Bereich zwischen **if** und **then** kennen Sie bereits, dabei handelt es sich um einen Boolschen Ausdruck, eine so genannte Bedingung, die zu **True** oder **False** ausgewertet wird. Ergibt die Bedingung **True**, so wird die dazugehörige Anweisung ausgeführt. Die zweite **if**-Abfrage überprüft, ob vielleicht doch ein Fehler aufgetreten ist. Wenn die Variable **Fehler** einen Wert hat, der verschieden von "0" ist, so wird der Anweisungsblock zwischen **begin...end** ausgeführt. Da hier zwei **WriteLn**-Anweisungen ausgeführt werden sollen, muss der Bereich auf diese Weise mit **begin** und **end** zusammengefasst werden. Bei einer einzelnen Anweisung ist es selbstverständlich, dass sie zur vorhergehenden **if**-Anweisung gehört. Bitte beachten Sie bei diesem Beispiel auch die Stellen, an denen Semikolons gesetzt werden: Immer nur zwischen zwei Anweisungen, wobei **end** keine Anweisung darstellt.

Ein Beispiel für zusammengesetzte Bedingungen ist eine Aufschrift auf einem Kinderkarussell: Nur Personen, die maximal 14 Jahre alt und kleiner als 1.2m sind dürfen dort mitfahren. Das Gegenteil dieser Bedingung lautet[17]: Personen, die älter als 14 Jahre sind oder mindestens 1.2m groß sind.

16 vgl. Kapitel Variablen und Typen

17 Nach den Morganschen Regeln: **not (A and B) = not A or not B**.

Folgendes Beispiel verdeutlicht das:

Programm: Kinderkarussell1

```
program Kinderkarussell1;

var
  Alter, Groesse: Integer;

begin
  Write ('Wie alt bist Du?: ');
  ReadLn (Alter);
  Write ('Wie groß bist Du (in cm)?: ');
  ReadLn (Groesse);
  if (Alter <= 14) and (Groesse < 120) then
    WriteLn ('Du darfst auf das Kinderkarussell!');
  if (Alter > 14) or (Groesse >= 120) then
    WriteLn ('Du darfst nicht auf das Kinderkarussell.')
end.
```

Erklärung

Bei zusammengesetzten Bedingungen werden die einzelnen Bedingungen geklammert. Das liegt daran, dass, wie bei der "Punkt- vor Strichrechnung", die Operatoren **and** bzw. **or** eine höhere Priorität haben als **<, <=, >, >=**. Statt **not (Alter <= 14)...** in der zweiten **if**-Abfrage zu verwenden, wurde dieser Ausdruck fertig berechnet.

else

Der **else**-Ausdruck verzweigt in die Alternative zu **if**. Das obige Beispiel des Kinderkarussells sieht umgeschrieben unter Verwendung Verwendung von **else** folgendermaßen aus:

Programm: Kinderkarussell2

```
program Kinderkarussell2;

var
  Alter, Groesse: Integer;

begin
  Write ('Wie alt bist Du?: ');
  ReadLn (Alter);
  Write ('Wie groß bist Du (in cm)?: ');
  ReadLn (Groesse);
  if (Alter <= 14) and (Groesse < 120) then
    WriteLn ('Du darfst auf das Kinderkarussell!')
  else
    WriteLn ('Du darfst nicht auf das Kinderkarussell.')
end.
```

Erklärung

Else spart uns die zweite **if**-Abfrage aus dem oberen Beispiel. Jeder Grund, warum eine Person nicht auf das Kinderkarussell darf wird von diesem Ausdruck abgefangen. Das Programm wird dadurch lesbarer und es wird nur eine (zusammengesetzte) Bedingung ausgewertet, nicht zwei.

Mehrfachverzweigung mit if

Die hier vorgestellten Verzweigungen dürfen beliebig ineinandergeschachtelt werden:

Programm: Kinderkarussell3

```
program Kinderkarussell3;

var
  Alter, Groesse: Integer;

begin
  Write ('Wie alt bist Du?: ');
  ReadLn (Alter);
  if Alter <= 14 then
    begin
      Write ('Wie groß bist Du (in cm)?: ');
      ReadLn (Groesse);
      if Groesse < 120 then
        WriteLn ('Du darfst auf das Kinderkarussell!')
      else
        WriteLn ('Du darfst nicht auf das Kinderkarussell.')
    end
  else
    WriteLn ('Du bist zu alt für das Kinderkarussell.')
end.
```

Erklärung

Dieses Beispiel stuft die zu einem Scheitern des Versuches führenden Gründe, auf das Kinderkarussell zu kommen, viel feiner ab. Außerdem wird bei einem zu hohen Alter keine weitere Abfrage benötigt. Falls das Alter passt, wird der gesamte innere **begin...end**-Block durchlaufen.

Eine andere Art der Mehrfachauswahl stellt das folgende Programm dar:

Programm: Kinderkarussell4

```
program Kinderkarussell4;

var
  Alter: Integer;

begin
  Write ('Wie alt bist Du?: ');
  ReadLn (Alter);
  if Alter <= 14 then
    WriteLn ('Du darfst auf das Kinderkarussell.')
  else if Alter >= 18 then
    begin
      WriteLn ('Wir suchen noch Mitarbeiter für das ');
      WriteLn ('Fahrgeschäft. Bitte melde Dich in der',
        ' Personalabteilung.')
    end
  else
    WriteLn ('Du darfst leider nicht auf das Kinderkarussell')
end.
```

Erklärung

Hier wird aus drei Bereichen ausgewählt: Die Person kann 14 Jahre oder jünger sein, 18 Jahre oder älter oder ein anderes Alter haben. Je nach Alter wird entsprechend verzweigt.

Mehrfachverzweigung mit case

Das letzte Kinderkarussell kann noch viel feiner differenzieren und benötigt dafür viel weniger Zeilen, ist damit übersichtlicher ohne den Komfort zu verlieren:

Programm: Kinderkarussell5

```
program Kinderkarussell5;

var
  Alter: Integer;

begin
  Write ('Wie alt bist Du?: ');
  ReadLn (Alter);
  case Alter of
         0: WriteLn ('Du bist bestimmt zu jung!');
     3..14: WriteLn ('Du darfst auf das Kinderkarussell');
    18..26: begin
              WriteLn ('Wir suchen noch Mitarbeiter für',
                ' das Fahrgeschäft. Bitte melde Dich in');
              WriteLn ('der Personalabteilung.')
            end
    otherwise
      WriteLn ('Du darfst leider nicht auf das Kinderkarussell')
  end
end.
```

Erklärung

Mit **case Alter of** beginnt der so genannte **case**-Block, der erst am gleich eingerückten **end** endet. Dazwischen befinden sich "**case**-Labels", die entweder sehr scharf (**0:**) oder innerhalb eines aufzählbaren und konstanten[18] Bereiches (**3..14:**) das Alter abfragen. Das Schlüsselwort **otherwise** dient dazu, alle Fälle abzufangen, für die kein "**case**-Label" vorgesehen ist. Dies entspricht dem **else** [19] in **if**-Abfragen.

18 Als "**case**-Labels" dürfen keine Variablen verwendet werden.

19 Tatsächlich ist hier auch **else** erlaubt. Die Wahl des Wortes ist Geschmackssache.

Schleifen

Schleifen dienen dazu, sich wiederholenden Programmcode übersichtlich zu formulieren. Ausserdem wird es mit der Verwendung von Schleifen möglich, einen Programmabschnitt beliebig oft zu wiederholen. Letztlich werden nur die Bedingungen notiert, unter denen sich der Code wiederholen muss und der Quellcode wird genau einmal geschrieben. Dieser sich wiederholende Programmteil wird Schleifenkörper oder Schleifenrumpf genannt. Es gibt drei verschiedenen Arten von Schleifen.

Schleifen mit for

Eine **for**-Schleife dient dazu, Programmcode eine bekannte Anzahl Mal ausführen zu lassen:

Programm: Schleife1

```
program Schleife1;

const
  Text = 'Hallo, Welt!';

var
  i: Integer;

begin
  for i := 1 to Length (Text) do
    WriteLn (i, '-ter Buchstabe ist , Text[i], .')
end.
```

Erklärung

Mit dieser Schleife wird jeder Buchstabe des Textes ausgegeben. Die Variable **i** ist dabei zu Beginn **1** und zum Schluss **Length (Text)**, dazwischen hat sie jeden Wert in aufsteigender Reihenfolge genau einmal. Der Schleifenkörper gibt dabei den Wert der Variablen aus und den Buchstaben an der gegenwärtigen Position. Das Schüsselwort **to** in der oberen **for**-Anweisung sorgt dafür, dass vorwärts gezählt wird. Würden wir rückwärts zählen wollen, also den Text in umgekehrter Reihenfolge ausgeben wollen, so müssten wir das Schlüsselwort **downto** verwenden, also **for i := Length (Text) downto 1 do**.

Wie im Kapitel **FF: set-section** über Mengen angemerkt, folgt hier das Beispiel zum Thema Durchlaufen von Mengen:

Programm: Menge2

```
program Menge2;

var
  MeineMenge: set of Char;
  Element: Char;

begin
  MeineMenge := ['a'..'c', 'B', 'C'];
  for Element in MeineMenge do
    begin
      Write ('Meine Menge enthält das Element: ', Element);
      WriteLn (' mit der Ordnung: ', Ord (Element))
    end
end.
```

Erklärung

Da die Mächtigkeit[20]-der Menge zu Beginn der Schleife feststeht, kann die Menge auf diese Weise durchlaufen werden. **Element** enthält zu Beginn des Schleifendurchlaufs allerdings den Wert '**B**', nicht etwa '**a**', wie es die Mengeninitialisierung nahelegt, weil es der Wert mit der kleinsten Ordnung ist, wie sich leicht bei einem Programmlauf feststellen lässt:

```
Meine Menge enthält das Element: B mit der Ordnung: 66
Meine Menge enthält das Element: C mit der Ordnung: 67
Meine Menge enthält das Element: a mit der Ordnung: 97
Meine Menge enthält das Element: b mit der Ordnung: 98
Meine Menge enthält das Element: c mit der Ordnung: 99
```

Der Inhalt von Variablen vom Typ einer Aufzählung (vergl. **FF aufz-section**) lässt sich nicht direkt auf den Bildschirm schreiben. Wie sollte auch der Computer mit einem Befehl umgehen, der "Schreibe Rot auf den Bildschirm" heißt? Meint der Anwender einen roten Apfel oder das Wort "Rot"? Hierbei hilft die **case**-Anweisung:

Programm: Aufzaehlung2

```
program Aufzaehlung2;

var
  Farbe: (Rot, Gelb, Blau);

begin
  for Farbe := Rot to Blau do
    case Farbe of
      Rot:   WriteLn ('Rot');
      Gelb:  WriteLn ('Gelb');
      Blau:  WriteLn ('Blau')
    end
end.
```

Erklärung

Dieses Programm schreibt nacheinander die Worte "Rot", "Gelb" und "Blau" auf den Bildschirm. Schleifen lassen sich vorher beenden, hierzu dient der Ausdruck **Break**. Folgendes Programm zeigt dessen Arbeitsweise:

Programm: Schleife2

```
program Schleife2;

var
  Anzahl: Integer;

begin
  for Anzahl := 1 to 10 do
    begin
      WriteLn (Anzahl, ' ist da.');
      if Anzahl = 3 then
        Break
    end;
end.
```

20 Synonym für Anzahl der Elemente innerhalb der Menge.

Erklärung

Es werden nur die ersten Drei **WriteLn**-Anweisungen ausgeführt. Nach der dritten Anweisung wird **Break** aufgerufen wird. Die Schleife wird damit unverzüglich verlassen.

Ebenso wie es eine Anweisung für das Verlassen von Schleifen gibt, gibt es eine Anweisung, um wieder in den Programmkopf zurückzukehren ohne den Rest des Schleifenkörpers auszuführen:

Programm: Schleife3

```
program Schleife3;

var
  Anzahl: Integer;

begin
  for Anzahl := 1 to 10 do
    begin
      if Anzahl Mod 2 = 0 then
        Continue;
      WriteLn (Anzahl, ' ist da.')
    end;
end.
```

Erklärung

Wenn die **Integer**-Division von **Anzahl** und **2** den Rest **0** ergibt, was bei allen geraden Zahlen der Fall ist, dann sorgt **Continue** dafür, dass sofort in den Schleifenkopf zurückgesprungen wird ohne die benachbarte **WriteLn**-Anweisung auszuführen. Dadurch werden nur ungerade Zahlen ausgegeben.

Schleifen mit while

Mit dem Schlüsselwort **While** wird die abweisende Schleife gebildet. Das Besondere an abweisenden Schleifen ist, dass schon in der Schleifenbedingung überprüft wird, ob diese Schleife überhaupt durchlaufen wird. Damit wird diese Schleife gegebenenfalls nie durchlaufen. Den Schleifenkörper auszuführen kann **while** also abweisen.

Programm: Wuerfelglueck1

```
program Wuerfelglueck1;

const
  ZufallGrenze = 10;
  WunschZahl = 5;

var
  Anzahl: Integer = 1;
  Zufall: Integer;

begin
  Zufall := Random (ZufallGrenze);
  WriteLn ('ZufallsZahl = ', Zufall);
  while Zufall <> WunschZahl do
    begin
      Zufall := Random (ZufallGrenze);
      WriteLn ('ZufallsZahl = ', Zufall);
      Inc (Anzahl)
    end;

  WriteLn ('Gefunden nach ', Anzahl, ' Versuchen')
end.
```

Erklärung

Zuerst wird eine Zufallszahl im Bereich von 0 bis **ZufallGrenze - 1** gezogen. Wenn die Zufallszahl nicht der Wunschzahl entspricht, wird der Schleifenkörper durchlaufen und die Anzahl der Versuche, eine passende Zufallszahl zu ziehen wird protokolliert. Dies geschieht mit Hilfe der Prozedur **Inc**. Diese Prozedur ist gleichbedeutend mit der Anweisung **Anzahl := Anzahl + 1**, was bedeutet, die Variable **Anzahl** um eins zu erhöhen. Am Ende des Programms wird die Anzahl der Versuche ausgegeben. Das "Gegenteil" von **Inc** ist **Dec**, was die Variable um eins erniedrigt.

Schleifen mit repeat...until

Repeat...until ist eine nichtabweisende Schleife. Nichtabweisende Schleifen werden mindestens einmal durchlaufen, weil die Abbruchbedingung am Ende des Schleifenkörpers liegt. Der Schleifenkörper braucht im Falle von mehreren Anweisungen nicht von **begin** und **end** umschlossen zu werden, da **repeat** und **until** selbst einen Block bilden. Ein positiver Nebeneffekt bei der Benutzung dieser Schleife ist die geringere Einrückung. Obiges Beispiel sieht umgeschrieben so aus:

Programm: Wuerfelglueck2

```
program Wuerfelglueck2;

const
  ZufallGrenze = 10;
  WunschZahl = 5;

var
  Anzahl: Integer = 0;
  Zufall: Integer;

begin
  repeat
    Zufall := Random (ZufallGrenze);
    WriteLn ('ZufallsZahl = ', Zufall);
    Inc (Anzahl)
  until Zufall = WunschZahl;
  WriteLn ('Gefunden nach ', Anzahl, ' Versuchen')
end.
```

Erklärung

Im Gegensatz zu **Wuerfelglueck1** wird die Variable **Anzahl** hier mit **0** initialisiert. Der Schleifenkörper, ab **repeat** wird mindestens einmal durchlaufen, dabei wird der Zufallswert ermittelt und die Anzahl der Versuche auf "1" gesetzt. Der Schleifenkopf soll durchlaufen werden, bis **Zufall = WunschZahl** ist. Dann wird, wie im entsprechenden **while**-Beispiel, die Anzahl der Versuche ausgegeben.

