

Modul 118

| | |
|--|--|
| 1 ANSI C-Grundlagen.....3 | 1.12 Arrays.....15 |
| 1.1 Hinweis zur Syntax.....3 | 1.12.1 Deklaration.....16 |
| 1.2 Der Präprozessor.....3 | 1.12.2 Initialisierung.....16 |
| 1.2.1 Die include-Anweisung.....3 | 1.12.3 C-Strings.....16 |
| 1.2.2 Die define-Anweisung.....3 | 1.12.4 Zugriff.....16 |
| 1.3 Das Hauptprogramm.....3 | 1.12.5 Dimensionen.....17 |
| 1.4 Kommentare.....4 | 1.13 Zeiger.....17 |
| 1.5 Variablen.....4 | 1.13.1 Deklaration.....17 |
| 1.5.1 Datentypen.....4 | 1.13.2 Initialisierung.....17 |
| 1.5.2 Deklaration.....5 | 1.13.3 Zugriff.....18 |
| 1.5.3 Konstanten.....6 | 1.13.4 Merkregel.....18 |
| 1.6 Ausdrücke.....6 | 1.13.5 Zeiger auf Arrays.....19 |
| 1.6.1 wahre und unwahre Ausdrücke.....6 | 1.13.6 Zeiger auf Strukturen.....19 |
| 1.7 Operatoren.....6 | 1.14 Funktionen.....20 |
| 1.7.1 Arithmetische Operatoren.....7 | 1.14.1 Der Funktionskopf.....20 |
| 1.7.2 Zuweisungsoperatoren.....7 | 1.14.2 Der Funktionsrumpf.....21 |
| 1.7.3 In- und Dekrementierung.....8 | 1.14.3 Der Rückgabewert.....21 |
| 1.7.4 Vergleichsoperatoren.....8 | 1.14.4 Der Prototyp.....22 |
| 1.7.5 Logische Operatoren.....8 | 1.14.5 Funktionsaufruf.....22 |
| 1.7.6 Bit-Operatoren.....8 | 1.14.6 Wert- und Referenzübergabe.....22 |
| 1.7.7 Operatoren für den Speicherzugriff.....9 | 1.15 Makros.....23 |
| 1.7.8 sonstige Operatoren.....9 | 1.16 Die C-Standardbibliothek.....23 |
| 1.8 Anweisungen.....10 | 1.16.1 Ein – und Ausgaben.....24 |
| 1.8.1 Codeblöcke.....10 | 1.16.2 Mathematische Funktionen.....24 |
| 1.9 Verzweigungen.....10 | 1.16.3 String-Verarbeitung.....25 |
| 1.9.1 Verzweigungen mit if.....10 | 1.16.4 Speichermanipulation.....25 |
| 1.9.2 Else if.....11 | 1.17 Anhang A – der Formatstring.....26 |
| 1.9.3 Else.....11 | 2 C#-Grundlagen.....27 |
| 1.9.4 Switch case.....11 | 2.1 Syntax.....27 |
| 1.10 Schleifen.....12 | 2.2 OOP.....27 |
| 1.10.1 Goto.....12 | 2.2.1 Der strukturierte Ansatz.....27 |
| 1.10.2 Die while-Schleife.....12 | 2.2.2 Der objektorientierte Ansatz.....27 |
| 1.10.3 Die do-while-Schleife.....12 | 2.2.3 Klassen.....27 |
| 1.10.4 Die for-Schleife.....13 | 2.2.4 Objekte.....28 |
| 1.10.5 break.....13 | 2.2.5 Objektstatus.....28 |
| 1.10.6 continue.....13 | 2.2.6 Eigenschaften.....28 |
| 1.11 Eigene Datentypen.....14 | 2.2.7 Methoden.....28 |
| 1.11.1 Enumeratoren.....14 | 2.2.8 Signatur.....28 |
| 1.11.2 Strukturen.....15 | 2.3 Programmaufbau.....28 |

| | | | |
|--|--------------------|--|---------------------------|
| 2.3.1 Die Main-Methode..... | 28 | 2.7.1 Statische Datenfelder..... | 35 |
| 2.3.2 wohin mit Main?..... | 28 | 2.7.2 Dynamische Datenfelder..... | 35 |
| 2.4 Erweiterte Datentypen verwenden..... | 29 | 2.7.3 Erster dynamischer Ansatz..... | 35 |
| 2.4.1 Datentypen..... | 29 | 2.8 Verkettete Dynamische Datenfelder..... | 36 |
| 2.4.2 Schlüsselwörter..... | 30 | 2.9 ArrayList..... | 36 |
| 2.4.3 Konstante..... | 30 | 2.9.1 Eigenschaften und Methoden..... | 37 |
| 2.4.4 Typsuffixe..... | 30 | 3 Strukturierte Diagramme..... | 38 |
| 2.4.5 Casts..... | 31 | 3.1 Das Flussdiagramm..... | 38 |
| 2.5 Enumeratoren | 31 | 3.2 Struktogramme..... | 39 |
| 2.5.1 Zugriff..... | 31 | 4 Projekte..... | 42 |
| 2.6 Datenfelder (Arrays)..... | 31 | 4.1 Projektorganisation..... | 42 |
| 2.6.1 Deklaration..... | 32 | 4.2 Phasenkonzepte..... | 42 |
| 2.6.2 Instanziierung..... | 32 | 4.3 Beispiel..... | 43 |
| 2.6.3 Initialisierung..... | 32 | 4.3.1 Initialisierung & Vorstudie..... | 43 |
| 2.6.4 Zugriff mit for-Schleife..... | 32 | 4.3.2 Grobkonzept..... | 45 |
| 2.6.5 Zugriff mit foreach-Schleife..... | 33 | 4.3.3 Evaluation (Variantenanalyse)..... | 46 |
| 2.6.6 Mehrdimensionale Arrays..... | 33 | 4.3.4 Vertragswesen..... | 46 |
| 2.6.7 Initialisierung..... | 33 | 4.3.5 Detailkonzept..... | 46 |
| 2.6.8 Zuweisen von Arrays..... | 33 | 4.3.6 Realisieren & Testen..... | 47 |
| 2.6.9 Leeren von Arrays..... | 34 | 4.3.7 Einführung..... | 47 |
| 2.6.10 Eigenschaften und Methoden..... | 34 | 4.4 Vorteile des phasenweisen Vorgehens...48 | |
| 2.7 Datenfelder..... | 35 | 5 Referenzen..... | 49 |

1 ANSI C-Grundlagen

Dieser Teil des Dokuments soll eine kurze Zusammenfassung über ANSI-C bieten, ohne Anspruch auf Vollständigkeit. Es wird vor allem auf die Sprache selber (Syntax) eingegangen, die Funktionsbibliothek wird dann im letzten Teil noch kurz angeschnitten.

1.1 Hinweis zur Syntax

Die in diesem Dokument verwendete C-Syntax unterscheidet sich in manchen Punkten stark von der Syntax, welche im Modul 103 unterrichtet wurde. Das liegt daran, dass der Unterricht nicht auf dem offiziellen C99-Standard aufgebaut ist. Dieses Dokument bezieht sich hingegen so gut wie möglich auf diesen Standard.

1.2 Der Präprozessor

Der Präprozessor ist nichts weiteres als ein einfaches Textersetzungs-Werkzeug. Präprozessor-Befehle unterscheiden sich grundlegend von C-Anweisungen, sie beginnen z.B. immer mit einem Rautezeichen (#) und dürfen nicht beliebigen *Whitespace*¹ enthalten.

Eine Präprozessor-Anweisung wird mit dem Zeilenende abgeschlossen, eine C-Anweisung mit einem Strichpunkt (;).

Diese Textersetzungen werden vor dem Kompilierungsvorgang vorgenommen.

1.2.1 Die include-Anweisung

Mit der include-Anweisung kann man eine Quellcodedatei in eine andere einfügen.

```
#include "code.c"
```

So wird der gesamte Programmcode aus der Datei code.c in das File miteinbezogen, in welchem diese Anweisung steht. Die Schreibweise mit den Anführungs- und Schlusszeichen darf nur bei relativen Pfaden verwendet werden!

Wenn man Quellcodedateien verwenden will, die ein Compiler zur Verfügung stellt, so liegen diese oft in einem speziellen Verzeichnis. Damit man solche Verzeichnisse nicht auswendig kennen muss, werden dafür Umgebungsvariablen angelegt. Der Zugriff auf solche Quellcodedateien erfolgt mit den Zeichen < und >:

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

1.2.2 Die define-Anweisung

Wenn man ein bestimmtes Element im gesamten Programmcode ersetzen will, kann das mit der define-Anweisung definiert werden.

```
#define ELEMENT wert
```

Diese Anweisung durchsucht den gesamten Quelltext nach dem Wort ELEMENT und ersetzt dieses jeweils durch wert. Die Anweisung define bietet so eine Möglichkeit, mit Konstanten zu arbeiten. Weiter können mit define auch Makros definiert werden.

1.3 Das Hauptprogramm

Jedes C-Programm beginnt in der Funktion main. Diese muss immer so aussehen:

```
int main()
{
    // Programmcode
    return 0;
}
```

¹ Platz, der auf dem Bildschirm leer erscheint (Tabs, Leerzeichen und Zeilenumbrüche)

Die Funktion muss immer den Namen `main` tragen. Der Rückgabewert muss dabei immer `int` sein – die Angabe von `void` beim Rückgabebetyp verstösst gegen den C99-Standard und darf so nicht angewendet werden!

Der Funktionskopf darf sich lediglich in der Parameterliste unterscheiden, hierbei gibt es folgende drei Möglichkeiten:

```
int main()
int main(void)
int main(int argc, char *argv[])
```

Die ersten Beiden Funktionsköpfe unterscheiden sich in der Praxis nicht. Bei der dritten Variante ist es zusätzlich möglich, übergebene Parameter auszuwerten und entsprechend darauf zu reagieren.

Die `return`-Anweisung beendet die Funktion – und hiermit auch das Programm. In diesem Fall wird dem Betriebssystem 0 zurückgemeldet. Das bedeutet so viel wie, das Programm sei ordnungsgemäss abgelaufen.

1.4 Kommentare

Um die Funktionalität eines Programms zu verdeutlichen, kann man Kommentare einfügen. Kommentare haben keine bestimmte Syntax, der Text kann in jeder beliebigen Sprache verfasst sein und es können zudem alle Sonderzeichen verwendet werden.

Bei Kommentaren wird zwischen zwei Arten unterschieden:

1. einzeilige Kommentare
2. mehrzeilige Kommentare

Ein einzeiliger Kommentar beginnt mit zwei Schrägstrichen (`//`). Danach kann beliebiger Text bis ans Zeilenende verfasst werden. Die neue Zeile muss wieder in C-Syntax sein.

Mehrzeilige Kommentare hingegen, werden mit `/*` eingeleitet und müssen wieder mit `*/` beendet werden. Dazwischen kann auch beliebiger Text verfasst werden. Mit dieser Kommentar-Art können auch einzeilige Kommentare verfasst werden, dies ist aber in C eher ungebrauchlich.

Beispiele:

```
/*
in folgendem Programmabschnitt
findet eine Zuweisung statt.
*/

x = 943; // der Variable x wird die Zahl 943 zugewiesen
```

1.5 Variablen

Ein Programm besteht grundsätzlich aus Anweisungen und Daten. Diese Daten werden immer in C immer in Variablen abgelegt. Eine Variable ist ein Platzhalter für einen Wert, deren Wert sich während der Laufzeit eines Programmes ändern kann.

1.5.1 Datentypen

In C wird zwischen verschiedenen Datentypen unterschieden. Diese können verschiedene Arten von Werten aufnehmen.

In folgenden Datentypen können ganze Zahlen gespeichert werden.

| Datentyp | Speicherbedarf (in Byte) | Mögliche Werte |
|----------------------------|--------------------------|-----------------------------|
| <code>_Bool</code> | 1 | 0 und 1 |
| <code>char</code> | 1 | -128 bis 127 oder 0 bis 255 |
| <code>unsigned char</code> | 1 | 0 bis 255 |

| Datentyp | Speicherbedarf (in Byte) | Mögliche Werte |
|--------------------|---------------------------------|---|
| signed char | 1 | -128 bis 127 |
| int | 2 oder 4 | -32'768 bis 32'767 oder -2'147'483'648 bis 2'147'483'647 |
| unsigned int | 2 oder 4 | 0 bis 65'535 oder 0 bis 4'294'967'295 |
| short | 2 | -32'768 bis 32'767 |
| unsigned short | 2 | 0 bis 65'535 |
| unsigned long | 4 | 0 bis 4'294'967'295 |
| long long | 8 | -9'223'372'036'854'775'808 bis 9'223'372'036'854'775'807 |
| unsigned long long | 8 | 0 bis 18'446'744'073'709'551'615 |

Der Datentyp char wird vor allem für die Speicherung von ASCII-Werten verwendet. Dies geschieht nur aus dem Grund, dass eine char-Variable den ganzen ASCII-Zahlenbereich (0 bis 255) abdecken kann, jedoch keine Redundanzen aufweist. Der Typ unterscheidet sich also in der Praxis ausser dem Wertebereich nicht von den anderen ganzzahligen Typen.

Weiter gibt es Datentypen zur Speicherung von reellen Zahlen (Gleitkommazahlen), welche in folgenden Datentypen gespeichert werden können:

| Datentyp | Speicherbedarf (in Byte) | Mögliche Werte | Genauigkeit |
|-----------------|---------------------------------|-------------------------|--------------------|
| float | 4 | 1.2E-38 bis 3.4E+38 | 6 Stellen |
| double | 8 | 2.3E-308 bis 1.7E+308 | 15 Stellen |
| long double | 10 | 3.4E-4932 bis 1.1E+4932 | 19 Stellen |

1.5.2 Deklaration

Damit eine Variable in einem Programm verwendet werden kann, muss diese zuerst deklariert werden. Dies geschieht mit folgender Syntax:

```
[Datentyp] [Variablenname];
```

Es ist auch möglich, in einer Zeile mehrere Variablen des gleichen Typs zu deklarieren. Diese werden dann jeweils durch ein Komma voneinander getrennt.

```
[Datentyp] [Variable 1], [Variable 2], ..., [Variable n];
```

Es ist zudem möglich, dass man den Variablen gleich bei der Deklaration mit einem Startwert versieht, dies ist aber optional.

```
[Datentyp] [Variable 1] = [Wert 1], [Variable 2], [Variable 3] = [Wert 2];
```

Dazu einige Beispiele:

```
int a, b, c;
double d = 16.15, e, f = 512.0;
float g;
```

Der Variablenname muss folgende Bedingungen erfüllen:

1. Besteht aus einer Folge von Buchstaben (A bis Z und a bis z), Ziffern (0 bis 9) und Underscores (_)
2. Das erste Zeichen darf keine Ziffer sein
3. Es wird zwischen Gross- und Kleinschreibung unterschieden
4. in der Regel sind die ersten 31 Zeichen signifikant, längere Variablennamen sollten vermieden werden

Diese Bedingungen treffen nicht nur auf Variablen, sondern auch auf Konstanten, Enumeratoren, Funktionsnamen und Strukturen (sprich auf alle Bezeichner) zu.

1.5.3 Konstanten

Neben Variablen können auch Konstanten deklariert werden. Diese unterscheiden sich wie folgt von Variablen:

1. Konstanten können ihren Wert während der Laufzeit nicht ändern
2. Konstanten müssen bei der Deklaration auch gleich initialisiert werden!

Die Deklaration/Definition einer Konstanten erfordert das zusätzliche Schlüsselwort `const`. Die Syntax sieht folgendermassen aus:

```
const [Datentyp] [Konstante 1] = [Wert 1], [Konstante 2] = [Wert 2];
```

Beispiele:

```
const int a = 15, b = 17;  
const int c; // Fehler, kein Startwert!  
a = 19; // Fehler, a ist eine Konstante!
```

1.6 Ausdrücke

Mit einem Ausdruck bezeichnet man in C etwas, was einen Wert zurück gibt, welcher weiterverwendet werden kann, jedoch nicht muss. Dieser Wert ist immer von einem bestimmten Datentyp. Ausdrücke können sich aus Variablen, Konstanten (*literale*² und symbolische) und Funktions- sowie Makroaufrufe zusammensetzen.

```
16 * 22 + 64 // int  
1.0 + sin(x) // double  
59 // int
```

Eine einzelne Zahl ist ebenso ein Ausdruck, wie die beiden anderen. Der zurückgegebene Wert eines Ausdrucks kann in einer Variablen gespeichert werden, dazu verwendet man den Zuweisungsoperator.

```
int a = 16 * 22 + 64;  
double b = 1.0 + sin(x);
```

1.6.1 wahre und unwahre Ausdrücke

Ein Ausdruck kann zwar einen beliebigen Wert haben, oftmals muss aber nur zwischen zwei Zuständen unterschieden werden. Falls ein Ausdruck 0 ist, gilt er als unwahr – alle anderen Werte gelten als wahr.

Bei reellen Zahlen gelten alle Werte ausser der dezimalen 0 als wahr. Bei Referenztypen gelten alle Ausdrücke als wahr, die auf ein tatsächliches Objekt im Speicher verweisen.

1.7 Operatoren

Operatoren ermöglichen es, einzelne Ausdrücke mit einer bestimmten Logik zu verbinden, sodass der Rückgabewert den gestellten Anforderungen entspricht.

Es gibt mehrere Arten von Operatoren, die wichtigsten werden in den folgenden Abschnitten erklärt.

² Ein expliziter Wert, kein Bezeichner (z.B. 4,53, „Das ist ein Test“, usw.)