Programmierbeispiel: Zahlenraten

Das folgende Beispiel implementiert mit den uns bekannten Mitteln das Spiel "Zahlenraten", bei dem es darum geht, eine Zufallszahl zwischen 1 und 100 zu erraten. Die einzigen Hilfestellungen sind die Hinweise "Zu groß" und "Zu klein":

Programm: Zahlenraten

```
program Zahlenraten;

const
  ObereGrenze = 100;

var
  Anzahl: Integer = 0;
  Zufallszahl, RateZahl: Integer;

begin
  Zufallszahl := Random (ObereGrenze) + 1;
  Write ('Ich denke mir eine Zufallszahl zwischen 1 und ');
  WriteLn (ObereGrenze, ' aus, die Sie raten müssen.');
  repeat
    Write ('RateZahl: ');
    ReadLn (RateZahl);
    Inc (Anzahl);
    if RateZahl > ZufallsZahl then
      WriteLn ('Zu groß!')
    else if RateZahl < ZufallsZahl then
      WriteLn ('Zu klein!')
  until Zufallszahl = RateZahl;
  WriteLn ('Gefunden nach ', Anzahl, ' Versuchen')
end.
```

Erklärung

Mit **Random** wird eine Zufallszahl erzeugt, die es zu raten gilt. Vom Benutzer wird ein Rateversuch eingegeben, die Anzahl der Versuche, die richtige Zahl zu finden wird protokolliert (**Inc (Anzahl)**). Je nachdem, ob die geratene Zahl zu groß oder zu klein ist, wird ein Hinweis ausgegeben. Der Schleifenkörper wird so lange durchlaufen, wie die Zufallszahl und die Ratezahl nicht übereinstimmen.

Programmierbeispiel: Palindrom

Ein Palindrom ist ein Wort[21], welches von hinten und von vorne gleich gelesen werden kann. Beispiele für solche Worte sind "OTTO" und "UHU". Dieses Programm testet ein Wort darauf, ob es ein Palindrom ist:

Programm: Palindrom

```
program Palindrom;

var
  Wort: String (100);
  i, Laenge: Integer;
  IstPalindrom: Boolean = True;

begin
  WriteLn ('Das Programm untersucht, ob ein eingegebenes');
  WriteLn ('Wort ein Palindrom ist.');
  Write ('Geben Sie ein Wort ein: ');
  ReadLn (Wort);
  Laenge := Length (Wort);
  for i := 1 to Laenge div 2 do
    begin
      if Wort[i] <> Wort[Laenge - i + 1] then
        begin
          IstPalindrom := False;
          Break
        end
    end;
  if IstPalindrom then
    WriteLn (', Wort,  ist ein Palindrom.')
  else
    WriteLn (', Wort,  ist kein Palindrom.')
end.
```

Erklärung

Vom eingegebene Wort wird die Länge benötigt, da der Algorithmus überprüft, ob links und rechts der Mitte des Wortes die Buchstaben übereinstimmen.

Der Algorithmus sieht wie folgt aus: Wenn der erste Buchstabe und der Letzte nicht übereinstimmen, so ist das Wort kein Palindrom. Wenn der Zweite und der vorletzte Buchstabe nicht übereinstimmen, so ist das Wort ebenfalls kein Palindrom..., sonst ist es ein Palindrom. Die Bedingung **Wort[i] <> Wort[Laenge - i + 1]** überprüft genau diese Bedingung. **i** ist dabei der Zähler, der bis maximal zur Mitte des Wortes reicht[22].

Wort[i] und **Wort[Laenge - i + 1]** sind spiegelbildliche Buchstabenpositionen innerhalb des Wortes. Die **+1** kommt daher, dass der Anfangsindex des Wortes Eins ist. Wenn zu Beginn auf das letzte Zeichen des Wortes zugegriffen werden soll, so ist damit **Wort[Laenge]** gemeint, nicht etwa **Wort[Laenge - i]** (wobei i = 1 ist). Die Addition von **1** führt genau zu dieser Korrektur.

21 Ein Palindrom kann auch ein Satz sein. In diesem Beispiel würden die Stellungen der Leerzeichen mitberücksichtigt werden, so dass bekannte palindrome Sätze nicht funktionieren.

22 Bei ungeraden Wortlängen bis ein Buchstaben vor der Mitte, da einbuchstabige Worte palindrom sind. Bei geraden Wortlängen gibt es keinen mittleren Buchstaben oder derer zwei, wie man will

Typen im Eigenbau

Die grundlegenden Typen wurden bereits im Kapitel Variablen und Typen besprochen. Dieses Kapitel beschäftigt sich damit, auf der Basis dieser grundlegenden Typen eigene Typen zu entwickeln.

Üblicherweise wird selbstdefinierten Typen immer ein "T" vorangestellt.

Eigene Stringtypen

Grundsätzlich ist an dieser Stelle bekannt, wie Strings mit einer bestimmten Kapazität erzeugt werden. Statt aber, wie in Kapitel Zeichenketten alle Strings mit ihrer Kapazität anzugeben, ist es viel bequemer, sich eigene Stringtypen selbst zu erzeugen:

Program: Stringtyp

```
program Stringtyp;

const
  Kapazitaet = 20;

type
  TMeinString = String (Kapazitaet);

var
  MeinString: TMeinString;

begin
  MeinString := 'Hallo, Welt!';
  WriteLn (MeinString)
end.
```

Erklärung

Im Bereich von **type** werden eigene Typen definiert. **TMeinString** ist damit ab sofort ein Synonym für **String (20)**. Dieser Typ kann im **var**-Bereich sofort genutzt werden.

Die hier gezeigte Reihenfolge von **const**, **type** und **var** ist die natürlichste Art der Anordnung. Wenn keine Gründe gegen diese Reihenfolge sprechen, so sollte sie beibehalten werden.

Eigene ganzzahlige Typen

Manchmal ist es nützlich, eigene ganzzahlige Typen zu definieren. Das folgende Programm demonstriert die Vorgehensweise:

Program: Integertyp

```
program Integertyp;

type
  TMeinInteger = Integer (3);
  TMeinCardinal = Cardinal (3);

begin
  WriteLn ('Kleinster TMeinInteger Wert:  ', Low (TMeinInteger));
  WriteLn ('Groesster TMeinInteger Wert:  ', High (TMeinInteger));
  WriteLn ('Kleinster TMeinCardinal Wert: ', Low (TMeinCardinal));
  WriteLn ('Groesster TMeinCardinal Wert: ', High (TMeinCardinal))
end.
```

Erklärung

Es werden zwei verschiedene ganzzahlige Typen definiert, wobei einer vorzeichenbehaftet, der Andere vorzeichenlos ist. Beide haben eine Kapazität von drei Bits. Die Funktion **Low** liefert den kleinsten Wert, den eine Variable von diesem Typ annehmen kann, die Funktion **High** den Höchsten.

Records

Records sind aus beliebigen Typen in beliebiger Anzahl zusammengesetzte Typen. Ein Record könnte so beispielsweise alle Anschriftendaten über Personen zusammenfassen, geometrische Figuren organisieren oder die Informationen einer `/etc/passwd`-Datenzeile gruppieren.

Arbeiten mit Records

Ein Record wird definiert, indem alle "Record-Felder" aufgeführt werden:

Program: Record1a

```
program Record1a;

type
  TEintrag = String (100);
  TPostleitzahl = String (5);

  TMitarbeiter = record
    PersonalNummer: Integer;
    Vorname, Nachname, Strasse, Stadt: TEintrag;
    PLZ: TPostleitzahl
  end;

var
  Person: TMitarbeiter;

begin
  Person.PersonalNummer := 7;
  Person.Vorname        := 'Hans';
  Person.Nachname       := 'Dampf';
  Person.Strasse        := 'In den Gassen 1-100';
  Person.Stadt          := 'Ueberall';
  Person.PLZ            := '12345'
end.
```

Erklärung

Gefolgt von dem Schlüsselwort **record** werden die einzelnen Felder wie bei der Variablendeklaration aufgeführt. Eine Initialisierung während der Typendefinition ist nicht möglich. Der Record endet mit dem abschließenden **end**. Auf die einzelnen Felder eines Records wird mit Hilfe der Punkt-Schreibweise zugegriffen. So meint **Person.Stadt** das "Stadt-Feld" des Records.

Records können bei der Variablendeklaration initialisiert werden, wie folgendes Beispiel zeigt:

Program: Record2

```
program Record2;

type
  TPunkt = record
    X, Y: Integer
  end;
  TKreis = record
    Mitte: TPunkt;
    Radius: Real
  end;

var
  Ort: TPunkt = (100, 50);
  Kreis: TKreis = ((19, 23), 30.1);

begin
  WriteLn ('Ort = (', Ort.X, ';', Ort.Y, ')');
  WriteLn ('Kreis = (', Kreis.Mitte.X, ';', Kreis.Mitte.Y,') Radius = ',
   Kreis.Radius : 0 : 2)
end.
```

Erklärung

Es werden zwei Record-Typen erzeugt, **TPunkt** und **TKreis**, wobei **TKreis TPunkt** als Feld **Mitte** enthält. Die Variablendeklaration **Ort** initialisiert die zwei Feldvariablen mit Hilfe von **(100, 50)**, sodass **Ort.X = 100** und **Ort.Y = 50** ist.

Kreis hingegen wird so initialisiert, dass **Kreis.Mitte** wie **Ort** initialisiert wird, diesmal allerdings zu **(19, 23)**. Die Feldvariable **Radius** wird mit der Zahl **30.1** initialisiert. Beachten Sie dabei die geschachtelte Klammerung.

Der Zugriff auf die Record-Felder erfolgt wieder in der Punkt-Schreibweise. **Ort.X** ist die X-Koordinate des Punktes, **Kreis.Radius** der Radius des Kreises. Der Mittelpunkt des Kreises ist **Kreis.Mitte**, seine X-Koordinate lautet **Kreis.Mitte.X**.

Ein häufiges Einsatzgebiet von Records ist das Speichern von Datensätzen innerhalb von Arrays. Folgendes Beispiel demonstriert das Vorgehen:

Program: Record3

```
program Record3;

const
  IndexEnde = 10;

type
  TPunkt = record
    X, Y: Integer
  end;

  TPunkte = array [1..IndexEnde] of TPunkt;

var
  Orte: TPunkte;
  Index: Integer;

begin
  for Index := 1 to IndexEnde do
    begin
      Orte[Index].X := Index;
      Orte[Index].Y := Index * 2
    end;
  for Index := 1 to IndexEnde do
    WriteLn ('Orte[', Index, '] = (',
      Orte[Index].X, ';', Orte[Index].Y, ')')
end.
```

Erklärung

Die Deklaration des Arrays erfolgt wie bei elementaren Typen, lediglich die Initialisierung ist verschieden: Auf die einzelnen Record-Felder wird gesondert zugegriffen.

With

Der Zugriff auf die einzelnen Komponenten des Arrays aus Beispiel **Record1** ist schon bei wenigen Feldern sehr mühsam zu schreiben. Jedes Mal musste "**Person.**" vor das gemeinte Feld notiert werden. Eine abkürzende Schreibweise auf Record-Komponenten demonstriert folgendes Beispiel:

Program: Record1b

```
program Record1b;

type
  TEintrag = String (100);
  TPostleitzahl = String (5);

  TMitarbeiter = record
    PersonalNummer: Integer;
    Vorname, Nachname, Strasse, Stadt: TEintrag;
    PLZ: TPostleitzahl
  end;

var
  Person: TMitarbeiter;

begin
  with Person do
    begin
      PersonalNummer := 7;
      Vorname        := 'Hans';
      Nachname       := 'Dampf';
      Strasse        := 'In den Gassen 1-100';
      Stadt          := 'Ueberall';
      PLZ            := '12345'
    end
end.
```

Erklärung

Die abkürzende Schreibweise mit **with** erspart das mehrfache Schreiben von "**Person.**". Trotz dieser Struktur darf innerhalb des **with**-Blockes weiterhin **Person.Nachname** geschrieben werden, wenn es der Umstand erfordert. **With**-Blöcke dürfen beliebig geschachtelt werden, wobei darauf zu achten ist, dass nicht mehrere Record-Variablen die gleichen Feldbezeichner haben. Fehler, die daraus resultieren, sind schwer zu finden.

Variante Records

Erweiterten wir die Menge der geometrischen Objekte aus **Record2** um einige weitere Typen, so würden wir schnell feststellen, dass sie gemeinsame Eigenschaften haben die es nahelegen, alle diese Typen in einem gemeinsamen Record zu vereinigen. Eine Möglichkeit könnte darin bestehen, alle diese Typen einzeln in das Record aufzunehmen, wie folgendes Beispiel zeigt:

Program: Variant1

```
program Variant1;

type
  TObjektTyp = (Punkt, Kreis, Rechteck);

  TPunkt = record
    X, Y: Integer
  end;

  TKreis = record
    X, Y: Integer;
    Radius: Real
  end;

  TRechteck = record
    X, Y, Breite, Hoehe: Integer
  end;

  TGeometrie = record
    Typ: TObjektTyp;
    Punkt: TPunkt;
    Kreis: TKreis;
    Rechteck: TRechteck
  end;

begin
  WriteLn ('Größe TObjektTyp = ', SizeOf (TObjektTyp));
  WriteLn ('Größe TPunkt     = ', SizeOf (TPunkt));
  WriteLn ('Größe TKreis     = ', SizeOf (TKreis));
  WriteLn ('Größe TRechteck  = ', SizeOf (TRechteck));
  WriteLn ('Größe TGeometrie = ', SizeOf (TGeometrie))
end.
```

Erklärung

In diesem Beispiel werden alle Typen in einem Record zusammengefasst. die Funktion **SizeOf** liefert uns die Größe eines Objektes in Bytes. Bei diesem Beispiel stellt sich der Nachteil ein, dass **TGeometrie** die gleiche Größe hat, wie die Summe der einzelnen Typen.