1.7.1 Arithmetische Operatoren

| Operator | Bedeutung | Beispiel | Ergebnis |
|-----------------|-------------------------------|-----------------|---------------------------|
| * | Multiplikation | $x * y$ | Produkt von x und y |
| / | Division | x / y | Quotient aus x und y |
| % | Modulo-Division (Restbildung) | $x \% y$ | Divisionsrest aus x und y |
| + | Addition | $x + y$ | Summe aus x und y |
| - | Subtraktion | $x - y$ | Differenz von x und y |
| + (unär) | Plus-Vorzeichen | $+x$ | Wert von x |
| - (unär) | Minus-Vorzeichen | $-x$ | Negation aus x |

Das Plus-Vorzeichen hat eigentlich keinen praktischen Nutzen, da bei weglassen des Operators immer der Wert von x genommen wird. Bei der Negation spielt es keine Rolle ob ein vorzeichenbehafteter oder -freier Datentyp verwendet wird, es wird immer dessen Gegenteil verwendet (aus 77 wird -77, aus 118 wird -118, ...).

1.7.2 Zuweisungsoperatoren

| Operator | Bedeutung | Beispiel | Ergebnis |
|-----------------|----------------------------|-------------------|-------------------------|
| = | Einfache Zuweisung | $x = y$ | x erhält den Wert von y |
| [op]= | Zusammengesetzte Zuweisung | $x \text{ += } y$ | x wird um y erhöht |

Bei einer einfachen Zuweisung muss der Ausdruck links vom Gleichheitszeichen immer eine Variable sein, der Ausdruck rechts des Gleichheitszeichen kann ein beliebiger Ausdruck sein. In einer Zuweisung können auch mehrere Gleichheitszeichen vorkommen. Beispiel:

```
a = b = 66; // b erhält den Wert 66, a erhält den Wert von b (66)
```

Diese Anweisung ist äquivalent mit:

```
b = 66;
a = b; // oder a = 66
```

Bei der Zusammengesetzten Zuweisung kann für [op] ein beliebiger arithmetischer Ausdruck eingesetzt werden, der Modulo-Operator ist allerdings ungültig:

```
a += 15; // a = a + 15;
a -= 12; // a = a - 12;
a /= 2; // a = a / 2;
a *= 5; // a = a * 5;
a %= 2; // Fehler
```

In einer Variablen eines ganzzahligen Typs können auch Zeichen gespeichert werden. Dazu sollte ausschliesslich der Datentyp char verwendet werden. Damit man hier nicht immer auf die ASCII-Liste zurückgreifen muss, gibt es eine einfachere Möglichkeit:

```
a = 'B'; // 'B' steht für den ASCII-Code 66
```

1.7.3 In- und Dekrementierung

| Operator | Bedeutung | Beispiel | Ergebnis |
|----------|-----------|------------|--|
| ++ | Inkrement | x++ ++y | x wird um 1 erhöht y wird um 1 erhöht |
| -- | Dekrement | x-- --y | x wird um 1 verringert y wird um 1 verringert |

Beide Operatoren haben zwei verschiedene Notationen. Wenn der Operator vor dem Variablenname steht, so spricht man von der so genannten Präfix-Notation. Ansonsten handelt es sich um die Postfix-Notation. Für die betroffene Variable macht dies keinen Unterschied, der gesamte Ausdruck erhält jedoch einen anderen Wert:

```
int a = 10, b = 20;
int x = ++a; // x erhält 11 (zuerst a inkrementieren, dann zuweisen)
int y = b++; // y erhält 20 (zuerst zuweisen, dann b inkrementieren)
```

1.7.4 Vergleichsoperatoren

| Operator | Bedeutung | Beispiel | Ergebnis (1 = wahr, 0 = falsch) |
|----------|----------------|----------|---|
| < | Kleiner als | x < y | Falls x kleiner als y ist 1, sonst 0 |
| <= | Kleiner gleich | x <= y | Falls x kleiner gleich y ist 1, sonst 0 |
| > | Grösser als | x > y | Falls x grösser als y ist 1, sonst 0 |
| >= | Grösser gleich | x >= y | Falls x grösser gleich y ist 1, sonst 0 |
| == | gleich | x == y | Falls x gleich y ist 1, sonst 0 |
| != | ungleich | x != y | Falls x ungleich y ist 1, sonst 0 |

1.7.5 Logische Operatoren

| Operator | Bedeutung | Beispiel | Ergebnis (1 = wahr, 0 = falsch) |
|----------|---------------|----------|----------------------------------|
| && | Logisches AND | x && y | 1, falls x und y ungleich 0 sind |
| | Logisches OR | x y | 1, falls x oder y ungleich 0 ist |
| ! | Logisches NOT | !x | 1, falls x 0 ist |

Bei den logischen Operatoren gilt, jeder Wert der sich von 0 unterscheidet ist „wahr“. Ein Ausdruck ist „unwahr“, sobald er exakt 0 ist.

1.7.6 Bit-Operatoren

Diese Operatoren können nur auf ganze Zahlen angewendet werden.

| Operator | Bedeutung | Beispiel | Ergebnis (1 = wahr, 0 = falsch) |
|----------|---------------|----------|---------------------------------|
| & | Bitweises AND | x & y | 1, falls x und y 1 sind |

| Operator | Bedeutung | Beispiel | Ergebnis (1 = wahr, 0 = falsch) |
|----------|---------------|--------------|---|
| | Bitweises OR | $x \mid y$ | 1, falls x oder y 1 ist |
| ^ | Bitweises XOR | $x \wedge y$ | 1, falls x oder y 1, aber nicht beide 1 sind |
| ~ | Bitweises NOT | $\sim x$ | 1, falls x 0 ist |
| << | Links-shift | $x \ll y$ | x wird um y Positionen nach links verschoben |
| >> | Rechts-shift | $x \gg y$ | x wird um y Positionen nach rechts verschoben |

Bit-Operatoren sind keinesfalls äquivalent zu den logischen Operatoren und dürfen deshalb nicht verwechselt werden! Ein logischer Operator betrachtet immer den ganzen Ausdruck, während die Bit-Operatoren jedes Bit einzeln bearbeiten!

Beispiele für die shift-Operatoren:

```
x = 5; // dezimale 5 entspricht binärer 101
x = x << 1; // aus 101 wird 1010, was einer dezimalen 10 entspricht
x = x >> 2; // aus 1010 wird 10, was einer dezimalen 2 entspricht
```

Beim links-shift-Operator wird von rechts immer mit 0en aufgefüllt, die Bits links gehen dementsprechend verloren.

Der rechts-shift sollte nur auf vorzeichenlose Ganzzahlen angewendet werden. Dieser füllt Ganzzahlen von links mit 0en auf, die Bits rechts gehen verloren. Bei vorzeichenbehafteten Ganzzahlen ist es möglich, dass 1en nachgeschoben werden. Dies kann zu schwer lokalisierbaren Fehlern führen.

1.7.7 Operatoren für den Speicherzugriff

| Operator | Bedeutung | Beispiel | Ergebnis |
|----------|-----------------|-------------------|--|
| & | Adressoperator | $\&x$ | Speicheradresse von x |
| * | Verweisoperator | $*x$ | Objekt oder Funktion, auf das x zeigt |
| [] | Arrayelement | $x[n]$ | Das n. Element im Array |
| . | Strukturzugriff | $x.a$ | Komponente a in der Struktur x |
| -> | Strukturzugriff | $x \rightarrow a$ | Komponente a der Struktur, auf die x zeigt |

Weitere Erklärungen zu den Speicherzugriffsoperatoren kann man im Abschnitt Zeiger nachlesen.

1.7.8 sonstige Operatoren

| Operator | Bedeutung | Beispiel | Ergebnis |
|----------|--------------------|----------------------|--|
| () | Funktionsaufruf | $\max(x, y)$ | Aufruf einer Funktion |
| ([Typ]) | Expliziter Cast | $x = (\text{int})y;$ | In x wird der Ganzzahlanteil von y gespeichert |
| sizeof | Grösse | $\text{sizeof}(x)$ | Anzahl der Byte, die x belegt |
| ?: | Bedingte Bewertung | $x ? y : z$ | Falls x wahr ist y, sonst z |
| , | Sequenzoperator | x, y, z | x vor y und z auswerten |

Bei der bedingten Bewertung kann mit einer Bedingung gesteuert werden, welcher Wert der Ausdruck haben soll. Ein Beispiel:

```
int a = 5, b = 7;
char z = ' ';
z = (a < b) ? 'b' : 'a'; // z = 'b', wenn Bedingung zutrifft, sonst 'a'
```

Diesen Operator sollte man nur sparsam einsetzen, die if-Anweisung bildet eine vielseitigere Alternative. Es ist jedoch von Vorteil, wenn man den Operator lesen und verstehen kann.

Der Sequenzoperator wird bei Funktionsaufrufen, bei der Deklaration mehrerer Variablen sowie anderen Programmteilen verwendet. Dieser Operator ist kein Operator im eigentlichen Sinne, sondern eher ein syntaktisches Trennzeichen.

1.8 Anweisungen

Mithilfe von Anweisungen kann man auszuführende Aktionen festlegen. Dies kann ein Funktionsaufruf, eine Verzweigung oder eine Kombination aus mehreren Konstrukten sein.

Eine Anweisung wird immer mit einem Semikolon (;) abgeschlossen, die Ausnahme bilden Blockanweisungen.

```
x = max(a, b);
x += a % b;
```

1.8.1 Codeblöcke

Ein Codeblock wird von geschweiften Klammern ({, }) umgeben und fasst keinen, einen oder mehrere Anweisungen zusammen. In C werden alle Funktionseinheiten (Schleifen, Verzweigungen, Funktionen, ...) durch Blöcke dargestellt.