Eine wesentlich elegantere Art, **TGeometrie** zu definieren ist folgende:

Program: Variant2

```
program Variant2;

type
  TObjektTyp = (Punkt, Kreis, Rechteck);

  TGeometrie = record
    X, Y: Integer;     { gemeinsam für alle Geometrie-Typen }
    case Typ: TObjektTyp of
      Kreis:    (Radius: Real);
      Rechteck: (Breite, Hoehe: Integer)
  end;

var
  MeinKreis, MeinPunkt: TGeometrie;

begin
  WriteLn ('Größe TGeometrie = ', SizeOf (TGeometrie));

  with MeinKreis do
    begin
      Typ := Kreis;
      X := 10;
      Y := 10;
      Radius := 3.0
    end;

  with MeinKreis do
    WriteLn ('(', X, ';', Y, ') R= ', Radius);

  with MeinPunkt do
    begin
      Typ := Punkt;
      X := 20;
      Y := 20
    end;

  with MeinPunkt do
    WriteLn ('(', X, ';', Y, ')')
end.
```

Erklärung

Diejenigen Record-Felder, die allen Geometrietypen gemeinsam waren, wurden in der neuen Definition von **TGeometrie** an den Anfang des Records gestellt. Danach folgt eine Auswahl der Record-Felder mittels **case Typ: TObjektTyp of**. Das Record-Feld **Typ** wird tatsächlich zum Bestandteil des Records, diese Art der Darstellung erlaubt es, aus der Menge von Varianten **Kreis** und **Rechteck** beschreibend auszuwählen. Wenn **Typ = Kreis** ist, so steht uns das Record-Feld **Radius** zur Verfügung, ist **Typ = Rechteck**, dann sind **Breite** und **Hoehe** verfügbar.[23]

Der Fall **Typ = Punkt** braucht nicht gesondert abgefangen zu werden, seine Beschreibung liegt außerhalb der Varianten im oberen Bereich des Records. Die Größe des jetzigen **TGeometry**-Typs ist nur noch das Maximum der Größe der Varianten plus der Größe der außerhalb des Varianten-Teils liegenden Record-Felder. Eine Variable vom Typ **TGeometry** benötigt mit dieser Methode ungefähr die Hälfte des Speichers, den eine entsprechende Variable aus **Variant1** belegen würde.

23 Aktuell ist es prinzipiell möglich, alle verfügbaren Record-Felder zu überschreiben, unabhängig davon, wie **Typ** belegt wurde. Hierauf muss bei der Programmierung selbst geachtet werden.

Zeiger

Alle bisher beschriebenen Variablendeklarationen gingen davon aus, dass wir eine zum Zeitpunkt des Programmierens bekannte Anzahl von Datenelementen speichern wollen. Ist die Anzahl der Elemente, die wir speichern wollen aber beliebig und die Typen vielleicht unterschiedlich, so helfen Zeiger dabei, eine dynamische Speicherverwaltung zu erzeugen. Ein Zeiger ist lediglich ein Verweis auf die Speicherstelle einer Variablen.[24] Ein einführendes Beispiel mit Zeigern zeigt folgendes Listing:

Program: Zeiger

```
program Zeiger;

type
  PInteger = ^Integer;

var
  RefZahl: PInteger = nil;

begin
  New (RefZahl);
  RefZahl^ := 42;
  WriteLn ('Inhalt  = ', RefZahl^);
  WriteLn ('Adresse = ', Cardinal (RefZahl));
  Dispose (RefZahl)
end.
```

Erklärung

Ein Zeigertyp wird definiert, indem ein Dach "^" vor den Grundtyp gestellt wird. Das Bedeutet in obigem Fall "Zeiger auf **Integer**". Allgemeine Zeiger auf nicht näher spezifizierte Typen sind **Pointer**.

Deklariert werden Variablen von Zeigertypen genauso wie von allen anderen Typen, neu ist die Initialisierung mit dem Schlüsselwort **nil**. Es sollte hier am Anfang des Kapitels nur der Vollständigkeit aufgeführt werden und bedeutet einen Zeiger auf "nichts", also einen leeren Zeiger.[25] Die Prozedur **New** erzeugt den Speicherplatz, auf dem in Zukunft ein Wert vom Typ **Integer** liegen soll. Dieser Wert kann zugewiesen werden, wobei das Dach diesmal nachgestellt wird. Diese Art des Zugriffs nennt man dereferenzieren. Auf die gleiche Weise kann der Inhalt des Zeigers ausgegeben werden.

Der Zeiger selbst kann mit Hilfe eines Casts ausgegeben werden, wobei hier die Speicheradresse ermittelt[26] wird, an der die Zahl 42 abgelegt wurde.[27]

Dispose gibt den Speicherplatz, der mit **New** erzeugt wurde, wieder frei. Obwohl der Speicher freigegeben wurde, verweist der Zeiger möglicherweise noch an die Position, wo die Zahl **42** abgelegt wurde. Verlassen sollte man sich darauf auf keinen Fall, denn ein freigewordener Speicher kann jeden beliebigen Wert haben, auch den ursprünglichen. Erst beim erneuten Überschreiben der Speicheradresse durch das Betriebssystem ändert sich dieser Wert.

Zeigertypen sollte, so lange nichts dagegen spricht, immer ein "P" vorrangestellt werden.

Ein weitaus komplexeres Beispiel der Speicherverwaltung stellen so genannte Stapel dar. Ein Stapel ist eine Datenstruktur, bei der neu eingefügte Daten über[28] alle anderen Daten gestellt werden. Diejenigen Daten, die zuerst eingegeben wurde sind die Untersten des Stapels. Schaut man sich die einzelnen Elemente des Stapels an, so beginnt man von oben, also in umgekehrter Reihenfolge der Eingabe der Daten. Das folgende Programm liest einen "Stapel Namen" ein, gibt ihn aus und löscht ihn anschließend wieder:

24 So wie die Adresse einer Person nicht die Person selber bedeutet sondern einen Verweis auf sie.
25 Stellen Sie sich ruhig einen Zeiger auf "nichts" wie eine leere, noch nicht ausgefüllte Anschrift vor.
26 An die wir einen Brief schicken könnten mit der Frage nach ihrem Inhalt.
27 Die wir mit Hilfe der verschachtelten Typencasts **Integer (Pointer (Cardinal (RefZahl))^)** wieder erhalten könnten}
28 Wie bei einem Stapel Teller: Der zuletzt auf den Stapel gelegte Teller ist der Oberste.

Program: Stapel1

```
program Stapel1;

type
  TNamenString = String(100);
  PNamen = ^TNamen;
  TNamen = record
    Name: TNamenString;
    Naechster: PNamen
  end;

var
  NamenStapel, TempName: PNamen = nil;
  Abbruch: Boolean = False;
  Name: TNamenString;
  Nummer: Integer;

begin
  Write ('Erzeugt eine Namensliste. ');
  WriteLn ('Abbruch durch leere Zeile');
  { Namensstapel aufbauen }
  repeat
    Write ('Geben Sie einen Namen ein: ');
    ReadLn (Name);
    if Length (Name) = 0 then
      Abbruch := True
    else
      begin
        { neuen Speicherplatz reservieren }
        New (TempName);
        { Namen eintragen }
        TempName^.Name := Name;
        { alten Stapel unter den Neuen legen }
        TempName^.Naechster := NamenStapel;
        {  Alter Stapel ist neuer Stapel }
        NamenStapel := TempName
      end
  until Abbruch;

  TempName := NamenStapel;  { oberstes Stapelelement }
  Nummer := 1;

  { Namensstapel ausgeben }
  while TempName <> nil do
    begin
      WriteLn (Nummer, 'ter Name: ', TempName^.Name);
      Inc (Nummer);
      TempName := TempName^.Naechster
    end;

  { Namensstapel loeschen }
  TempName := NamenStapel;  { oberstes Stapelelement }
  while TempName <> nil do
    begin
      { Namenstapel zeigt auf zweites Element }
      NamenStapel := NamenStapel^.Naechster;
      WriteLn ('entferne ', TempName^.Name);
      Dispose (TempName);      { Speicher wieder freigeben }
      TempName := NamenStapel  { oberstes Stapelelement }
    end
end.
```

Erklärung

Im Typenbereich werden drei Typen deklariert, einer für Strings mit einer festen Größe (**TNamenString**), ein Zeiger auf einen Record (**PNamen**) und der Record selber (**TNamen**). Auffällig ist, dass der Zeigertyp (**PNamen**) vor dem Typen (**TNamen**), dessen Verweis er ist, deklariert wurde. Dies ist erlaubt und an dieser Stelle auch durchaus erwünscht, denn innerhalb der Definition des Records **TNamen** wird dieser Zeigertyp als Typ des Record-Feldes **Naechster** eingesetzt. Verweist ein Record-Feld auf diese Weise auf sich selbst, so spricht man von einer "rekursiven Datenstruktur".

Der Namensstapel wird wie folgt aufgebaut: Wurde ein "richtiger Name" eingegeben, so wird mit Hilfe der Prozedur **New** ein Speicherplatz für **TNamen** erzeugt. Der Name wird dem Record-Feld **Name** zugewiesen. Dabei wird mit Hilfe des Daches der Zeiger dereferenziert, so dass eine Variable vom Typ **TNamen** vorliegt.

Auf die Record-Felder dieser Variablen greift man mit der bekannten "Punkt-Schreibweise" zu. Das Feld **Naechster** wird mit dem Zeiger auf den alten Stapel belegt, wobei dieser "alte Stapel" beim ersten Durchlauf **nil** ist. Der Zeiger **Namensstapel**, welcher soeben noch einen Zeiger auf das erste Element des restlichen Stapels war, wird nun zu einem Zeiger auf das oberste Element des Stapels. Auf diese Weise wird fortwährend der alte Stapel unter das neu erzeugte Element **TempName** gelegt.

Die Daten des Namensstapels werden ausgegeben, indem der Stapel von "oben nach unten" durchlaufen wird. Am Anfang zeigt **TempName** auf das oberste Element des Stapels und wird bei jedem Durchlauf auf seinen Nachfolger gesetzt, die Anzahl der Stapelelemente wird zwecks Anzeige mitprotokolliert. Beachten Sie, dass die Reihenfolge der Ausgabe in umgekehrter Reihenfolge der Eingabe erfolgt. Speicher, der dynamisch alloziert[29] wurde, muss spätestens zum Ende des Programms wieder freigegeben werden. **TempName** verweist zu Beginn auf das oberste Stapelelement. Solange der Stapel noch nicht vollständig abgebaut wurde, wird die Variable **NamenStapel** auf das Element hinter **TempName** gesetzt. Damit enthält **NamenStapel** wieder den "Rest" des Stapels. Der Speicher des obersten Elementes wird mit **Dispose** entfernt. Dieser nun freigewordene Zeiger kann anschließend wieder auf das erste Element des Stapels gesetzt werden und die Schleife kann erneut durchlaufen werden.

Schemata

Schemata dienen ebenso wie Zeiger dazu, Speicher dynamisch zu verwalten. Im Gegensatz zu Zeigern muss die maximale Anzahl der Elemente, die gespeichert werden sollen, bekannt sein, wobei diese Anzahl erst zur Laufzeit feststehen muss. Einen Schematypen kennen Sie bereits, es handelt sich dabei um **String**s. Schemata und Zeiger lassen sich auch kombinieren. Das einführende Beispiel zeigt den grundsätzlichen Umgang mit Schemata:

Program: Schema1

```
program Schema1;

type
  TSchemaType (Anz: Byte) = array [1..Anz] of Integer;

var
  MeinSchema: TSchemaType (4);
  i: Integer;

begin
  WriteLn ('MeinSchema hat maximal ', MeinSchema.Anz, ' Elemente.');
  for i := 1 to MeinSchema.Anz do
    MeinSchema[i] := 5 * i;
  for i := 1 to MeinSchema.Anz do
    WriteLn (MeinSchema[i])
end.
```

29 So nennt man das Zuteilen von Speicher, zum Beispiel mit der Prozedur **New**.

Erklärung

Schematypen werden definiert, indem mindestens eine Variable, in unserem Beispiel **Anz**, dem Typenname folgt. Diese Variable, die auch als Diskiminante bezeichnet wird, wird als eine der Grenzen eines Arrays[30] benutzt. Das Deklarieren eines Schemas erfolgt analog zum Deklarieren eines **String**s.

Die maximale Anzahl der Elemente wird angegeben, was in unserem Fall bedeutet, dass **MeinSchema** ein Array von vier **Integer**-Variablen ist. Die Anzahl der Elemente lässt sich nachträglich während des Programmlaufes herausfinden, indem das zum Schema passende Feld **Anz** in der bekannten Weise ausgelesen wird. Auf die Variable **MeinSchema** kann zugegriffen werden, wie auf ein Array.

Schemata dienen der Bequemlichkeit. Benötigt man mehrere Variablen vom gleichen Array-Typ, jedoch mit möglicherweise unterschiedlicher Anzahl von Variablen, so sind sie genau die richtige Wahl.

mehrdimensionale Arrays zu erzeugen oder den Anfang und das Ende des Arrays festzulegen, wie folgendes Beispiel zeigt:

Program: Schema2

```
program Schema2;

type
  TSchemaType (Anfang, Ende: Cardinal) =
    array [Anfang..Ende] of Integer;

var
  MeinSchema: TSchemaType (1, 4) = (5, 6, 7, 8);
  i: Integer;

begin
  for i := MeinSchema.Anfang to MeinSchema.Ende do
    WriteLn (MeinSchema[i])
end.
```

Erklärung

In diesem Beispiel werden mit Hilfe der beiden Diskriminanten **Anfang** und **Ende** die Grenzen des Arrays festgelegt. Diese Grenzen müssen bei der Deklaration der Variablen angegeben werden. Wie auf diese Diskriminanten zugegriffen wird, zeigt die **for**-Schleife.

30 Schemata müssen nicht, so wie in unseren Beispielen, auf Arrays basieren, aber es ist vermutlich das häufigste Einsatzgebiet.

Schemata können mit Zeigern kombiniert werden und zur Laufzeit dynamisch erzeugt werden. Das folgende Programm ist eine Abwandlung von **Schema2** welches diese Fähigkeit demonstriert:

Program: Schema3

```
program Schema3;

type
  PSchemaType = ^TSchemaType;
  TSchemaType (Anfang, Ende: Cardinal) = array [Anfang..Ende] of Integer;

var
  MeinSchema: PSchemaType;
  i: Integer;

begin
  New (MeinSchema, 4, 6);
  for i := MeinSchema^.Anfang to MeinSchema^.Ende do
    MeinSchema^[i] :=  10 * i;
  for i := MeinSchema^.Anfang to MeinSchema^.Ende do
    WriteLn (MeinSchema^[i]);
  Dispose (MeinSchema)
end.
```

Erklärung

Es wird ein Zeiger auf den Schematyp definiert, der genutzt wird, um **MeinSchema** zu deklarieren. Der Speicher für **MeinSchema** wird dynamisch angefordert, wobei der Prozedur **New** beide Grenzen des zugehörigen Arrays übergeben werden. Auf **Anfang** und **Ende** wird mit der bekannten Zeiger-Schreibweise zugegriffen, als sei **MeinSchema** ein Record[31] Das Array erhält man, indem der Schemazeiger dereferenziert wird. Auf die einzelnen Variablen kann sodann mit der Index-Schreibweise zugegriffen werden.