Deklarationen sollten in einem Block zuoberst stehen, übrige Anweisungen danach. Dies ist zwar seit C99 nicht mehr zwingend, fördert jedoch die Übersichtlichkeit des Programms.

1.9 Verzweigungen

Oftmals will man eine Anweisungen oder einen ganzen Block nur unter einer bestimmten Bedingung ausführen. Siehe auch den Teil AUSDRÜCKE. Diese Bedingung ist ein normaler Ausdruck, der 0 oder 1 zurück gibt

1.9.1 Verzweigungen mit if

Zur Prüfung von Ausdrücken wird das Schlüsselwort if verwendet.

```
if([Bedingung])
    [Anweisung];

if([Bedingung])
{
    [Anweisung 1];
    [Anweisung 2];
    ...
    [Anweisung n];
}
```

Einer if-Anweisung darf nur eine einzige Anweisung folgen! Dies kann eine einfache Anweisung oder auch eine Blockanweisung sein. Alle weiteren Anweisungen werden nicht abhängig von der Bedingung, sondern in jedem Fall ausgeführt.

Die Zugehörige Anweisung wird immer dann ausgeführt, wenn der Ausdruck wahr (d.h. ungleich 0) ist.

1.9.2 Else if

Schlägt eine if-Anweisung fehl, so wird die bedingte Anweisung ignoriert. Mit else if ist es möglich, auf diese Fälle zu reagieren. Es kann eine weitere Bedingung und Anweisung angegeben werden. Else if wird nur ausgewertet, wenn nicht vorher bereits die if-Anweisung ausgeführt wurde.

```
if([Bedingung 1])
    [Anweisung 1];
else if([Bedingung 2])
    [Anweisung 2];
```

Es können auch mehrere else if Anweisungen aufgeführt werden. Dies sollte man aber aus Gründen der Übersichtlichkeit so gut wie möglich vermeiden, bzw. in Grenzen halten. Man sollte statt mehreren else if Anweisungen die switch-case Anweisung benötigen, falls dies für die jeweilige Bedingung möglich sein sollte.

1.9.3 Else

Auf Fehlschläge der if-Anweisung muss nicht immer ein weiterer Ausdruck überprüft werden. Oftmals genügt es immer eine Anweisung auszuführen wenn if fehlgeschlagen ist. Else kann auch nach mehreren else if Anweisungen stehen.

```
if([Bedingung 1])
    [Anweisung 1];
else if([Bedingung 2])
    [Anweisung 2];
else if([Bedingung 3])
    [Anweisung 3];
else
    [Anweisung 4];
```

Else muss dabei immer ganz am Schluss stehen. Sobald ein else in einer Verzweigung auftritt, so kann man sicher sein, dass mindestens eine Anweisung ausgeführt wird.

1.9.4 Switch case

Sobald mehr als eine else if Anweisung benötigt werden sollte, ist das Ausweichen auf die switch case Anweisung ratsam. Dies ist jedoch nicht immer möglich, da mit der switch case Anweisungen nur konstante Ausdrücke ausgewertet werden können! Dies umfasst also nur literale und symbolische Konstanten - Variablen können nicht ausgewertet werden!

Die Syntax sieht folgendermassen aus:

```
switch([Ausdruck 1])
{
    case [Konstante 1]:
        [Anweisung 1];
        [Anweisung 2];
        break;
    case [Konstante 2]:
        [Anweisung 3];
        break;
    default:

```

Dem Schlüsselwort switch muss in Klammern ein Ausdruck folgen. Danach folgt ein Block. Im Block selber wird die Fallunterscheidung mit dem Schlüsselwort case gemacht, welchem ein konstanter Ausdruck folgen muss. Ein Doppelpunkt leitet nun einen pseudo-Codeblock ein. Diese Codeblöcke müssen unbedingt mit der Anweisung break abgeschlossen werden! Das liegt an der Funktionsweise der switch-case-Anweisung:

Der Ausdruck wird mit jeder Konstanten verglichen. Nun gelangt die Programmausführung in den ersten pseudo-Block und arbeitet sich bis zum Ende durch. Die case-Anweisungen dienen dabei lediglich als Sprungmarken und trennen keine Codeblöcke ab!

Manchmal ist es wünschenswert, dass alle folgenden Codeblöcke ausgeführt werden. Dies sollte man mit Kommentaren verdeutlichen.

1.10 Schleifen

Um Anweisungen mehrmals hintereinander auszuführen, greift man auf Programmkonstrukte zurück, die als Schleifen bezeichnet werden.

Es gibt drei Arten von Schleifen (while, do-while und for), dazu kommt eine pseudo-Schleifeninvariante mit dem Schlüsselwort goto, welche in folgendem Abschnitt erklärt wird.

1.10.1 Goto

Das goto-Schlüsselwort ist vor allem ein Überbleibsel aus Programmiersprachen wie Basic und Assembler. Die Verwendung von goto kann zu sehr unübersichtlichem Programmcode führen und sollte niemals verwendet werden! Die Anweisung ist wohl nur noch aus Kompatibilitätsgründen in den Standard C99 übernommen worden. Da man in älteren Programmen aber noch öfters auf diese Anweisung treffen könnte, wird diese hiermit auch noch erläutert.

Die goto-Anweisung setzt sich aus zwei Teilen zusammen; die Sprungmarke und die Sprunganweisung. Die Sprungmarke muss den Namen eines gültigen Bezeichners, der noch nicht verwendet wird, tragen. Es folgt ein Doppelpunkt. Sobald man zu dieser Sprungmarke wechseln will, wird die goto-Anweisung angewendet. Es kann sowohl auf-, wie auch abwärts gesprungen werden:

```
int n = 0;
top:
    printf("%i\n", n);
    n++;
    if(n <= 10)
        goto top;
```

In obigem Programm werden alle Zahlen von 0 bis 10 ausgegeben. Sobald n grösser als 10 ist, wird die goto-Anweisung nicht mehr ausgeführt, das Programm fährt nach if weiter.

1.10.2 Die while-Schleife

Die while-Schleife ist kopfgesteuert. Das bedeutet, dass der Ausdruck zu Beginn geprüft wird, die Schleifenanweisung wird ausgeführt, solange der Ausdruck wahr ist.

Die Syntax sieht folgendermassen aus:

```
while ([Ausdruck])
    [Anweisung];
```

Wie immer wird nur die erste Anweisung unmittelbar nach dem Schleifenkopf ausgeführt. Diese kann natürlich auch eine Blockanweisung sein:

```
while ([Ausdruck])
{
    [Anweisung 1];
    [Anweisung 2];
    ...
    [Anweisung n];
}
```

1.10.3 Die do-while-Schleife

Hier handelt es sich um eine fussgesteuerte Schleife. Das bedeutet, dass zuerst die Schleifenanweisung ausgeführt und erst dann die Ausdrucksprüfung stattfinden. Ist dieser Ausdruck wahr, so schreitet das Programm bei der Schleifenanweisung weiter.

Syntax:

```
do
    [Anweisung];
while([Ausdruck]);
```

1.10.4 Die for-Schleife

Die for-Schleife ist die komplexeste Schleife. Sie fasst Schritte im Kopf zusammen, die oftmals bei den while-Schleifen in der Schleifenanweisung ausgeführt werden.

```
for([Initialisierung]; [Ausdruck]; [Anweisung 1])
    [Anweisung 2];
```

Die Initialisierung wird beim ersten Betreten der Schleife ausgeführt. Es ist auch möglich, bei der Initialisierung eine Variable zu deklarieren, dies entspricht aber nicht dem C99-Standard.

Der Ausdruck nimmt die Funktion analog der anderen Schleifen ein.

Die Anweisung 1 steht für eine beliebige Anweisung, diese wird aber für eine Erhöhung eines Zählers verwendet.

Sowohl bei der Initialisierung wie auch bei der Anweisung 1 kann man beliebig viele Anweisungen anfügen, diese müssen dann jeweils durch den Sequenzoperator (,) voneinander getrennt werden. Es ist zudem auch möglich, dass man den Schleifenkopf leer lässt, die Semikolons müssen aber trotzdem aufgeführt werden.

Ein Beispiel:

```
int a, b;
for(a = 0, b = 0; a < 10 && b < 10; a++, b++)
    printf("%i:%i\n", a, b);
```

1.10.5 break

Dieses Schlüsselwort wurde bereits bei der switch-Anweisung verwendet, in welcher es einen Codeblock beendet.

Bei den Schleifen hat das break-Schlüsselwort allerdings eine andere Funktion. Es dient dazu, die Schleife unabhängig von der Schleifenbedingung zu beenden.

Ein Beispiel:

```
int n = 0;
do
{
    n++;
    if(n > 10)
        break;
} while(1);
```

Obwohl die Schleife als Endlosschleife deklariert wurde (1 ist immer wahr), wird sie beendet, sobald n grösser als 10 geworden ist.

Für das Beenden der Schleife sollte man möglichst nur die Schleifenbedingung einsetzen, da es bei grösseren Konstrukten schnell unübersichtlich werden kann. Manche Programmierer vergleichen die break-Anweisung auch mit goto, da beide Anweisungen den Code einfach unterbrechen und an eine andere Stelle springen können.

1.10.6 continue

Will man vom Schleifenrumpf wieder in den Kopf zurückkehren, ohne den Rumpf zu Ende auszuführen, so kann man die continue-Anweisung verwenden:

```
int a = -5;
for(; a <= 5; a++)
{
    if(a == 0)
        continue;
```

```
    printf("%i\n", a);  
}
```

Obwohl die printf-Anweisung nicht der if-Abzweigung untersteht, wird niemals 0 ausgegeben. Dies hat den Grund, dass die Schleife unterbrochen wird, sobald die Variable a den Wert 0 hat.

Für die continue-Anweisung gilt das selbe wie für break – sie sollte nur sparsam eingesetzt werden. Es gilt die Regel; die Anweisungen break, continue und goto müssen niemals zwingend eingesetzt werden, man kann die Funktionalität mit den anderen Programmkonstrukten (if, while, ...) sauber nachstellen. Es schadet aber nicht, wenn man genannte Anweisungen kennt und in einem fremden Programmcode nachvollziehen kann.

1.11 Eigene Datentypen

Sobald man komplexere Datenstrukturen realisieren will, reichen die einfachen Datentypen nicht mehr aus. Dafür bietet C zwei Möglichkeiten für unterschiedliche Zwecke – die Strukturen und die Enumeratoren.

1.11.1 Enumeratoren

Eine Funktion hat oftmals numerische Rückgabewerte, welche verschiedene Bedeutungen haben. Mit diesen Nummern kann man meistens wenig anfangen, man müsste sich in die Funktion einlesen bzw. deren Dokumentation lesen, falls vorhanden.

Wenn eine Funktion nur eine Hand voll Rückgabewerte liefern soll, welche eine unterschiedliche Bedeutung haben, so ist die Verwendung eines Enumerators angebracht.

Ein Enumerator ist nichts weiter als ein Behälter für Konstanten, welcher als Datentyp benutzt werden kann. Statt sich weiter mit abstrakten Zahlen herumzuschlagen, hat man nun eine textuelle Beschreibung dieser Rückgabewerte.

Enumeratoren werden wie folgt deklariert:

```
enum [Bezeichnung]  
{  
    [Konstante 1] = 0,  
    [Konstante 2] = 1,  
    ...,  
    [Konstante n] = n  
};
```

Die Nummernvergabe ist dabei fakultativ, da man diese später im Programm nicht mehr benötigt. Zur besseren Verständlichkeit ein Beispiel:

```
enum AMPEL  
{  
    ROT = 1, // Nummerierung ab 1  
    ORANGE,  
    GRUEN  
};
```

Wenn man das erste Element mit einer Zahl initialisiert, so werden die nachfolgenden automatisch von dieser Zahl an durchnummeriert. Die Verwendung im Programm könnte dann etwa so aussehen:

```
enum AMPEL ampel;  
ampel = ROT; // äquivalent zu ampel = 1  
printf(„%i“, ampel); // gibt 1 aus
```

Ein Enumerator kann nur Ganzzahlen aufnehmen, Fließkommazahlen sind nicht möglich. Die einzelnen Elemente eines Enumerators sind Konstanten vom Typ int. Konstanten sollten immer gross geschrieben werden, Kleinschreibung macht aber keine Probleme.

1.11.2 Strukturen

Um komplexe Datenstrukturen wie verkettete Listen und Bäume zu realisieren, reichen die primitiven Datentypen nicht aus. Damit man mehrere Informationen in einer Variablen speichern kann, sind deshalb Strukturen (oft auch als „records“ bezeichnet) von Nöten.

Diese werden folgendermassen deklariert:

```
struct [Strukturname]
{
    [Datentyp] [Variablenname 1];
    [Datentyp] [Variablenname 2];
    ...;
    [Datentyp] [Variablenname n];
} [Strukturvariable 1], [Strukturvariable 2];
```

Dem Schlüsselwort struct folgt der Name der Struktur. Im Rumpf folgen dann die einzelnen Elemente der Struktur. Dies können Variablen primitiver Datentypen oder andere Strukturen (ja sogar die eigene Struktur!) sein. Im Gegensatz zur Enumeration folgt jedem Element ein Semikolon. Die Variablendeklaration erfolgt wie die Deklaration in einer normalen Funktion, nur dass man keinen Startwert belegen darf.

Nach dem abgeschlossenen Deklarationsblock hat man die Möglichkeit Instanzen der neu erstellten Struktur zu definieren, dies ist aber nicht zwingend.

Ein Beispiel für eine einfache Struktur:

```
struct processor
{
    int taktfrequenz;
    float preis;
};
```

Hier wurde auf die Deklaration verzichtet, diese wird im Hauptprogramm vorgenommen. Wenn man eine Variable vom Typ processor deklarieren will, muss man das Schlüsselwort struct ebenfalls voranstellen. Einige Compiler verlangen dies zwar nicht, ein Weglassen des Schlüsselwortes ist jedoch ein Verstoß gegen den offiziellen Standard C99.

So sieht die Deklaration aus:

```
struct processor pentium4, pentium 3;
```

Wie bei normalen Variablen kann man also auch bei den Strukturen mehrere in einer Zeile deklarieren. Das Belegen eines Startwerts macht dabei keinen Sinn, der Zugriff muss auf die einzelnen Elemente der Struktur erfolgen. Dies geschieht mit dem Strukturzugriffsoperator:

```
pentium4.taktfrequenz = 3200;
pentium4.preis = 299.95;
pentium3.taktfrequenz = 700;
pentium3.preis = 75.95;
```

Das Lesen von Strukturwerten erfolgt äquivalent dazu:

```
int x = pentium4.taktfrequenz;
double y = pentium3.preis;
```

1.12 Arrays

Verschiedene Datentypen zu einem neuen zusammenfassen funktioniert mit Strukturen. Wenn man aber grosse Anzahlen des gleichen Datentyps verwenden will, sind Strukturen eher ungeeignet. Dafür gibt es in C die so genannten Arrays (auch „Felder“ und „Vektoren“ genannt), welche eine beliebige Anzahl³ des gleichen Typs (sogar Strukturen!) logisch zusammenfassen.

³ Nur durch den Platz des Speichers beschränkt

1.12.1 Deklaration

Der Unterschied zwischen der Deklaration einer normalen Variable und eines Arrays liegt darin, dass man bei einem Array angeben muss, wie viele Werte dieses Typs aufgenommen werden können:

```
[Datentyp] [Variablenname][[Grösse]];
```

Die Anzahl wird dabei in den Eckklammern angegeben, muss vom Typ unsigned int und konstant sein. Ein Beispiel:

```
char vorname[40]; // der Vorname kann aus 40 Zeichen bestehen...
char nachname[50]; // ... der Nachname aus 50
```

Der Speicherverbrauch ist dabei das Produkt aus dem Speicherverbrauch des angegebenen Datentyps und der angegebenen Arraygrösse.

Ein Array ist eine sogenannte „statische Datenstruktur“. Das heisst, dass sich die Grösse des Arrays während der Laufzeit nicht verändern kann.

1.12.2 Initialisierung

Es ist empfehlenswert jedes Array zu initialisieren, da der Wert der einzelnen Elemente nach der Deklaration nicht klar definiert, sondern compilerabhängig ist. Dies könnte bei Lesezugriffen zu Problemen führen.

Ein Array wird wie folgt bei der Deklaration initialisiert:

```
[Datentyp] [Variablenname][[Grösse]] = {[Wert 1], ..., [Wert n] };
```

Die geschweiften Klammern müssen dabei die entsprechende Anzahl und den richtigen Datentyp aufweisen. Ausserdem müssen die Elemente ein konstanter Ausdruck sein. Es empfiehlt sich, bei dieser Initialisierung auf eine Grössenangabe zu verzichten.

Ein Beispiel:

```
int zahlen[] = {25, 97, 46, 21, 75}; // Grösse entfällt
```

1.12.3 C-Strings

Der Datentyp char bildet bei den Arrays eine Besonderheit. Character werden meistens dazu benutzt, ASCII-Zeichen aufzunehmen. Ein char-Array ist stellvertretend für eine Zeichenkette, oftmals wird dafür der Ausdruck „C-String“ verwendet.

Die Initialisierung kann gleich erfolgen, wie bei den übrigen Datentypen. Hier gibt es allerdings ein kleines Problem: Für die Ein- und Ausgabe von C-Strings wird immer eine Endmarkierung (\0) benötigt. Diese müsste bei oben erwähnter Initialisierung immer aufgeführt werden. Dies könnte schnell einmal vergessen gehen. Darum ist es möglich, einem Array eine Zeichenkette zuzuweisen:

```
char vorname[] = "Hans"; // Arraygrösse = 5 (4 Zeichen + Endmarke)
```

Diese Anweisung ist äquivalent zu:

```
char vorname[] = {'H', 'a', 'n', 's', '\0'};
```

Die erste Variante ist wesentlich kürzer und ausserdem auch sicherer und sollte somit bevorzugt werden.