31 Tatsächlich sind Schemata wie Records implementiert.

Routinen

Routinen dienen dazu, wiederkehrende Aufgaben zu gruppieren und den Quelltext übersichtlich und wartungsfreundlich zu gestalten. Es ist einfacher, an genau einer Stelle einen bestimmten Code zu ändern als an vielen Stellen immer wieder den gleichen Quelltext zu bearbeiten, was schnell zu Fehlern führt. Es gibt drei verschiedene Sorten von Routinen: Prozeduren, Funktionen und Operatoren.[32]

Routinen können ihre eignen, nur für diese Routine geltenden Konstanten, Typen und Variablen definieren und deklarieren. In diesem Fall spricht man von lokalen Konstanten, lokalen Typen und lokalen Variablen. Variablen, auf die alle Teile des Programms zugreifen können nennt man im Gegensatz dazu "global".

Prozeduren

Prozeduren wurden in den vergangen Kapitel benutzt, ohne dass über ihr Wesen gesprochen wurde. Diese Prozeduren waren **WriteLn**, **Inc**, **New** und viele andere. Prozeduren sind eigenständige Anweisungsfolgen, wobei sie durchaus globale Variablen verändern können. Das Beispiel aus Kapitel Stapel1 sieht umgeschrieben mit Prozeduren aus wie folgt:

Programm: Stapel2

```
program Stapel2;

type
  TNamenString = String(100);
  PNamen = ^TNamen;
  TNamen = record
    Name: TNamenString;
    Naechster: PNamen
  end;

var
  NamenStapel: PNamen = nil;

procedure Stapeln;
var
  Abbruch: Boolean = False;
  Name: TNamenString;
  TempName: PNamen = nil;

begin
  repeat
    Write ('Geben Sie einen Namen ein: ');
    ReadLn (Name);
    if Length (Name) = 0 then
      Abbruch := True
    else
      begin
        New (TempName);
        TempName^.Name := Name;
        TempName^.Naechster := NamenStapel;
        NamenStapel := TempName
      end
  until Abbruch
end;
```

32 Eigentlich gehören Operatoren nicht in diese Klasse von Sprachelementen, sondern bilden eine Eigene. Sie werden hier aufgeführt, weil sie eine Reihe von Gemeinsamkeiten mit Funktionen und Prozeduren haben.

```
(Fortsetzung)

procedure StapelAusgeben;
var
  Nummer: Integer;
  TempName: PNamen = nil;

begin
  TempName := NamenStapel;
  Nummer := 1;
  while TempName <> nil do
    begin
      WriteLn (Nummer, 'ter Name: ', TempName^.Name);
      Inc (Nummer);
      TempName := TempName^.Naechster
    end
end;

procedure StapelLoeschen;
var
  TempName: PNamen = nil;

begin
  TempName := NamenStapel;
  while TempName <> nil do
    begin
      NamenStapel := NamenStapel^.Naechster;
      WriteLn ('entferne ', TempName^.Name);
      Dispose (TempName);
      TempName := NamenStapel
    end
end;

begin
  WriteLn ('Erzeugt eine Namensliste. Abbruch durch leere Zeile.');
  Stapeln;
  StapelAusgeben;
  StapelLoeschen;
end.
```

Erklärung

Der Datentyp ist geblieben wie im Beispiel von Kapitel Stapel1 . Die Anzahl der globalen Variablen hat sich drastisch reduziert, übriggeblieben ist die Variable **NamenStapel**, da sie sozusagen den gesamten Stapel festhält.

In diesem Beispiel werden drei Prozeduren deklariert, **Stapeln**, **StapelAusgeben** und **StapelLoeschen**. Allen Prozeduren ist gemeinsam, dass sie über mindestens eine lokale Variable verfügen, die nur innerhalb dieser Prozedur Gültigkeit hat. Ähnlich wie bei Programmen werden die Anweisungen, welche die Prozedur ausführt, in einem Block zwischen **begin** und **end** gruppiert. Den Inhalt der Prozeduren kennen sie bereits aus einem früheren Beispiel. Durch den Einsatz von Prozeduren verringert sich die Zeilenzahl des Hauptprogramms drastisch, in unserem Beispiel wird die Zeilenzahl bei vollem Funktionsumfang auf Drei reduziert. Sollte der Stapel ein zweites Mal ausgeben werden, so müsste lediglich eine weiter Zeile zum Hauptprogramm hinzugefügt werden.

Prozeduren können eine Argumentenliste haben, denn sonst ließen sich Routinen wie **WriteLn** und **New** nicht realisieren:

Programm: Proc

```
program Proc;

procedure Summe (Von, Bis: Integer; Zwischensumme: Boolean);
var
  i, Summe: Integer = 0;
begin
  for i := Von to Bis do
    begin
      Summe := Summe + i;
      if Zwischensumme then
        WriteLn ('Zwischensumme = ', Summe)
    end;
  WriteLn ('Summe = ', Summe)
end;

begin
  Summe (-2, 3, True)
end.
```

Erklärung

Dieses Programm berechnet die Summe zwischen **Von** und **Bis**, wobei auf Wunsch Zwischenergebnisse ausgegeben werde. Die Parameterliste einer Prozedur kann beliebig lang sein oder aber ganz weggelassen werden. Parameter einer Prozedur werden analog zu einer Variablendeklaration aufgezählt, wobei das Schlüsselwort **var** entfällt, da es in Parameterlisten eine besondere Bedeutung hat, auf die wir im Abschnitt Call by Reference zu sprechen kommen.

Funktionen

Funktionen unterscheiden sich dadurch von Prozeduren, dass sie nie alleine auftreten, sondern immer einen Teil eines Ausdrucks bilden. Der Grund dafür liegt darin, dass Funktionen immer einen Rückgabewert haben. Ansonsten gilt alles, was über Prozeduren geschrieben wurde auch hier. Einige Funktionen sind bereits bekannt: **Card**, **Ord**, **Length** und weitere.

Funktionen werden ähnlich wie Prozeduren deklariert:

Programm: Funk

```
program Funk;

var
  MeineSumme: Integer;

function Summe (Von, Bis: Integer): Integer;
var
  i, ZwSumme: Integer Value 0;
begin
  for i := Von to Bis do
    ZwSumme := ZwSumme + i;
  Summe := ZwSumme
end;

begin
  MeineSumme := Summe (1, 100);
  WriteLn ('Die Summe der ersten 100 Zahlen ist ', MeineSumme)
end.
```

Erklärung

Funktionen werden deklariert, indem das Schlüsselwort **function** vor den Bezeichner geschrieben wird. Darauf folgt eine optionale Parameterliste und der Typ, den diese Funktion zurückliefert. In obigem Beispiel liefert die Funktionen einen **Integer**-Wert zurück, der gleich der Summe ist. Innerhalb der Funktion entspricht dies einer Variablen mit dem Funktionsnamen (**Summe**), die den Typ des Rückgabewertes hat.

Forward-Deklarationen

Benötigt eine Routine eine Andere, die jedoch erst zu einem späteren Zeitpunkt innerhalb des Quelltextes deklariert wird, so wird das Übersetzen des Quelltextes fehlschlagen. Eine Lösung besteht darin, die Reihenfolge aller deklarierten Routinen innerhalb des Quelltextes zu verändern. Oft möchte man dies nicht, da gerade diese Reihenfolge eine besondere Lesbarkeit[33] gewährleistet. Ein Grund dafür könnte sein, dass alle Routinen alphabetisch oder thematisch sortiert wurden. Forward-Deklarationen dienen dazu, die Reihenfolge beizubehalten und die gegenseitigen Abhängigkeiten aufzulösen. Das folgende Beispiel demonstriert den Mechanismus:

33 Eine implizite Forward-Deklaration kennen Sie schon aus dem Kapitel Stapel1. Dort wurde PNamen vor TNamen definiert.

Programm: ForW

```
program ForW;

procedure SchreibeA;
begin
  WriteLn ('A')
end;

procedure SchreibeB; forward;
procedure SchreibeAB;
begin
  SchreibeA;
  SchreibeB
end;

procedure SchreibeB;
begin
  WriteLn ('B')
end;

begin
  SchreibeAB
end.
```

Erklärung

In diesem Beispiel wurden drei alphabetisch sortierte Prozeduren deklariert, wobei **SchreibeAB** die Prozedur **SchreibeB** aufruft, von der sie noch keine Kenntnis haben kann. Aus diesem Grund wurde der Prozedurkopf von **SchreibeB** deklariert, indem die Direktive **forward** nachgestellt wurde.

Call by Reference

Bei manchen Prozeduren ist es sinnvoll, wenn sie einen Rückgabewert haben. Eine solche Prozedur kennen Sie bereits, es ist **Inc**. Diese Prozedur verändert den ihr übergebenen Wert dahingehend, dass sie diesen um Eins erhöht. Weitere Einsatzgebiete für diese Technik, die im folgenden Abschnitt vorgestellt wird, sind Routinen, die mehr als einen Wert zurückliefern sollen. Diese Art, Parameter von Routinen zu deklarieren nennt man "Call By Reference".

Zuerst wird eine Prozedur implementiert, die ähnlich wie **Inc** arbeitet:

Programm: CBR1

```
program CBR1;

var
  MeineZahl: Integer;

procedure PlusDrei (var Zahl: Integer);
begin
  Zahl := Zahl + 3
end;

begin
  MeineZahl := 10;
  WriteLn ('MeineZahl = ', MeineZahl);
  PlusDrei (MeineZahl);
  WriteLn ('MeineZahl = ', MeineZahl)
end.
```

Erklärung

Das zusätzliche Schlüsselwort **var** bewirkt, dass das Original des übergebenen Parameters verändert wird. Allen bisherigen Beispielen aus dem Bereich Funktionen und Prozeduren war gemeinsam, dass als Parameter nur Kopien der Argumente übergeben wurde. Das ist bei den vorgestellten Beispielen nicht aufgefallen, da nie die Notwendigkeit bestand, das Argument selbst zu ändern. Ein anderes Beispiel ist eine Funktion, die zwei Argumente tauscht:

Programm: CBR2

```
program CBR2;

var
  Zahl1, Zahl2: Integer;

procedure Zahlentauschen (var Param1, Param2: Integer);
var
  Hilfsvariable: Integer;
begin
  Hilfsvariable := Param1;
  Param1 := Param2;
  Param2 := Hilfsvariable
end;

begin
  Zahl1 := 10;
  Zahl2 := 20;
  WriteLn ('Vorher : Zahl1 = ', Zahl1, ' Zahl2 = ', Zahl2);
  Zahlentauschen (Zahl1, Zahl2);
  WriteLn ('Nachher: Zahl1 = ', Zahl1, ' Zahl2 = ', Zahl2)
end.
```

Erklärung

Die Prozedur **Zahlentauschen** vertauscht die Werte der ihr übergebenen Argumente. Ohne das Schlüsselwort **var** würde sich an den Variablen **Zahl1** und **Zahl2** nichts ändern. Gleichzeitig ist diese Prozedur ein Beispiel für eine Routine, die mehr als einen Rückgabewert hat.

Statische Variablen

Statisch deklarierte Variablen dienen dazu, die Dauer der Gültigkeit dieser Variablen zu verlängern. In bisherigen Fällen verloren die lokalen Variablen ihre Gültigkeit, sobald die Routine abgearbeitet war. Hierzu ein Beispiel:

Programm: Statisch

```
program Statisch;

procedure SchreibeZahl;
var
  Zahl: Integer = 0; attribute (static);
begin
  WriteLn ('Zahl ist jetzt = ', Zahl);
  Inc (Zahl)
end;

begin
  SchreibeZahl;
  SchreibeZahl;
  SchreibeZahl
end.
```

Erklärung

Trotz des dreifachen Aufrufs von **SchreibeZahl** wird die statische Variable nur einmal deklariert und initialisiert. Bei jedem Aufruf wird die lokale Variable **Zahl** um Eins erhöht, beim dritten Aufruf der Prozedur **SchreibeZahl** ist der Wert bereits auf 2 angewachsen. Weitere Attribute werden in späteren Abschnitten erläutert.

Rekursion

Rekursive Routinen sind solche, die sich selber aufrufen. Üblicherweise werden sie aus Gründen der Eleganz als Funktionen implementiert. Diese Technik gestaltet Quellcode besonders übersichtlich, weil generell weniger Zeilen zur Lösung eines Problems verwendet werden. Hierzu ein Beispiel, wobei die rekursive Funktion die Summe von Zahlen berechnet:

Programm: Rekurs

```
program Rekurs;

var
  DieSumme: Integer;

function Summe (Von, Bis: Integer): Integer;
begin
  if Von = Bis then
    Summe := Von
  else
    Summe := Von + Summe (Von + 1, Bis)
end;

begin
  DieSumme := Summe (1, 100);
  WriteLn ('Das Ergebnis lautet: ', DieSumme)
end.
```

Erklärung

Die gesamte Funktion **Summe** besteht aus einer **if**-Anweisung mit vier Zeilen. Es ist im Gegensatz zu vorherigen Implementationen dieser Funktion keine lokale Variable nötig. Der Algorithmus funktioniert so: Eine Summe von 1 bis 100 ist gleich 1 plus der Summe aus den Zahlen 2 bis 100 und das ist gleich der Summe der Zahlen 1 plus 2 plus der Summe der verbleibenden Zahlen und das ist gleich 1 + 2 + 3 plus der Summe der verbleibenden Zahlen und so fort. Wenn die letzte Zahl, in unserem Beispiel 100 erreicht ist und damit **Von** und **Bis** übereinstimmen, so wird die Summe von hinten tatsächlich berechnet, also 1 + 2 + 3 + ... + 98 + 99 + 100 = 1 + 2 + 3 + ... + 98 + 199 = 1 + 2 + 3 + ... + 297 usw. Die Tatsache, dass eine Funktion sich selbst aufrufen kann mag ungewöhnlich erscheinen, ist aber tatsächlich ein sehr mächtiges Instrument um geeignete Problemlösungen elegant zu formulieren. Alle mit Rekursion lösbaren Probleme lassen sich auch auf die herkömmliche Weise lösen.

Funktionsergebnisse ignorieren

In einigen Fällen ist es sinnvoll, den Erfolg einer Operation mitgeteilt zu bekommen. Im Falle einer Division zum Beispiel kann so bemerkt werden, dass der Nenner Null ist und das Ergebnis kann anschliessend verworfen werden:

Programm: Ignore1

```
program Ignore1;

var
  Geteilt: Real;
  Erfolg: Boolean;

function Division (Zaehler, Nenner: Integer; var Ergebnis: Real): Boolean;
begin
  if Nenner = 0 then
    Division := False
  else
    begin
      Ergebnis := Zaehler / Nenner;
      Division := True
    end
end;

begin
  Erfolg := Division (4, 0, Geteilt);
  WriteLn (Erfolg)
end.
```

Falls ein Programm auf der Grundlage einer Vorbedingung die Null als Nenner ausschließt, kann es interessant sein, sich die dann überflüssige Variable **Erfolg** zu sparen.