1.12.4 Zugriff

Der Zugriff auf ein einzelnes Arrayelement erfolgt mit Eckklammern.

```
Array[index] = [Wert]; // Wert ins Array schreiben
[Variable] = Array[index]; // Wert aus Array lesen
```

Der angegebene Index kann in einem Bereich zwischen 0 und der bei der Deklaration angegebenen Anzahl – 1 sein. Angenommen, man gibt bei der Deklaration den Wert 46 an, kann man auf die Elemente 0 bis 45 zugreifen (insgesamt 46 Elemente), auch hier sind nur natürliche Zahlen möglich. Der Zugriff auf das Element 46 könnte zu einem Speicherzugriffsfehler führen.

Der Programmierer muss immer aufpassen, dass er sich im gültigen Bereich befindet. Bei C-Strings ist dies etwas einfacher, da diese immer mit '\0' abgeschlossen werden müssen und man das Ende somit gut erkennen kann.

1.12.5 Dimensionen

Ein Array hatte bisher immer nur eine Dimension, man kann es sich als einfache Liste vorstellen. Es ist jedoch möglich, ein Array mehrdimensional aufzubauen. D.h. bei zwei Dimensionen muss man sich das Array nicht mehr als Liste, sondern eher als Tabelle oder Rechteck vorstellen. Ein dreidimensionales Array stünde dann für einen Quader.

Alles was über drei Dimensionen geht sollte vermieden werden, da sich das abstrakte Denkvermögen der meisten Menschen auf drei Dimensionen beschränkt.

Die Deklaration erfolgt mit mehreren Eckklammern:

```
[Datentyp] [Variablenname][[Grösse D1]][[Grösse D2]];
```

Die gesamte Grösse setzt sich aus Grösse D1 * Grösse D2 zusammen. Die Initialisierung kann gleich erfolgen, wie bei den eindimensionalen Arrays:

```
int zahlen[][2] = {1, 2, 3, 4};
```

Wie hier zu sehen ist, darf nur die erste Grösse weggelassen werden, die zweite ist zwingend.

Beim Zugriff auf ein Element müssen beide Indizes angegeben werden:

```
int x = zahlen[0][1];  
zahlen[1][0] = 945;
```

Wird der zweite Index mitsamt Klammern weggelassen, wird immer auf das 0. Element der aktuellen Dimension zugegriffen. Es ist aber sauberer, wenn man immer beide Indizes explizit angibt.

1.13 Zeiger

Zeiger (oder englisch „pointer“) sind das mächtigste Element von C, sie ermöglichen komplexe Datenstrukturen und eine enorme Ausführungsgeschwindigkeiten. Sie sind jedoch nicht ganz einfach in der Anwendung, Fehler in der Handhabung mit Zeigern können zu schweren Abstürzen führen.

Ein Zeiger kann eine Adresse im Speicher aufnehmen, hinter welcher sich eine Variable oder eine Funktion befinden kann. Ein Zeiger muss immer den gleichen Typ haben, wie das Element auf das er zeigt. In Wirklichkeit ist aber jeder Zeiger vom Datentyp int, da Speicheradressen als Ganzzahlen angegeben werden.

1.13.1 Deklaration

Ein Zeiger wird sehr ähnlich wie eine gewöhnliche Variable deklariert. Es gibt nur den Unterschied, dass vor dem Variablennamen ein Stern (*) stehen muss:

```
[Datentyp] *[Variablenname] = [Initialisierungswert];
```

Dazu ein Beispiel:

```
int *f, *g;
```

Zeiger können auch in derselben Zeile wie normale Variablen des gleichen Typs deklariert werden. Dabei kann auch zwischen Zeigern und normalen Werten abgewechselt werden.

```
int a = 0, b = 0, *c, d = 0, *e;
```

Falls man dem Zeiger nicht schon bei der Deklaration auf eine Variable zeigen lassen will, sollte man ihn unbedingt mit dem Wert 0 belegen. Es kommen nur die Werte 0, oder direkt eine Adresse einer Variablen in Frage – andere Werte führen zu Fehlern!

1.13.2 Initialisierung

Zeiger können wie oben beschrieben auf Speicherzellen verweisen. Dies ist möglich, indem in einem Zeiger die Nummer der Speicherzelle angegeben wird, auf die gezeigt werden soll. Da man den Inhalt

seines Arbeitsspeichers aber in der Regel nicht kennt, macht es wenig Sinn diese Speicheradressen direkt als Zahlen in den Code einzufügen. Die Speicherzelle könnte möglicherweise bereits verwendet werden oder gar nicht existieren.

```
float *p = 194324; // sollte vermieden werden
```

Eine Ausnahme bildet der numerische Wert 0. Dieser steht für einen Zeiger ins „leere“, es wird nicht wirklich auf den Speicher verwiesen.

Mit einem Zeiger der auf 0 zeigt kann man nichts anfangen, er muss zuerst auf eine Variable verweisen. Zeiger können bekanntlich nur Speicheradressen aufnehmen. Damit man an die Speicheradresse einer Variablen herankommt, verwendet man den Adressoperator (&). Dieser gibt die Speicheradresse zurück, welche man nun in einer Zeigervariable abspeichern kann.

```
int a = 5, *p = 0;
p = &a;
```

Der Zeiger p (vom Datentyp int) verweist nun auf die Variable a (ebenfalls ein int).

1.13.3 Zugriff

Der eigentliche Wert eines Zeigers ist eigentlich selten bis nie von Interesse, wichtiger ist der Wert der Variablen, auf welche der Zeiger verweist. Um auf den Wert zuzugreifen, auf welchen der Zeiger verweist, kommt erneut der Stern zum Einsatz:

```
int a = 5, *p = 0;
p = &a;
printf(„%i“, p); // gibt die Speicheradresse von a aus
printf(„%i“, *p); // gibt 5 aus
```

Sobald der Stern dafür eingesetzt wird, um auf den Wert hinter einen Zeiger zuzugreifen, wird dieser als Dereferenzierungsoperator bezeichnet.

Der Wert kann mithilfe des Dereferenzierungsoperator nicht nur gelesen, sondern auch überschrieben werden:

```
int a = 5, *p = &a;
printf(„%i“, *p); // gibt 5 aus
*p = 16
printf(„%i“, *p); // gibt 16 aus
printf(„%i“, a); // gibt auch 16 aus - der Zeiger hat auf a zugegriffen!
```

Ein Zeiger kann während seiner Lebensdauer auf verschiedene Variablen zeigen, d.h. ihm kann während der Programmausführung eine neue Speicheradresse zugewiesen werden.

```
int a = 5, b = 7, *p = &a;
printf(„%i“, *p); // gibt 5 aus
p = &b;
printf(„%i“, *p); // gibt 7 aus
```

1.13.4 Merkregel

Der Dereferenzierungsoperator (*) wird bei der Initialisierung der Pointervariable, sowie beim Lese- und Schreibzugriff auf einen Pointer benötigt. Bei einem Schreibzugriff auf einen Pointer ohne den Dereferenzierungsoperator muss eine Adresse angegeben werden. Sobald aber der Dereferenzierungsoperator verwendet wird, erfolgt der Zugriff auf den Wert, auf den die Pointervariable verweist!

```
int *a, b;
a = &b; // Zugriff auf die Adresse
*a = 50; // Zugriff auf den Wert „hinter“ dem Pointer
```

Bei der Initialisierung muss der Dereferenzierungsoperator immer angegeben werden.

```
int a = 15;
int *b = &a;
int *c = 0;
```

Zur Erinnerung, neben einer Variablenadresse darf ein Zeiger nur mit dem Wert 0 belegt werden, alle anderen Werte führen zu Problemen!

1.13.5 Zeiger auf Arrays

Zeiger können nicht nur auf eine einzelne Variable zeigen, man kann sie auch zur Referenzierung auf ganze Arrays verwenden. Zeiger werden oft dazu benutzt, um durch ein Array durch zu iterieren.

Die Instantiierung von einem Zeiger auf ein Array erfolgt wie bei einer normalen Variable:

```
int field[3] = {16, 85, 34};
int *p = field[0]; // auf das erste Element verweisen
```

Um auf das erste Element zu verweisen, muss der Index nicht zwingend angegeben werden. Der Arrayname steht immer für eine Referenz auf das Element 0.

Das Interessante an Zeiger auf Arrays ist, dass man durch ein Array iterieren kann. Um auf das nächste Element zu kommen, muss nur der Zeiger um 1 erhöht werden. Dabei spielt es keine Rolle, ob das Array vom Typ double oder int ist, beim Inkrementieren wird immer auf das nächste Arrayelement gesprungen.

```
int field[3] = {16, 85, 34};
int *p = field[0]; // *p = 16
p++; // *p = 85
p++; // *p = 34
p++; // Fehler, Überschreiten der Arraygrenze!!!
```

Obiges Beispiel zeigt, dass man immer darauf achten sollte, dass man nicht über die Arraygrenze hinaus iteriert.

Der Zugriff auf den Wert hinter dem Zeiger erfolgt wie bei Zeigern auf normale Variablen. Ob man mit dem Zeiger durch ein Array iteriert oder direkt auf eine einfache Variable verweist – im Hintergrund wird immer nur auf eine Variable verwiesen.

Ein Array kann auch rückwärts durch iteriert werden, dazu muss der Zeiger jeweils um 1 verringert werden:

```
int field[3] = {16, 85, 34};
int *p = field[2]; // *p = 34
p--; // *p = 85
p--; // *p = 16
p--; // Fehler, Unterschreiten der Arraygrenze!!!
```

Auch hier muss man die Arraygrenzen beachten!

Wenn man Zeiger per Addition und Subtraktion auf andere Variablen zeigen lässt, ist die Rede von Zeigerarithmetik. Diese ist nur mit der Addition und der Subtraktion möglich. Zeiger dürfen auf keinen Fall multipliziert oder dividiert werden.

1.13.6 Zeiger auf Strukturen

Mit Zeigern kann nicht nur auf einfache Variablen und Arrays verwiesen werden, auch der Zugriff auf Strukturen ist möglich. Dies kann ein Zeiger auf ein einzelnes Strukturelement, oder gleich auf die ganze Struktur sein.

Für alle Beispiele in diesem Abschnitt wird folgende Struktur verwendet:

```
struct processor
{
    int taktfrequenz;
    float preis;
};
```

Das Zeigen auf ein einzelnes Strukturelement erfolgt gleich wie auf eine einzelne Variable, das Element muss aber über den Strukturnamen aufgelöst werden:

```
struct processor pentium4;
int *p;
```

```
pentium4.taktfrequenz = 2400;  
p = &pentium4.taktfrequenz; // *p = 2400
```

Dabei muss der Zeiger wie immer unbedingt mit dem gleichen Datentyp deklariert worden sein, wie das Strukturelement, auf das er zeigen soll. Es spielt dabei keine Rolle, wie die Struktur genau aussieht – wichtig ist nur der Datentyp des Elements!

Mit Zeigern kann man wie oben erwähnt auch auf ganze Strukturen verweisen. Hier muss der Zeiger nun mit dem gleichen Datentyp deklariert werden, wie die Struktur selber:

```
struct processor pentium4, *pt;  
pentium4.taktfrequenz = 2400;  
pentium4.preis = 179.0;  
pt = &pentium4;
```

Bisher ist alles gleich abgelaufen, wie beim Zeigen auf normale Variablen eines primitiven Datentyps. Der Zugriff auf die einzelnen Elemente der Struktur bietet den ersten Unterschied. Da der Punktoperator eine höhere Priorität als der Dereferenzierungsoperator hat, muss man den Zeiger mit runden Klammern umgeben:

```
pt.preis = 159; // Fehler  
(*pt).preis = 159;
```

Da diese Variante etwas umständlich ist, wurde der Pfeiloperator (->) geschaffen. Dieser wird wie folgt angewendet:

```
pt->preis = 159; // äquivalent zu (*pt).preis = 159;
```

1.14 Funktionen

Ein C-Programm besteht meistens aus mehreren, aber immer aus mindestens einer Funktion – der Funktion main(). Diese bildet den Einstiegspunkt in ein jedes C-Programm, siehe dazu die Funktion main...

Eine Funktion sollte immer nur eine Aufgabe erfüllen. Unteraufgaben können als weitere Funktionen implementiert werden, diese können wieder weitere Funktionen aufrufen. Eine Funktion kann sich auch selbst aufrufen, dies wird dann als Rekursion bezeichnet.

Funktionen bestehen aus zwei Teilen – dem Funktionskopf und dem Funktionsrumpf, diese werden in den folgenden Abschnitten beschrieben.

1.14.1 Der Funktionskopf

In den meisten Fällen übergibt man einer Funktion eine Reihe von Werten, mit denen die Funktion ihre Arbeit erledigen kann. Diese Werte werden als Parameter oder Argumente bezeichnet. Das Ergebnis aus der Arbeit der Funktion kann auch als Wert zurückgegeben werden, dabei spricht man vom Rückgabewert einer Funktion. Ausserdem hat eine Funktion auch immer einen Namen, durch den sie eindeutig identifiziert wird.

Obige Punkte werden alle im Funktionskopf definiert, welcher wie folgt aufgebaut ist:

```
[Datentyp der Rückgabe] [Funktionsname]([Parameterliste])
```

Der Datentyp der Rückgabe kann ein primitiver oder ein neu definierter Datentyp sein. Es können sogar Zeiger auf Variablen und Strukturen zurückgegeben werden. Die Rückgabe von void ist ebenfalls zulässig.

Für die Definition des Funktionsnamen muss man sich an die allgemeinen Richtlinien für die Benennung von Bezeichnern halten...

Die Parameterliste ist folgendermassen aufgebaut:

```
[Datentyp 1] [Variablenname 1], ..., [Datentyp n], [Variablenname n]
```

Es wird beschrieben, in welcher Reihenfolge welche Parameter übergeben werden sollen. Nun einige Beispiele für Funktionsköpfe:

```
int addieren(int *summand1, int *summand2)
*double dividieren(double dividend, double divisor)
struct processor createProcessor(float preis, int takt)
```

Die Funktion `addieren` erwartet zwei Zeiger auf `int` und gibt eine Variable vom Typ `int` zurück. `Dividieren` erwartet zwar keine Zeiger, sondern normale Variablen vom Typ `double`, liefert jedoch ein Zeiger auf eine Variable vom Typ `double` zurück. Die Funktion `createProcessor` erwartet zwei Variablen, eine vom Datentyp `float` und eine vom Datentyp `int`. Es wird eine Strukturvariable vom Typ `processor` zurückgegeben.

Ein Funktionskopf kann nur beschreiben, wie die Funktion aufgerufen werden muss und was diese allenfalls zurück gibt, die eigentliche Arbeit wird aber vom Funktionsrumpf übernommen.

1.14.2 Der Funktionsrumpf

Zu jedem Funktionskopf gehört auch ein Funktionsrumpf. Dieser bildet die eigentliche Logik einer Funktion. Der Funktionsrumpf ist nichts anderes als eine Blockanweisung, oftmals mit einer Rückgabe eines Werts kombiniert. Der zugehörige Funktionskopf muss unmittelbar vor dem Funktionsrumpf aufgeführt werden.

```
[Funktionskopf]
{
    [Anweisung 1];
    [Anweisung 2];
    ...
    [Anweisung n];
}
```

Innerhalb der Funktion darf jeder Befehl aufgerufen werden, der in `main` aufgerufen wird – `main` ist selber auch nur eine Funktion. Dazu gehören auch Variablendeklarationen. Alle Parameter, die im Funktionskopf erwartet werden, können als normale Variablen innerhalb des Rumpfes benutzt werden.

1.14.3 Der Rückgabewert

Eine Funktion dient dazu, bestimmte Anweisungen auszuführen und meistens auch um ein Ergebnis zurückzuliefern. Dieses Ergebnis kann das Resultat aus mathematischen Berechnung, ein Fehlercode oder sonst was sein.

Der Rückgabewert ist eine normale Variable, diese wird im Funktionsrumpf deklariert, es kann auch eine Variable aus der Parameterliste zurückgegeben werden. Der Typ des Rückgabewerts wird am Anfang vom Funktionskopf definiert. Die Rückgabe muss zwingend diesen Typ haben. Statt einer Variable können auch symbolische oder literale Konstanten zurückgegeben werden.

Die Rückgabe erfolgt mit dem Schlüsselwort `return`:

```
return [Ausdruck];
```

Falls die Funktion mit dem Rückgabety `void` definiert wurde, so darf kein Ausdruck zurückgegeben werden. In diesem Falle müsste `return` ohne einen Ausdruck da stehen und würde somit nur das Verlassen der Funktion bewirken.

Sobald `return` ausgeführt wurde, wird die Funktion verlassen. Jeder Befehl, der nach `return` steht wird niemals ausgeführt und ist somit nutzlos. Mehrere `return`-Anweisungen machen nur Sinn, wenn diese abhängig bestimmter Bedingungen ausgeführt werden.

```
if(x < 10)
    return x;
else
    return x - 10;
```

Wie man hier sieht, kann mit `return` ein beliebiger Ausdruck zurückgegeben werden, egal ob dieser berechnet, eine Konstante oder auch eine Variable ist.

1.14.4 Der Prototyp

In einem C-Programm hat man zwei Möglichkeiten, damit die Funktion main die anderen Funktionen findet.

1. Man definiert die Funktionen vor der Funktion main
2. Man verwendet Prototypen

Von der ersten Variante ist aber dringend abzuraten, da die Funktion main so am Ende der Quellcodedatei gesucht werden müsste. Schöner ist es, wenn man die Funktionen zwar vor main deklariert, jedoch erst danach definiert.

Neben der Funktionsdefinition, die sich aus dem Funktionskopf und dem Funktionsrumpf zusammensetzt, gibt es also noch eine Funktionsdeklaration, auch Prototyp genannt. Dieser Prototyp setzt sich aus dem Funktionskopf der zu deklarierenden Funktion und einem Semikolon zusammen.