Damit in diesem Fall der Compiler bei der Übersetzung nicht warnt, dass der Funktionswert nicht zugewiesen wurde, hilft folgende Konstruktion:

Programm: Ignore2

```
program Ignore2;

var
  Z: Integer;
  Geteilt: Real;

function Division (Zaehler, Nenner: Integer;
  var Ergebnis: Real): Boolean; attribute (ignorable);
begin
  if Nenner = 0 then
    Division := False
  else
    begin
      Ergebnis := Zaehler / Nenner;
      Division := True
    end
end;

begin
  for Z := 1 to 10 do
    begin
      Division (Z, 3, Geteilt);
      WriteLn (Z, '/3 = ', Geteilt : 0 : 5)
    end
end.
```

Erklärung

Das Attribut **ignorable** schaltet die Warnung des Compilers, die sich auf das fehlende Zuweisen des Funktionsergebnisses bezieht, aus. Dies ist in obigem Programm sinnvoll, da wir davon ausgehen können, dass der Nenner niemals den Wert Null annehmen kann. Darüberhinaus können wir so zum Zeitpunkt des Programmierens bestimmen, ob wir die Routine lieber als Funktion oder als Prozedur benutzen wollen.

Operatoren

Die folgenden Operatoren sind in GNU-Pascal enthalten[34], wobei die hier gezeigte Reihenfolge die Rangfolge wiederspiegelt:

```
<,    =,    >,    in,
<>,   >=,   <=
+,    -,    or
*,    /,    div,  Mod,
and,  shl,  shr,  xor
pow,  **,   ><
not,  @
```

34 Darüber hinaus existieren noch die Operatoren der PXSC-Pascal Erweiterung: +<, +>, -<, ->, *<, *>, /< und />.

Einigen dieser Operatoren lässt sich eine neue Bedeutung geben. Das folgende Beispiel definiert den +-Operator für Integer-Zahlen zu einem --Operator um:

Programm: NeuPlus

```
program NeuPlus;

operator + (A, B: Integer) R: Integer;
begin
  R := A - B
end;

begin
  WriteLn ('10 + 7 = ', 10 + 7)
end.
```

Erklärung

Das Schlüsselwort **operator** leitet die Definition eines Operators ein. Statt eines Bezeichners wird das Symbol eines Operators benutzt, der umdefiniert werden soll. Das Ergebnis der Operation wird wie eine Variable deklariert, die im Körper der Funktion genutzt werden kann.

Es können alle Operatoren umdefiniert werden, die über genau zwei Operanden verfügen. Aus obiger Aufzählung fallen das unäre Minus (z.B. **Wert := -4;**) wie auch das unäre Plus heraus, **not** und **@** ebenso. Parameterlisten von Operatoren dürfen das Schlüsselwort **var** enthalten, so dass die Operanden auf diese Weise einen Wert zurückgeben können. Des weiteren können innerhalb von Operatoren eigene Typen, Konstanten und Variablen definiert und deklariert werden. Es lassen sich keine zusätzlichen Operatoren definieren die nicht in obiger Liste und zugehöriger Fußnote sind. Wohl aber lassen sich Operatoren mit Hilfe eines Bezeichners definieren, z. B. **operator Plus (A, B: Integer) = R: Integer;**. Anwendungsbereiche für Operatoren sind Operationen auf selbst definierte Strukturen wie z. B. Vektoren. Auch lässt sich ein Operator definieren, der zu einem Stapel ein Element hinzufügt.

Programmierbeispiel: Logic-Spiel

Das folgende Programm ist eine vereinfachte Version des beliebten Spieles "Logic", welches mittlerweile auf jedem Mobiltelefon verfügbar ist. Das Ziel des Spieles ist es, eine bestimmte Folge von Symbolen, in unserem Beispiel Buchstaben, zu erraten, wobei das Programm lediglich Informationen darüber ausgibt, wie viele Buchstaben an der richtigen Position erraten wurden und wie viele Buchstaben an der falschen Stelle übereinstimmen.

Programm: Logical

```
program Logical;

const
  AnzZeichen = 4;  { max. Anz. der versch. Zeichen im String }
  StrLaenge  = 4;  { Länge des Strings }
  ErstesZeichen = 'a';
  UnbenutztesZeichen = '.';

type
  TRateString = String (StrLaenge);

var
  RateString, GeheimString: TRateString;
  Versuche, RichtigeZeichen, RichtigeStelle: Integer = 0;
  Gefunden: Boolean = False;
```

(*Fortsetzung)*

```
function InitGeheimString: TRateString;
var
  i: Integer;
  TmpString: TRateString;
begin
  for i := 0 to StrLaenge do
    TmpString[i] := Chr (Random (AnzZeichen) +
      Ord (ErstesZeichen));
  InitGeheimString := TmpString
end;

function AnzRichtigeStelle (Geheim, Rate: TRateString): Integer;
var
  Richtig, i: Integer = 0;
begin
  for i := 1 to StrLaenge do
    if Geheim[i] = Rate[i] then
      Inc (Richtig);
  AnzRichtigeStelle := Richtig
end;

function AnzRichtige (Geheim, Rate: TRateString): Integer;
var
  i, j, Richtig: Integer = 0;
  TmpRate: TRateString;
begin
  TmpRate := Rate;
  for i := 1 to StrLaenge do
    for j := 0 to StrLaenge do
      if Geheim[i] = TmpRate[j] then
        begin
          Inc (Richtig);
          TmpRate[j] := UnbenutztesZeichen;
          Break
        end;
  AnzRichtige := Richtig
end;

begin
  GeheimString := InitGeheimString;
  repeat
    Write ('Geben Sie ', StrLaenge, ' Zeichen ein von ',
      ErstesZeichen, ' bis ',
      Chr (Ord (ErstesZeichen) + AnzZeichen - 1), ' :');
    ReadLn (RateString);
    if Length (RateString) <> StrLaenge then
      Continue;
    Inc (Versuche);
    if RateString = GeheimString then
      Gefunden := True
    else
      begin
        RichtigeStelle  :=
          AnzRichtigeStelle (GeheimString, RateString);
        RichtigeZeichen :=
          AnzRichtige (GeheimString, RateString);
        WriteLn (RichtigeStelle, ' an der richtigen Stelle. ',
          RichtigeZeichen - RichtigeStelle, ' richtig.')
      end
  until Gefunden;
  WriteLn ('Gefunden nach ', Versuche, ' Versuchen.')
end.
```

Erklärung

Die Funktion **InitGeheimString** erzeugt einen Zufallsstring, wobei jeder Buchstabe aus der Menge der ersten **AnzZeichen** Kleinbuchstaben ist. In unserem Beispiel **'a'..'d'**. **AnzRichtigeStelle** bestimmt bei zwei Strings die Anzahl der übereinstimmenden Buchstaben, wobei die Stellung mitberücksichtigt wird. **AnzRichtige** überprüft die Anzahl der insgesamt übereinstimmenden Buchstaben, unabhängig von der Stelle. Dabei wird, sobald ein Buchstabe übereinstimmt, diese Stelle in ein unbenutztes Zeichen verwandelt, damit doppelt übereinstimmende Buchstaben nicht gezählt werden können. Das Programm selbst liest einen String ein. Wenn dieser String identisch mit dem zu ratenden ist, so ist das Programm beendet. Wenn nicht, so werden zuerst die in der Stellung übereinstimmenden Buchstaben gezählt (**RichtigeStelle**).

Anschließend werden die insgesamt übereinstimmenden Buchstaben gezählt, wobei darin auch solche Übereinstimmungen gezählt werden, die schon mit **AnzRichtigeStelle** berücksichtigt wurden. **RichtigeZeichen - RichtigeStelle** ergibt somit die Anzahl der Buchstaben, die an der falschen Stelle übereinstimmen.

Units

Units dienen dazu, Quelltexte zu modularisieren und wiederverwertbare Konstanten, Typen und Routinen zu gruppieren. Sie ersparen es, in alten Programmen nach bestimmten Routinen zu suchen oder Ideen doppelt zu entwickeln. Solche Units, auch Module[35] genannt, werden mit der **uses**-Anweisung eingebunden. Werden mehrere Units eingebunden, so werden die einzelnen Module durch Kommas getrennt angegeben.

Der Aufruf des Compilers zum Erzeugen des Programms lautet: `gpc --automake dateiname.pas -o programmname`

Die Option `--automake` sorgt dafür, dass alle benötigten Module eingebunden werden.

Die Standardunit GPC

Es existieren eine Reihe von Standardunits, die mit dem GNU-Pascal Compiler mitgeliefert werden. Eine von ihnen ist die Unit `GPC`, die im Folgenden in Auszügen präsentiert wird. `GPC` enthält viele nützlicher Routinen, darunter auch solche zum Thema " Datum und Zeit":

Programm: Zeit

```
program Zeit;

uses GPC;

type
  TTagesString = String (10);

var
  Tag, Monat, Jahr: Integer;
  Time: TimeStamp;
  NullString: array [Boolean] of String (1) = (, '0');
  KeinString: array [Boolean] of String (4) = ('kein', 'ein');

function TagName (t, m, j: Integer): TTagesString;
var
  TagNummer: Integer;
  TagNamen: array [0..6] of String (10) = (
    'Sonntag',    'Montag',  'Dienstag', 'Mittwoch',
    'Donnerstag', 'Freitag', 'Samstag');
```

35 Streng genommen ist ein Modul eine weitere Möglichkeit zur Modularisierung, die in dieser Einführung nicht behandelt wird.

(Fortsetzung)

```
begin
  TagNummer := GetDayOfWeek (t, m, j);
  TagName   := TagNamen[TagNummer]
end;

begin
  GetTimeStamp (Time);
  with Time do
    begin
      Tag   := Day;
      Monat := Month;
      Jahr  := Year
    end;
  WriteLn (TagName (Tag, Monat, Jahr),' ',
    NullString[Tag < 10], Tag, '.',
    NullString[Monat < 10], Monat, '.', Jahr);
  WriteLn ('Dieses Jahr ist ', KeinString[IsLeapYear (Jahr)],
    ' Schaltjahr.');
  WriteLn ('Es ist der ', GetDayOfYear (Tag, Monat, Jahr),
    '. Tag des Jahres.')
end.
```

Erklärung

Dieses Programm gibt den Tagnamen gefolgt vom Datum aus. Dazu kommen Informationen darüber, ob das gegenwärtige Jahr ein Schaltjahr ist und welcher Tag des Jahres gerade ist. Die drei Funktionen **GetDayOfWeek**, **IsLeapYear** und **GetDayOfYear** sind Bestandteil der Unit `GPC`.

Die Funktion **TagName** liefert uns den Namen des Tages. Da die Funktion **GetDayOfWeek** " Sonntag" als nullten Tag ansieht, wurde das Array **TagNamen** so gestaltet, wie es ist. **GetTimeStamp** liefert uns Informationen über das aktuelle Datum im Record **Time** zurück. Drei der Felder sind **Day**, **Month** und **Year**. **NullString** dient dazu, bei Werten kleiner als 10 eine führende Null auszugeben. **IsLeapYear** ist **True**, wenn das angegebene Jahr ein Schaltjahr ist. **KeinString[True]** hat den Wert " ein". **GetDayOfYear** gibt die aktuelle Tagesnummer des Jahres zurück.

Eine weitere Gruppe von Funktionen innerhalb der `GPC`-Unit beschäftigt sich mit dem Thema Kommandozeilenoptionen. Kommandozeilenoptionen sind Parameter, die einem Programms auf der Kommandozeile mitgegeben werden können. Die Option `-o` des GNU-Pascal Compilers ist ein solcher Parameter.

Programm: Kommandozeile

```
program Kommandozeile;

uses GPC;

var
  Optionen: array [1..3] of OptionType = (
    ('date',      NoArgument,        nil, 'd'),
    ('leapyear',  RequiredArgument,  nil, 'l'),
    ('time',      OptionalArgument,  nil, 't'));
  Index: Integer;
  Ch: Char;

procedure DatumAusgeben;
var
  Time: TimeStamp;
begin
  GetTimeStamp (Time);
  with Time do
    WriteLn (Day, '.', Month, '.', Year)
end;

procedure SchaltjahrAusgeben (Jahr: Integer);
begin
  if IsLeapYear (Jahr) then
    WriteLn (Jahr, ' ist ein Schaltjahr.')
  else
    WriteLn (Jahr, ' ist kein Schaltjahr.')
end;

procedure ZeitAusgeben (Arg: TString);
var
  Argument: TString = Arg;
  Time: TimeStamp;
begin
  GetTimeStamp (Time);
  if Argument <>  then
    begin
      LoCaseString (Argument);
      if Argument <> 'jetzt' then
        WriteLn ('Option für --time: jetzt!')
      else
        with Time do
          WriteLn (Hour, ':', Minute, ':', Second)
    end
  else
    with Time do
      WriteLn (Hour, ':', Minute, ':', Second)
end;

procedure ParseArgumente (Opt: Char; Arg: TString);
var
  Zahl, Fehler: Integer;
begin
  case Opt of
    'd':  DatumAusgeben;
```

(Fortsetzung)

```
 'l':  begin
            Val (Arg, Zahl, Fehler);
            if Fehler = 0 then
              SchaltjahrAusgeben (Zahl)
            else
              WriteLn ('Fehlerhafte Option von --leapyear: ', Arg)
          end;
    't':  ZeitAusgeben (Arg)
  end
end;

begin
  repeat
    Ch := GetOptLong ('+-', Optionen, Index, False);
    case Ch of
      EndOfOptions: { Ende }
      otherwise     ParseArgumente (Ch, OptionArgument)
    end;
  until Ch = EndOfOptions
end.
```

Erklärung

Mit **Optionen** definieren wir ein Array auf **OptionType**-Argumente, welches die Optionen `date`, `leapyear` und `time` mit den dazugehörigen kurzen Schreibweisen `d`, `l` und `t` auflistet. Es wird beim Aufruf des Programms möglich sein, den Shellbefehl

```
kommandozeile --date
```

oder

```
komandozeile -l2002
```

auszuführen. Der Option `date` darf kein Argument mitgegeben werden, `leapyear` benötigt das Jahr als Argument und `time` darf ein optionales Argument "jetzt" haben. Die Prozeduren **DatumAusgeben**, **SchaltjahrAusgeben** und **ZeitAusgeben** tun genau das, was ihr Name vermuten lässt.

Im Hauptprogramm sorgt die Funktion **GetOptLong** dafür, dass die dem Programm übergebenen Optionen entgegen genommen werden. Gibt diese Funktion den Wert der vordefinierten Konstanten **EndOfOptions** zurück, so wird die dazugehörige **repeat...until**-Schleife beendet. Der erste Parameter bedeutet, dass keine Kommandozeilenoptionen gelesen werden, die nicht im Array **Optionen** enthalten sind. Das Abschließende **-** innerhalb des Parameters sorgt dafür, dass nur die kurzen Optionen übergeben werden, auch wenn eine lange Option auf der Kommandozeile angegeben wurde. Der Parameter **False** verhindert, dass nur lange Optionen berücksichtigt werden. Dieses Vorgehen ist insgesamt sehr praktisch, da wir auf diese Weise kurze und lange Optionen nicht getrennt auswerten müssen.

ParseArgumente übernimmt die ausgewählte kurze Option und den Wert einer globalen Variablen **OptionArgument**, welche je nach Option ein zusätzliches Argument der Option beinhaltet. Im Fall von `-l` zum Beispiel das Jahr. Der Prozedurkörper dieser Routine hat nur noch die Aufgabe, die entsprechenden Aufrufe der einzelnen Programmteile vorzubereiten.

Die Standardunit CRT

Diese Unit enthält eine Sammlung von Routinen, die das Schreiben von Textmodus-Anwendungen erleichtern. So enthält sie Prozeduren, um farbige Fenster und Töne zu erzeugen. Manche der Routinen, so auch solche zum Erzeugen von Tönen, sind nicht auf jeder Plattform vorhanden, insbesondere nicht unter X11. Sie sollten solche Programme wirklich nur im Textmodus benutzen.

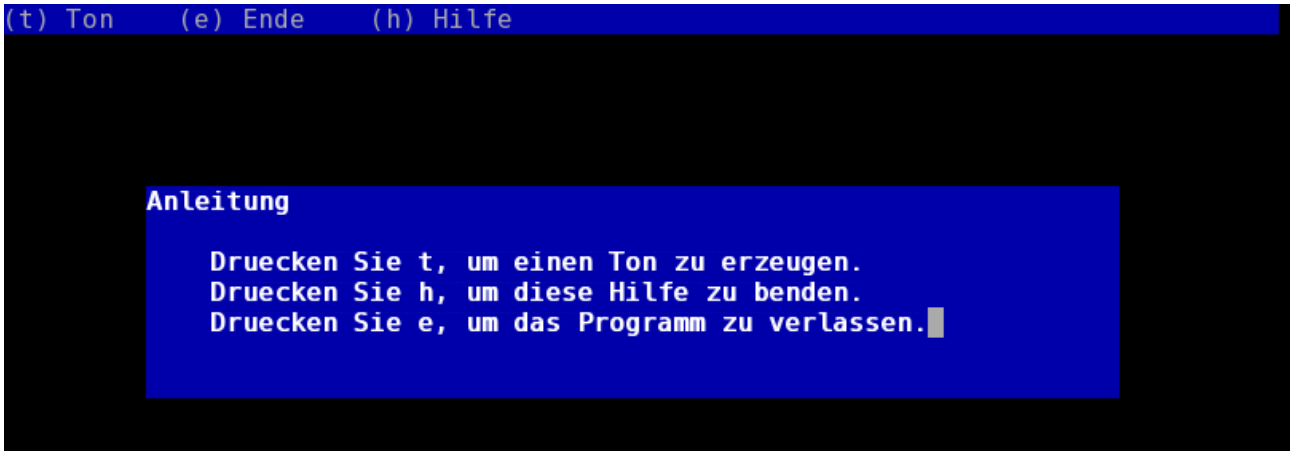
Programm: CrtTest

```
program CrtTest;

uses CRT;

var
  Ende: Boolean = False;

procedure Ton;
begin
  { Nicht unter X11! }
  Sound (440);
  Delay (100);
  NoSound
end;

procedure Hilfe;
var
  HilfeSichtbar: Boolean = False; attribute (static)
begin
  if HilfeSichtbar then
    begin
      TextBackground (Black);
      TextColor (Black);
      ClrScr
    end
  else
    begin
      TextBackground (Blue);
      TextColor (White);
      Window (10, 7, 70, 13);
      ClrScr;
      GotoXY (1, 1);
      Write ('Anleitung');
      GotoXY (5, 3);
      Write ('Druecken Sie t, um einen Ton zu erzeugen.');
      GotoXY (5, 4);
      Write ('Druecken Sie h, um diese Hilfe zu benden.');
      GotoXY (5, 5);
      Write ('Druecken Sie e, um das Programm zu verlassen.')
    end;
  HilfeSichtbar := Not HilfeSichtbar
end;
```

(Fortsetzung)

```
begin
  CRTInit;
  TextColor (LightGray);
  TextBackground (Blue);
  Window (1, 1, 80, 1);
  ClrScr;
  Write ('(t) Ton    (e) Ende    (h) Hilfe');
  repeat
    case ReadKey of
      'e', 'E':  Ende := True;
      't', 'T':  Ton;
      'h', 'H':  Hilfe
    end
  until Ende
end.
```

Erklärung



Ein typisches CRT-Programm

Die erste Prozedur, **CRTInit**, initialisiert die Unit. **TextColor** legt die Vordergrundfarbe, **TextBackground** die Hintergrundfarbe fest. **Window** legt ein farbiges Fenster mit den angegebenen X-, Y-Koordinaten fest, welches durch die Prozedur **ClrScr** in Blau gezeichnet wird. Der erste **Window**-Aufruf sorgt für ein Menü in der obersten Zeile. Die folgende **Write**-Anweisung schreibt in das zuletzt definierte Fenster hinein. **ReadKey** wartet auf eine Tastendruck, der entsprechend ausgewertet wird.

Innerhalb der Prozedur **Ton** schaltet **Sound** den Ton ein. Das Argument der Prozedur ist die Frequenz des Tones. Der Ton bleibt so lange hörbar, bis ein Aufruf von **NoSound** ihn abschaltet. **Delay** sorgt für eine kleine Pause von 0.1*s*, in der der Ton hörbar ist.

Hilfe schreibt eine Anleitung auf den Bildschirm. Bei jedem zweiten Aufruf der Hilfe wird das erzeugte blaufarbene Fenster schwarz überschrieben. Die Prozedur **GoToXY** setzt den Textcursor an die ausgewählte Stelle innerhalb des neu erzeugten Fensters. Die linke obere Ecke eines Fensters ist die Koordinate (1;1).

Schreiben eigener Units

Eigene Units bestehen zumeist aus Sammlungen von Prozeduren, Funktionen und Typen die im Laufe einer Programmiertätigkeit anfallen. Sie sollten der Übersichtlichkeit halber thematisch geordnet sein. Die zugehörigen Dateinamen müssen kleingeschrieben werden. Eine Unit enthält mindestens zwei Bereiche, genannt **interface**- und **implementation**-Teil. Jeder dieser Bereiche darf seine eigenen Units einbinden, eigene Typen definieren und selbstverständlich auch Konstanten und Variablen deklarieren, wobei dem **implementation**-Bereich alle Informationen des **interface**-Teils bekannt sind, umgekehrt jedoch nicht.

Unit: Zeit1

```
unit Zeit1;

interface

procedure SchaltjahrAusgeben (Jahr: Integer);
procedure DatumAusgeben;

implementation

uses GPC;

procedure SchaltjahrAusgeben (Jahr: Integer);
begin
  if IsLeapYear (Jahr) then
    WriteLn (Jahr, ' ist ein Schaltjahr.')
  else
    WriteLn (Jahr, ' ist kein Schaltjahr.')
end;

procedure DatumAusgeben;
var
  Time: TimeStamp;
begin
  GetTimeStamp (Time);
  with Time do
    WriteLn (Day, '.', Month, '.', Year)
end;

end.
```

Erklärung

Eine Unit wird deklariert, indem das Schlüsselwort **unit** an den Anfang der Datei geschrieben wird. Diesem Schlüsselwort folgt der Name der Unit, der mit dem Dateinamen inhaltlich übereinstimmen muss. Die Unit endet mit dem abschließenden **end** gefolgt von einem Punkt. Im **interface**-Bereich der Unit werden diejenigen Konstanten, Typen, Variablen und Routinen aufgeführt, welche diese Unit exportiert. Nur die hier aufgeführten Elemente sind einem Programm, welches diese Unit mit **uses** einbindet, bekannt.

Im **implementation**-Teil der Unit werden die im **interface**-Bereich aufgeführten Routinen deklariert. Da wir zur Deklaration der beiden Routinen die Unit **GPC** einbinden müssen, das Interface jedoch keinen Nutzen von dieser Einbindung hat, wird die entsprechende **uses**-Anweisung im **implementation**-Bereich aufgeführt.

Diese Unit alleine ist noch nicht lauffähig. Zu ihr gehört ein Programm, welches die Unit nutzt:

Programm: UnitTest1

```
program UnitTest1;

uses Zeit1;

var
  Jahr: Cardinal;

begin
  Write ('Bitte geben Sie eine Jahreszahl ein: ');
  ReadLn (Jahr);
  SchaltjahrAusgeben (Jahr)
end.
```

Initialisierung einer Unit

Eine Unit zu initialisieren bedeutet, zum Zeitpunkt des Einbindens eine Reihe von Anweisungen auszuführen. Diese Anweisungen werden noch vor denen des Hauptprogramms ausgeführt. Eine Deinitialisierung führt analog Code aus, nachdem das Programm abgearbeitet wurde. Mit diesen Techniken ist es beispielsweise möglich, einen Stapel zu initialisieren und zum Ende des Programms wieder freizugeben. Unser Beispiel zeigt die grundsätzliche Vorgehensweise:

Unit: Zeit2

```
unit Zeit2;

interface

procedure DatumAusgeben;

implementation

uses GPC;

procedure DatumAusgeben;
var
  Time: TimeStamp;
begin
  GetTimeStamp (Time);
  with Time do
    WriteLn (Day, '.', Month, '.', Year)
end;

to begin do
  begin
    WriteLn ('Dies ist die Initialisierung');
    DatumAusgeben
  end;

to end do
  begin
    WriteLn ('Dies ist die Deinitialisierung');
    DatumAusgeben
  end;

end.
```

Erklärung

Die Initialisierung erfolgt im **to begin do**-Block. Hier werden alle Anweisungen aufgeführt, die vor dem eigentlichen Programmlauf ausgeführt werden sollen. Die Deinitialisierung erfolgt im **to end do**-Block. Das folgende Programm nutzt diese Unit:

Programm: UnitTest2

```
program UnitTest2;

uses Zeit2;

begin
  WriteLn ('Dies ist das Hauptprogramm')
end.
```

Erklärung

Die Ausgabe des Programms ist:

```
Dies ist die Initialisierung
1.12.2002
Dies ist das Hauptprogramm
Dies ist die Deinitialisierung
1.12.2002
```

Dateien

Dateien dienen dazu, Informationen dauerhaft auf Medien zu speichern. In GNU-Pascal wird unterschieden zwischen Textdateien und allgemeinen Dateien, wobei sich dieser Unterschied nur darin zeigt, mit welchen Routinen die Datei gelesen oder beschrieben wird. Das Besondere an Dateivariablen ist, dass man sie einander nicht zuweisen kann.

Aus Textdateien lesen

Das folgende Beispiel zeigt, wie eine Textdatei, in diesem Fall `/etc/passwd`, ausgelesen wird:

Programm: Lesen

```
program Lesen;

var
  Datei: Text;
  Zeile: String (1000);
  AnzahlZeilen: Integer = 0;

begin
  Assign (Datei, '/etc/passwd');
  Reset (Datei);
  while not EOF (Datei) do
    begin
      ReadLn (Datei, Zeile);
      WriteLn (Zeile);
      Inc (AnzahlZeilen)
    end;
  Close (Datei);
  WriteLn ('Habe ', AnzahlZeilen, ' Zeilen gelesen.')
end.
```

Erklärung

Der Datentyp **Text** ist ein Typ für beliebige Textdateien. Mit der Prozedur **Assign** wird eine namentlich bekannte Datei einer Variablen zugewiesen. Diese Datei kann nun mit **Reset** zum lesen geöffnet werden. **EOF** ist eine Funktion, die **True** ergibt, wenn das Ende der Datei erreicht ist. **ReadLn** trägt ein Dateiargument, um mitzuteilen, aus welcher Datei eine Datenzeile gelesen werden soll. Mit **Close** wird die Datei wieder geschlossen.

In Textdateien schreiben

Das folgende Beispiel zeigt, wie in eine Textdatei geschrieben[36] wird:

Programm: Schreiben

```
program Schreiben;

var
  Datei: Text;
  Zeile: String (1000);

begin
  Assign (Datei, 'namen.text');
  Rewrite (Datei);
  Write ('Wie lautet ihr Name? ');
  ReadLn (Zeile);
  WriteLn (Datei, Zeile);
  Close (Datei)
end.
```

Erklärung

In bekannter Weise wird mit **Assign** ein Dateiname einer Datei zugewiesen. Diese Datei wird mit **Rewrite** zum Schreiben geöffnet. Eine Textzeile wird mit Hilfe von **WriteLn** in die Datei geschrieben, wobei auch hier ein zusätzlicher Parameter die Zieldatei angibt.

Vordefinierte Textdateien

Es existieren drei vordefinierte Textdateien, die Standard-Eingabe, Standard-Ausgabe und eine Standard-Fehlerdatei. Die ersten beiden Dateien benutzen sie immer, wenn Sie Texte mit **ReadLn** einlesen oder mit **WriteLn** ausgeben. Die dritte Datei muss explizit angesprochen werden:

Programm: Vordef

```
program Vordef;

begin
  WriteLn ('Normale Ausgabe');
  WriteLn (StdErr, 'Fehlerausgabe')
end.
```

Erklärung

Beide Strings werden in unterschiedliche Dateien geschrieben, allerdings merken Sie davon nichts, wenn Sie das Programm nur ausführen. Erst mit `vordef 2>/dev/null`, wobei Sie die Zwei[37] auch wahlweise durch Eins ersetzen können, ändert sich die Ausgabe. **StdErr** bezeichnet diesen Fehlerkanal, auf den Sie Fehlermeldungen ausgeben können.

36 Bitte beachten Sie, dass diese Datei in Ihrem gewählten Verzeichnis nicht existieren darf, sonst wird sie überschrieben!

37 Hintergrund: 1 bedeutet auf der Shell die Standard-Ausgabe während 2 den Standard-Fehlerkanal bezeichnet. Der Teil `2>/dev/null` leitet die Ausgabe des Fehlerkanals so um, dass Sie von der Fehlermeldung nichts zu sehen bekommen. Genauso verfährt man mit der Standard-Eingabe.

Schreiben und Lesen von allgemeine Dateien

Das Schreiben und Lesen von allgemeinen Dateien erfolgt analog zum Schreiben und Lesen von Textdateien:

Programm: Allgdatei

```
program Allgdatei;

var
  Datei: File;
  i, Wert: Integer;

begin
  Assign (Datei, 'werte.dat');

  { Schreiben }
  Rewrite (Datei, SizeOf (Integer));
  for i := 0 to 10 do
    BlockWrite (Datei, i, 1);
  Close (Datei);

  { Lesen }
  Reset (Datei, SizeOf (Integer));
  for i := 0 to 10 do
    begin
      BlockRead (Datei, Wert, 1);
      WriteLn ('Wert = ', Wert)
    end;
  Close (Datei)
end.
```

Erklärung

Der Datentyp **File** ist der Typ einer allgemeinen Datei. Geöffnet wird eine solche Datei zum Schreiben mit **Rewrite**, wobei als weiterer Parameter die Größe der Daten, die in die Datei geschrieben werden sollen, anzugeben ist. Geschrieben wird in eine solche Datei mit **BlockWrite**, wobei das erste Argument die Datei ist, das Zweite die Daten und das dritte Argument die Anzahl der Daten, die geschrieben werden sollen, bezeichnet. **Reset** hat ebenfalls einen zusätzlichen Parameter, der die Blockgröße der zu lesenden Daten angibt. **BlockRead** liest aus dieser Datei und hat dieselben Argumente wie **BlockWrite**.

Dateien als Programmparameter

Dateien lassen sich auch im Kopf eines Programms deklarieren. Damit bekommen sie eine Bedeutung, die mit oben erklärten Techniken nicht zu implementieren ist:

Programm: ZeilenZahl

```
program ZeilenZahl (Input);

var
  Zeile: String (1000);
  AnzahlZeilen: Integer = 0;

begin
  Reset (Input);
  while not EOF (Input) do
    begin
      ReadLn (Input, Zeile);
      Inc (AnzahlZeilen)
    end;
  WriteLn ('Habe ', AnzahlZeilen, ' Zeilen gelesen.');
  Close (Input)
end.
```

Erklärung

Wenn Sie dieses Programm ausführen, so erwartet es Eingaben von Ihnen, bis Sie `STRG+D` drücken. Viel interessanter ist es aber, das Programm als Filter einzusetzen und durch das Programm hindurchzupipen: `cat zeilenzahl.pas | zeilenzahl`. Dieses Programm behandelt die ihr übergebenen Textdaten als Datei mit unbekanntem Dateinamen, jedoch bekannter Dateivariablen **Input**. Diese Datei kann zum Lesen geöffnet werden. Hinter **Input** dürfen noch weitere Dateivariablen stehen.

Schreiben und Lesen von typbehafteten Dateien

Während man in allgemeine Dateien beliebige Daten ablegen kann und Textdateien zur Speicherung von Text dienen, gibt es in typisierten Dateien die Möglichkeit, eine Menge Daten vom selben Typ abzulegen:

Programm: Typisiert

```
program Typisiert;

var
  Datei: File of Integer;
  i: Integer;

begin
  Assign (Datei, 'foo.dat');

  Rewrite (Datei);
  for i := 0 to 10 do
    Write (Datei, i);
  Close (Datei);

  Reset (Datei);
  while not EOF (Datei) do
    begin
      Read (Datei, i);
      WriteLn (i)
    end;
  Close (Datei)
end.
```

Erklärung

In diesem Beispiel haben wir eine Datei erzeugt, die nur aus Integer-Werten bestehen kann. Dies erreichen wir, indem bei der Variablendeklaration der Datei der zu speichernde Datentyp angegeben wird. In diese Datei kann mit **Write** geschrieben und mit **Read** aus ihr gelesen werden. Bei der Verwendung dieser Dateisorte ist es nicht notwendig, **Rewrite** oder **Reset** zusätzliche Parameter mitzugeben.

Programmierbeispiel: Adressdatei

Das folgende Programmierbeispiel implementiert eine kleine Adressdatenbank. Sie werden aufgefordert, beliebig viele Adressen einzugeben. Anschließend wird die komplette Datenbank ausgegeben. Als Besonderheit ist das Programm so angelegt, dass die Daten bei einem erneuten Durchlauf nicht verloren gehen, sondern immer weiter angehängt werden:

Programm: Adressedat

```
program Adressedat;

type
  TEintrag = String (30);
  TPlz     = String (5);
  TAdresse = record
    Vorname, Name, Strasse: TEintrag;
    Plz: TPlz;
    Stadt: TEintrag
  end;
  TAdressDatei = File of TAdresse;

var
  Datei: TAdressDatei;
  Adresse: TAdresse;
  Eingabe: Char;

function FrageAdresse: TAdresse;
var
  Anschrift: TAdresse;
begin
  with Anschrift do
    begin
      Write ('Vorname: ');
      ReadLn (Vorname);
      Write ('Name: ');
      ReadLn (Name);
      Write ('Strasse: ');
      ReadLn (Strasse);
      Write ('Postleitzahl: ');
      ReadLn (Plz);
      Write ('Stadt: ');
      ReadLn (Stadt)
    end;
  FrageAdresse := Anschrift
end;

procedure DruckeAdresse (Anschrift: TAdresse);
begin
  with Anschrift do
    begin
      WriteLn (Vorname, ' ', Name);
      WriteLn (Strasse);
      WriteLn (Plz, '-', Stadt)
    end
end
```

(Fortsetzung)

```
procedure SchreibeAdresse (var AdressDatei: TAdressDatei; Anschrift:
TAdresse);
begin
  Write (AdressDatei, Anschrift)
end;

procedure LiesAdresse (var AdressDatei: TAdressDatei; var Anschrift:
TAdresse);
begin
  Read (AdressDatei, Anschrift)
end;

begin
  Assign (Datei, 'adressen.dat');
  Append (Datei);
  repeat
    Adresse := FrageAdresse;
    SchreibeAdresse (Datei, Adresse);
    Write ('Eine weitere Adresse eingeben (j/n): ');
    ReadLn (Eingabe)
  until Eingabe in ['n', 'N'];
  Close (Datei);

  { Adressen ausgeben }
  WriteLn ('=== Adressen ===');
  Assign (Datei, 'adressen.dat');
  Reset (Datei);
  while not EOF (Datei) do
    begin
      LiesAdresse (Datei, Adresse);
      DruckeAdresse (Adresse)
    end;
  Close (Datei)
end.
```

Erklärung

Es wird ein Typ einer typenbehafteten Datei definiert, der in den Funktionen genutzt werden kann. Dateivariablen müssen immer im Original, also Call by Reference, übergeben werden. Die Routine **Append** öffnet eine Datei zum Anhängen von Daten.

Internationalisierung

Veröffentlichte Programme werden nicht selten von Menschen aus den unterschiedlichsten Nationen benutzt. Damit diese Programme auch benutzt werden können ist es oft erforderlich, sie zu internationalisieren. Die folgenden Kapitel geben einen Überblick über die Möglichkeiten von GNU-Gettext[38]. Für die folgenden Beispiele benötigen Sie ein Programm namens `pas2po`[39], welches Sie im Quelltext von *http://www.gnu-pascal.de/contrib/eike/* herunterladen können.

Ein Programm Internationalisieren

Quelltext vorbereiten

Nehmen wir die englischsprachige Fassung des ersten Beispiels aus dem Kapitel Das erste Programm als Grundlage für die ersten Gehversuche mit GNU-Gettext:

Programm: Inter1

```
program Inter1;

begin
  WriteLn ('Hello World!')
end.
```

Das Programm gibt bei der Ausführung den englischsprachigen Text "Hello World!" aus. Dieses Programm wollen wir nun internationalisieren. Wir fügen dem Programm die benötigten Informationen hinzu, damit ein "Sprachenkatalog" in der gewählten Landessprache gefunden werden kann und natürlich muss das Programm wissen, welche Sprache es gerade auszugeben hat:

Programm: Inter2

```
program Inter2;

uses GPC, Intl;

const
  LocaleDir = '/usr/share/locale';

var
  Dummy: TString;

begin
  Dummy := SetLocale (LC_MESSAGES, );
  Dummy := BindTextDomain ('inter2', LocaleDir);
  Dummy := TextDomain ('inter2');
  WriteLn (GetText ('Hello World!'))
end.
```

38 Wenn Sie mehr darüber erfahren wollen, so schauen sie sich bitte die Hilfen zu den Themen `gettext`, `xgettext` und `msgfmt` an.

39 Dieses Programm ersetzt das bei C-Programmierern beliebte `xgettext`.

Erklärung

Zuerst legen wir das Verzeichnis fest, in dem die so genannten "Locale-Informationen" abgelegt sind. Unter Linux und den meisten anderen Betriebssystem ist dies `/usr/share/locale` oder `/usr/local/share/locale`[40]. Innerhalb dieses Verzeichnisses sollten diverse Verzeichnisse liegen, die mit einer Länderkennung versehen sind. Schauen Sie sich dort ruhig einmal um, denn innerhalb von z. B. `de` befindet sich ein weiteres Verzeichnis mit dem Namen `LC_MESSAGES`. In diesem Verzeichnis liegen alle deutschsprachigen Kataloge von den verschiedensten Programmen. Um die gegenwärtige verwendete Spracheinstellung zu benutzen ruft man **SetLocale**[41] auf. Wollen Sie eine bestimmte Sprache angeben, so können sie dies dort machen, wo die leere Zeichenkette steht. Eine französischsprachige Ausgabe erhalten Sie beispielsweise mit **SetLocale (LC_MESSAGES, 'fr_FR');**, sofern die entsprechenden Dateien auf ihrem System installiert sind.

Der Funktionsaufruf **BindTextDomain** sucht eine Datei mit dem angegebenen Namen im angegebene Pfad. In unserem Fall würde **BindTextDomain** also im Pfad `/usr/share/locale/de/LC_MESSAGES/` eine Datei suchen, die `inter2.mo` heißt. Diese Datei wird durch den Aufruf von **TextDomain** für alle weiteren Aufrufe von **GetText** benutzt. **GetText** sucht in der angegebenen Datei nach der Übersetzung des übergebenen Strings zur vorher gewählten Locale-Einstellung.

Sprachenkatalog erzeugen

Da wir den Quelltext vorbereitet haben sind wir vom erstellen des Sprachenkataloges nur noch wenige Programmaufrufe entfernt. Mit `pas2po -o inter2.po inter2.pas` erzeugen wir eine Datei, welche alle zur Übersetzung vorbereiteten Strings enthält. Nur diejenigen Zeichenketten werden hier aufgeführt, die mit **GetText** vorbereitet wurden:

```
# This file was created by pas2po with 'inter2.pas'.
# Please change this file manually.
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR Free Software Foundation, Inc.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"POT-Creation-Date: 2003-04-01 21:54+0200\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"

#foo1.pas:15
msgid "Hello World!"
msgstr ""
```

Ab dem Zeitpunkt, wo diese Datei vorliegt gibt es keinen Grund mehr, den originalen Quelltext zu besitzen. Jedes beliebige entsprechend vorbereitete Programm lässt sich übersetzen, wenn diese Datei vorliegt.

Alle großgeschriebenen Einträge hinter den Doppelpunkten müssen entsprechend bearbeitet werden. Für die genaue Bedeutung dieser Einträge wird auf die Dokumentation von `gettext` verwiesen. Das wichtigste an dieser Datei ist die Übersetzung der angegebenen Zeichenkette. Im Folgenden wird die entsprechend bearbeitete Datei angegeben:

40 Auf die betriebssystemspezifischen Details dieses Pfades gehen wir in Anhang A, Systemspezifische Details ein.
41 Mit dem Befehl `man setlocale` bekommen Sie weitere Informationen zu diesem Systemaufruf.

```
# This file was created by pas2po with 'inter2.pas'.
# Please change this file manually.
# Deutschsprachige Uebersetzung des Programms inter2
# Eike Lange <eike@gnu.de>, 2003.
#
msgid ""
msgstr ""
"Project-Id-Version: inter2 1.0\n"
"POT-Creation-Date: 2003-04-01 21:54+0200\n"
"PO-Revision-Date: 2003-04-01 22:30+0200\n"
"Last-Translator: Eike Lange <eike@gnu.de>\n"
"Language-Team: <>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=ISO-8859-1\n"
"Content-Transfer-Encoding: 8bit\n"

#foo1.pas:15
msgid "Hello World!"
msgstr "Hallo Welt!"
```

Der Befehl `msgfmt inter2.po -o inter2.mo` erzeugt aus der bearbeiteten Datei `inter2.po` eine Datei namens `inter2.mo`. Diese Datei war das Ziel all unserer Bemühungen.

Mit dem Befehl `cp inter2.mo /usr/share/locale/de/LC_MESSAGES/` wird diese Datei an die richtige Stelle im System installiert.

Wenn wir jetzt das Programm aufrufen, so erscheint der deutschsprachige übersetzte Text der im Quelltext weiterhin englischsprachig vorliegt. Es wird zur Laufzeit des Programms entschieden, welche Sprache zur Ausgabe benutzt wird.

Argumente Tauschen

In vielen Sprachen ist die Reihenfolge von Argumenten wichtig. Betrachten wir dazu die folgenden Sätze:

```
Die Antwort ist 42.
42 ist die Antwort.
Die Antwort kann 42 sein.
```

Um dieser Reihenfolge in verschiedenen Sprachen Rechnung zu tragen wurde die Funktion **FormatString** entwickelt, die wir direkt in unseren Quelltext einbauen können:

Programm: Inter3

```
program Inter3;

uses GPC, Intl;

const
  LocaleDir = '/usr/local/share/locale';

var
  Ignore, Answer, s: TString;

begin
  Ignore := SetLocale (LC_MESSAGES, );
  Ignore := BindTextDomain ('inter3', LocaleDir);
  Ignore := TextDomain ('inter3');
  Write (GetText ('What is the answer of all questions? '));
  ReadLn (Answer);
  s := FormatString (GetText ('%@1a %@2a'), Answer,
    GetText (' is the answer.'));
  WriteLn (s)
end.
```

Erklärung

Das erste Argument von **FormatString** ist eine Zeichenkette, die eine Nummerierung aller folgenden Argumente enthält. **%@1a** ist das Erste, **%@2a** das Zweite und **%@100a** das hundertste folgende Argument. Ohne eine Übersetzung würde das Programm etwa folgende Ausgabe liefern:

```
What is the answer of all questions? 42
42 is the answer.
```

Wir wollen nun das Programm internationalisieren, wobei wir die Argumente derart vertauschen, dass der eingegebene Text hinter den erklärenden Text platziert wird. Hierzu führen wir wieder das Programm `pas2po -o inter3.po inter3.pas` aus und erhalten, neben dem uns schon bekannten Kopfteil, folgende Ausgabe:

```
#inter3.pas:15
msgid "What is the answer of all questions? "
msgstr ""

#inter3.pas:17
msgid "%@1a %@2a"
msgstr ""

#inter3.pas:18
msgid "is the answer."
msgstr ""
```

Wenn wir diese Datei nun übersetzen, achten wir besonders darauf, die Formatreihenfolge zu vertauschen:

```
# Message File for Inter3
#
msgid ""
msgstr ""
"Project-Id-Version: inter3 1.0\n"
"POT-Creation-Date: 2003-04-02 15:34+0200\n"
"PO-Revision-Date: 2003-04-02 15:34+0200\n"
"Last-Translator: Eike Lange <eike@gnu.de>\n"
"Language-Team: German <de@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=ISO-8859-1\n"
"Content-Transfer-Encoding: 8bit\n"

#inter3.pas:15
msgid "What is the answer of all questions? "
msgstr "Wie lautet die Antwort auf alle Fragen? "

#inter3.pas:17
msgid "%@1a %@2a"
msgstr "%@2a %@1a"

#inter3.pas:18
msgid "is the answer."
msgstr "Die Antwort lautet:"
```

Nach dem Erzeugen der `.mo`-Datei mit `msgfmt inter3.po -o inter3.mo` und dem Installieren nach `/usr/local/share/locale/de/LC_MESSAGES/` lautet die Ausgabe unseres Programms wie folgt:

```
Wie lautet die Antwort auf alle Fragen? 42
Die Antwort lautet: 42
```

Pascal und C

Viele interessante Programmierbibliotheken werden in der Programmiersprache "C" geschrieben. Wie diese Bibliotheken in GNU-Pascal genutzt werden können, möchte dieses Kapitel vermitteln.

C-Funktionen in Pascal-Programmen

Das einführende Beispiel beschreibt, wie man eine einfache C-Funktion[42] in ein GNU-Pascal-Programm einbindet:

Programm: hallo.c

```
/* hallo.c */
#include <stdio.h>

void print_hallo (void)
{
  printf ("Hallo, Welt!\n");
}
```

Das aufrufende GNU-Pascal-Programm sieht so aus:

Programm: PasC1

```
program PasC1;

procedure PrintHallo; external; attribute (name='print_hallo');

begin
  PrintHallo
end.
```

Erklärung

Die beiden Quelltexte müssen mit `gpc hallo.c pasc1.pas -o pasc1` übersetzt werden, das Aufrufen des Programmes erzeugt die erwartete Ausgabe. Die Prozedur `PrintHallo` wird hier als `external` deklariert, was bedeutet, dass sie irgendwo definiert ist, wir uns aber für den Ort, wo sie definiert ist nicht näher interessieren. Das Attribut `name` legt fest, dass der vom Compiler intern vergebene Name dieser Prozedure der Gleiche ist, wie der unserer C-Funktion. Dadurch wird `PrintHallo` zu einem Synonym für `print_hallo`. Intern verweisen beide Funktionen auf denselben Namen und dadurch auf denselben Code

.

42 In der Programmiersprache "C" wird nicht zwischen Funktionen und Prozeduren unterschieden. In C ist jede Routine eine Funktion.

C-Funktionen in Pascal-Units

Bei größeren Sammlungen von C-Routinen empfiehlt es sich, diese Routinen in Units einzubetten. Wir benutzen dazu wieder obigen C-Quelltext als Grundlage:

Programm: HalloC

```
unit HalloC;

interface

procedure PrintHallo; external; attribute (name='print_hallo');

implementation
{$L hallo.c}

end.
```

Das aufrufende Programm sieht so aus:

Programm: PasC2

```
program PasC2;

uses HalloC;

begin
  PrintHallo
end.
```

Erklärung

Die Übersetzung des Programmes vereinfacht sich nun zu `gpc --automake pasc2.pas -o pasc2`, da der C-Quelltext in die Unit mit Hilfe von **{$L hallo.c}** eingebunden wird.

Strukturen und Funktionen

Das Teilen von Strukturen und Funktionen zwischen verschiedenen Sprachen ist zumeinst der Hauptzweck für das mehrsprachige Programmieren. Für einfache Typen gibt es folgende Übersetzungshilfe:

C-Datentyp	GNU-Pascal-Datentyp
char *	CString
(unsigned) char	ByteInt (ByteCard)
(unsigned) short int	ShortInt (ShortCard)
(unsigned) int	CInteger (CCardinal)
(unsigned) long int	MedInt (MedCard)
(unsigned) long long int	LongInt (LongCard)
float	ShortReal
double	Real

Records lassen sich aus den vorhandenen C-Strukturen übernehmen. Bei Konstanten und Aufzählungstypen verwenden wir Variablen, denen das Attribut `const` nachgestellt wird.

Funktionsparameter werden ähnlich übersetzt, wobei Referenzparameter in C wie Zeiger behandelt werden. Alle C-Funktionen, die `void` zurückliefern, können in Pascal als Prozeduren implementiert werden.

Wrapper schreiben

Wrapper sind Funktionen, die in der Quellsprache geschrieben werden und die eigentlich zu übernehmenden Funktionen überlagern. Dadurch wird selbst bei einem sich ändernden Parameter- oder Rückgabetyp der Originalfunktion der Aufruf aus einer Pascal-Routine nicht beeinträchtigt. Dieser Vorteil ist mit einem kleinen Nachteil verbunden: Es werden nunmehr zwei zusätzliche Dateien benötigt, die Pascal-Unit und der Wrapper Quelltext.

C-Datei: mathe.c

```
/* mathe.c */
int addiere (int a, int b)
{
  return a + b;
}
```

Auf die Datei `mathe.c` hat man üblicherweise keinen Zugriff, sie versteckt sich innerhalb veröffentlichter Programmier-Bibliotheken. Die öffentliche Schnittstelle zu solchen Funktionen ist die Header-Datei:

C-Datei: mathe.h

```
/* mathe.h */
int addiere (int a, int b);
```

Die Wrapper-Datei sieht wie folgt aus:

C-Datei: mathec.c

```
/* mathec.c */
#include "mathe.c"

int _c_addiere (int a, int b)
{
  return (int) addiere (a, b);
}
```

Die Unit, welche die Funktionen aus `mathec.c` benutzt folgt nun:

Unit: Mathe

```
unit Mathe;

interface

function Addiere (a, b: Integer): Integer; external name '_c_addiere';

implementation

{$L mathec.c}

end.
```

Erklärung

Die Schreibweise **external name '_c_addiere'** ist eine Kurzschreibweise für **external; attribute (name='_c_addiere')** . Die C-Funktion **addiere (int a, int b)** darf sich nun in gewissen Grenzen ändern, ohne dass das Pascal-Interface davon berührt wird. Selbst wenn sie in zukünftigen Versionen der Mathematik-Bibliothek ganz wegfallen sollte, so braucht sie nur noch in der Wrapper-Datei neu berücksichtigt zu werden.

Übersetzen einer Bibliothek

Das Übersetzen einer Bibliothek sollte mit dem bis hierher erworbenen Wissen leicht fallen. Es handelt sich dabei um einen kleinen Ausschnitt der Bibliothek `GTK`. Hier folgt ein Testprogramm aus dem Tutorial der `GTK`:

C-Programm: gtk_test.c

```
/* gtk_test.c */
#include <gtk/gtk.h>
int main(int argc, char *argv[])
{
  GtkWidget *window;
  gtk_init (&argc, &argv);
  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_widget_show (window);
  gtk_main ();
  return 0;
}
```

Das Programm enthält einen Typ (**GtkWidget**), eine Konstante (**GTK_WINDOW_TOPLEVEL**) und eine Reihe von C-Funktionen, die es nach Pascal zu übersetzen gilt.

GtkWidget wird in der gesamten Biblothek nur als Zeiger verwendet, weswegen für diesen Typ der allgemeine **Pointer** in Frage kommt.

Die erste Schwierigkeit taucht bei **gtk_init (&argc, &argv)** auf. Pascal verwendet nicht den selben Mechanismus, um Programmparameter zu referenzieren wie C. Zu diesem Zweck wurden in GNU-Pascal eigene Variablen eingefügt, welche **CParamCount** und **CParameters** heissen. Diese Variablen ersetzen die üblichen Parameter **argc** und **argv** der **main**-Funktion.

Die Konstante **GTK_WINDOW_TOPLEVEL** ist in der C Header-Datei **gtkenums.h** als Aufzählungstyp definiert:

```
/* Window types */
typedef enum
{
  GTK_WINDOW_TOPLEVEL,
  GTK_WINDOW_POPUP
} GtkWindowType;
```

Da sich prinzipiell ihr Wert zwischen zwei Versionen ändern kann, müssen wir sie durch eine Wrapper-Konstante umfassen:

C-Datei: gtkc.c

```
/* gtkc.c */
#include <gtk/gtk.h>

/* Wrapper für Konstanten */
const int _c_GTK_WINDOW_TOPLEVEL = GTK_WINDOW_TOPLEVEL;
const int _c_GTK_WINDOW_POPUP = GTK_WINDOW_POPUP;
```

Die oben genannten Überlegungen fassen wir nun in eine Unit zusammen:

Unit: GTK

```
unit GTK;

interface

type
  GtkWidget = Pointer;

var
  GTK_WINDOW_TOPLEVEL: CInteger;
    external; attribute (name = '_c_GTK_WINDOW_TOPLEVEL', const);
  GTK_WINDOW_POPUP   : CInteger;
    external; attribute (name = '_c_GTK_WINDOW_POPUP', const);

procedure GtkInit;
procedure GtkMain; external name 'gtk_main';
procedure GtkWidgetShow (Widget: GtkWidget);
  external name 'gtk_widget_show';
function  GtkWindowNew (WindowType: CInteger): GtkWidget;
  external name 'gtk_window_new';

implementation
{$L gtkc.c}

uses GPC;

procedure CGtkInit (ArgC: Pointer; ArgV: Pointer);
  external name 'gtk_init';
procedure GtkInit;
begin
  CGtkInit (@CParamCount, @CParameters)
end;

end.
```

Erklärung

Da die beiden Konstanten **GTK_WINDOW_TOPLEVEL** und **GTK_WINDOW_POPUP** mehr oder weniger eine Einheit bilden haben wir sie zusammen aufgeführt und in die Unit übernommen. Diese beiden Konstanten wurden als Variablen eingebunden, die einen Konstanten Wert haben und extern deklariert wurden. Die Prozedur **GtkInit** musste im Implementationsteil zweimal aufgeführt werden. Einmal in ihrer "Originaldarstellung" und einmal so, wie wir sie benutzen wollen, nämlich ohne Argumente. Dies können wir hier tun, da diese Prozedur immer mit denselben Argumenten aufgerufen wird.

Das Programm, welches diese Unit testet sieht wie folgt aus:

Programm: GTKTest

```
program GTKTest;

uses GTK;

var
  Window: GtkWidget;

begin
  GtkInit;
  Window := GtkWindowNew (GTK_WINDOW_TOPLEVEL);
  GtkWidgetShow (Window);
  GtkMain
end.
```

Dieses Programm ist die direkte Übersetzung des am Anfang des Abschnitts in C vorgeführten Quelltextes und wird wie folgt übersetzt:

```
gpc --automake gtktest.pas -o gtktest `pkg-config gtk+-2.0 --cflags --libs`
```

Ausblick

Das Übersetzen von Bibliotheken ist eine sehr dankbare Aufgabe. Wenn sie sich ihr gewachsen fühlen und tatsächlich die `GTK` oder eine andere Bibliothek für GNU-Pascal benutzbar machen wollen, so melden Sie sich bitte auf der Mailingliste. Andere Bibliotheken, die von Interesse sein könnten sind `ALSA` und `libsndfile`.

Anhang A

Systemspezifische Details

Wie an einigen Stellen innerhalb dieser Dokumentation schon angemerkt, gibt es systemspezifische Details bei der Benutzung von GNU-Pascal zu beachten. Dieses Kapitel dient dazu, solche Details aufzuzählen, sofern sie in der Dokumentation angemerkt wurden.

Dateinamen erzeugter Programme

Unter einigen Betriebssystemen ist es üblich, ausführbaren Programmen einen namentlichen Anhang zu geben der daraufhindeutet, dass diese Datei ausführbar ist. Solche Anhänge heißen zumeist `com` oder `exe`. GNU-Pascal erzeugt solche Anhänge auf Systemen, die sie benötigen wie folgt: `gpc erstes.pas -o erstes.exe` erzeugt also eine Datei, die ausführbar ist und `erstes.exe` heißt. Die meisten Betriebssysteme versehen diese Dateien mit einem Attribut, welches das Recht zur Ausführung bedeutet und können so auf Namensanhänge verzichten.

Locale Verzeichnisse

Das Verzeichnis, in dem sie die sogenannten Locale Informationen finden können ist betriebssystemspezifisch. Unter Linux und diversen UNIX-artigen Betriebssystemen lautet dieser `/usr/share/locale/` oder `/usr/local/share/locale/`. Unter MacOS-X lautet dieser Pfad `/usr/share/locale/` oder `/sw/share/locale/`. Unter DJGPP können sie diesen Pfad herausbekommen, indem sie

```
Path := GetEnv ('$DJDIR') + '/share/locale';
```

in ihren Quellcode schreiben. Für die Umgebungen MinGW32 und CygWin liegen dem Autor keine speziellen Details vor.

Weblinks

GNU-Pascal

GNU-Pascal: *http://www.gnu-pascal.de*

GNU-Projekt: *http://www.gnu.org* (Englischsprachig)

Wikibooks

Wikibooks: *http://de.wikibooks.org*

Dieses Buch: *http://de.wikibooks.org/wiki/GNU-Pascal in Beispielen*

Wikimedia Foundation: *http://www.wikimedia.org*

Anhang B

GNU Free Documentation License

Version 1.2, November 2002; Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.; 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA; Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies.

The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers. If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice.

These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Copyrighthinweise und Autorenverzeichnisse

Bilder

Name	*Lizenz*	*Urheber*	*Sonstiges*
Cover	GFDL	Free Software Foundation	
Copyleft-Symbol	Gemeinfrei		Schöpfungshöhe nicht erreicht
Abbildung 1	GFDL	GnuShi	Siehe www.gnushi.de

Ursprungstext aus de.wikibooks.org

Hier ist die Liste der Autoren zu entnehmen, wobei beachtet werden sollte, dass nicht unbedingt die Autoren mit den meisten Edits das meiste daran geleistet haben müssen. Über die Größe und Qualität der einzelnen Editierungen wird nichts genannt.

Autor	*Editierungen*	*Autor*	*Editierungen*
GnuShi	90	IP: 82.207.252.114	10
Stefan Majewsky	2	IP: 84.134.76.67	3
Klaus Eifert	2	IP: 80.140.81.175	2
MichaelFrey	1	IP: 80.144.156.26	1
Klartext	1	IP: 82.136.219.131	1
Zero	1	IP: 84.149.155.159	1
		IP: 217.224.173.7	1
		IP: 141.51.48.148	1
		IP: 82.207.235.162	1
		IP: 82.207.218.82	1
		IP: 134.28.33.26	1

Eine genaue Versionsgeschichte kann unter *http://de.wikibooks.org/GNU-Pascal in Beispielen* und dessen Unterseiten nachgelesen werden.

Enddokument

Modifikationen gegenüber Ursprungstexten

Die Modfikationen beschränken sich zur ersten Version hin hauptsächlich auf die bessere Les- und Druckbarkeit (Design, Bildauswahl, Tabellen, Links etc.) und die Beseitigung kleinerer Rechtschreibfehler. Abgesehen von der für diese Publikation nötigen rechtlichen Anhänge ist der Inhalt nicht geändert oder erweitert worden.

Transparent Copy

Die Spezifikationen des zum verwahren verwendeten Aufbewahrungstyps sind frei erhältlich, weshalb die von der Lizenz geforderte „transparent copy“ vernachlässigt werden kann. Das Dokument selbst ist „transparent“.

Historie

Datum	*Beschreibung*
2.1.07	• Fertigstellung der ersten Version 1.0 (Zualio)
1.4.07	• Behebung kleinerer Fehler und leichte Verbesserung des Designs von Anhang B; Version 1.1 (Zualio)

Konvertierung und Überarbeitung

Christopher Schlosser (alias de.Wikibooks.org-Account „Zualio“)