Von folgender Funktion ausgegangen:

```
int sum(int a, int b)
{
    return a + b;
}
```

würde der Funktionsprototyp wie folgt aussehen:

```
int sum(int a, int b);
```

Der Prototyp muss nur darüber eine Aussage machen, welchen Typ der Rückgabewert hat, wie die Funktion heisst und wie viele Argumente in welcher Reihenfolge mit welchen Datentypen übergeben werden müssen. Die Angabe der Parameternamen (hier a und b) ist fakultativ und hat lediglich den Wert eines Kommentars. Der Prototyp kann auch folgendermassen aussehen:

```
int sum(int, int);
```

Ob man die Variablennamen aufführt oder nicht, ist dabei eine reine Stilfrage. Es sind beide Möglichkeiten absolut zulässig, denn für den Compiler selber stellen sie absolut keinen Unterschied dar!

Funktionsprototypen müssen vor der Hauptfunktion main stehen, die dazugehörigen Funktion sollten nach main folgen.

1.14.5 Funktionsaufruf

Damit eine Funktion einen praktischen Nutzen hat, muss diese aufgerufen werden. Dieser Aufruf ist nichts weiteres als ein normaler Ausdruck, der in einer beliebigen Variable (des entsprechenden Typs!) gespeichert, oder mit anderen Ausdruck kombiniert werden kann.

```
int summe = sum(164, 812);
sum(942, 321);
summe += sum(13, 46) + 24;
summe += 5 + sum(2, 5) * 15;
```

Der Wert des Ausdrucks eines Funktionsaufrufs ist dabei immer der Rückgabewert, der von der Funktion zurückgeliefert wird.

1.14.6 Wert- und Referenzübergabe

Ein Funktionsparameter kann auf zwei mögliche Arten übergeben werden. Einerseits als Wert, andererseits als Referenz. Bei der Wertübergabe erwartet die Funktion eine ganz normale Variable. Erwartet die Funktion jedoch einen Parameter in Form eines Zeigers, so spricht man von Referenzübergabe.

Ausgegangen vom folgenden Prototyp

```
int addieren(int *a, int b);
```

muss man der Funktion zwei Parameter übergeben. Der erste Parameter ist ein Zeiger, der zweite eine normale Variable.

Das interessante an einer Referenzübergabe ist nicht nur die Speicherersparnis, sondern viel mehr, dass man mit der gleichen Variable arbeiten kann, wie die Quellfunktion. Auf diese Weise können Rückgabewerte umgangen werden.

Betrachten wir folgende Funktion:

```
int addieren(int a, int b)
{
    return a + b;
}
```

Aufruf:

```
int summe = addieren(18, 25);
```

Statt das Ergebnis zurückzugeben, kann dieses auch in einem Referenzparameter gespeichert werden:

```
void addieren(int a, int b, int *summe)
{
    *summe = a + b;
}
```

Aufruf:

```
int summe;
addieren(18, 25, &summe); // summe wird als Referenz übergeben
```

Nach beiden Funktionsaufrufen hat die Variable summe exakt den gleichen Wert.

Mit der zweiten Variante kann man zwar die Geschwindigkeit erhöhen und den Speicherverbrauch minimieren, jedoch kann sie auch zu Fehlern führen. Einige Aufgaben können nur mit einer Referenzübergabe gelöst werden. Bei allen anderen sollte man aber auf die Wertübergabe zurückgreifen, ausser wenn extrem sparsam programmiert werden soll.

1.15 Makros

Funktionen sind ein gutes Mittel um Aufgaben besser zu gruppieren. Bei einem Funktionsaufruf geht jedoch jeweils viel Zeit verloren, da innerhalb des Speichers grosse Sprünge vorgenommen werden müssen. Kleinere Aufgaben sollten darum nicht in Funktionen ausgelagert werden. Soll aber die gleiche kleine Aufgabe mehrmals ausgeführt werden, ist eine Auslagerung dennoch sinnvoll.

Für solche Fälle bieten sich Makros. Ein Makro ist eine mini-Funktion, mit welcher kleinere Aufgaben elegant und schnell gelöst werden können.

```
#define PI 3.14 // normale Konstante
#define POTENZ(faktor) ((faktor) * (faktor)) // ein Makro
#define SUMME(a, b, c) ((a) + (b) + (c)) // noch ein Makro
```

Nach dem Makronamen muss eine Art Parameterliste folgen. Diese darf aber keine Datentypen, sondern nur Variablennamen enthalten. Nach einem Abstand folgt nun die Implementierung. Diese Implementierung muss auf einer einzigen Zeile stattfinden, da ein Präprozessorbefehl mit dem Zeilenende abgeschlossen ist.

Die Klammersetzung ist zwingend notwendig, ein Makro könnte auch innerhalb eines Ausdrucks verwendet werden, fehlende Klammern könnten dabei zu fehlerhaften Ergebnissen führen.

```
int result = POTENZ(2); // result = 2 * 2 = 4
```

Makros sind zwar ein gutes Mittel um minimale Aufgaben zu erfüllen, man sollte aber deren Möglichkeiten nicht voll ausschöpfen, da grosse Makrodefinitionen sehr unübersichtlich und somit fehleranfällig sind.

1.16 Die C-Standardbibliothek

C bietet eine umfassende Funktionsbibliothek, welche mit jedem Compiler mitgeliefert werden sollte. In folgenden Abschnitten werden jeweils diejenigen Funktionen erklärt, welche auch im Unterricht verwendet wurden und somit Prüfungsrelevant sein könnten.

1.16.1 Ein- und Ausgaben

Sämtliche Funktionen für Ein- und Ausgaben befinden sich in der Bibliothek `stdio.h`

printf()

```
int printf(const char *, arg1, arg2, ..., argN);
```

Es wird der angegebene Formatstring ausgegeben. Die Platzhalter werden jeweils durch die Argumente 1 bis n ersetzt. Die Funktion liefert die Anzahl ausgegebener Zeichen zurück. Erklärungen zum Formatstring gibt es im Anhang A.

Beispielaufwurf:

```
printf("X = %i, Y = %f, Z = %c\n", 16, 52.35, 'z');
```

Ausgabe:

```
X = 16, Y = 52.3500, Z = z
```

scanf()

```
int scanf(const char *, adresse1, adresse2, ..., adresseN);
```

Es werden Variablen anhand des Formatstrings von der Tastatur mit Werten belegt. Der Formatstring entspricht demjenigen der Funktion `printf()`.

Beispielaufwurf:

```
scanf("%i %f %c", &a, &b, &c);
```

Es werden, getrennt durch einen Leerschlag, drei Variablen vom Typ `int`, `float` und `char` eingelesen. Den Variablen muss ein `&` vorangestellt werden, damit die Werte an der Adresse der jeweiligen Variablen gespeichert werden.

fflush()

```
fflush(FILE *);
```

Der Eingabepuffer wird geleert und in die angegebene Datei geschrieben. Die Funktion sollte nach jedem `scanf()` aufgerufen werden, damit der Buffer anschliessend wieder leer ist. Oftmals wird als Argument `stdin` angegeben.

1.16.2 Mathematische Funktionen

Folgende Funktionen sind in der Bibliothek `stdlib.h` definiert.

| Prototyp | Rückgabewert |
|-----------------------------|--|
| <code>int rand(void)</code> | Zufallszahl zwischen dem Wert der Konstanten <code>RAND_MAX</code> und 0 |
| <code>int abs(int)</code> | Absolutwert der übergebenen Zahl |

Weitere Funktionen befinden sich in der Bibliothek `math.h`

| Prototyp | Rückgabewert |
|-----------------------------|--|
| <code>int rand(void)</code> | Zufallszahl zwischen dem Wert der Konstanten <code>RAND_MAX</code> und 0 |
| <code>int abs(int)</code> | Absolutwert der übergebenen Zahl |

| <i>Prototyp</i> | <i>Rückgabewert</i> |
|---|--|
| <code>double sin(double)</code> | Sinus der übergebenen Zahl |
| <code>double cos(double)</code> | Cosinus der übergebenen Zahl |
| <code>double tan(double)</code> | Tangens der übergebenen Zahl |
| <code>double asin(double)</code> | Arcussinus der übergebenen Zahl |
| <code>double acos(double)</code> | Arcuscosinus der übergebenen Zahl |
| <code>double pow(double, double)</code> | Erstes Argument hoch zweites Argument |
| <code>double sqrt(double)</code> | Erste Wurzel aus übergebener Zahl |
| <code>double ceil(double)</code> <code>double floor(double)</code> | Nächste Ganzzahl des übergebenen Wertes |
| <code>double fabs(double)</code> | Absolutwert der übergebenen Gleitkommazahl |

1.16.3 String-Verarbeitung

Die Bibliothek `string.h` kennt eine Menge von Funktionen für die Bearbeitung von `char`-Arrays. Das Ergebnis der Funktion findet sich dann jeweils in deren Rückgabe wieder.

| <i>Prototyp</i> | <i>Rückgabewert</i> |
|---|---|
| <code>char *strcat(char *, char *)</code> | Hängt die zweite Zeichenkette an die erste an |
| <code>int strcmp(char *, char *)</code> | Vergleicht die beiden Strings und gibt 0 zurück, falls die Strings gleich sind. |
| <code>char *strcpy(char *, char *)</code> | Kopiert den zweiten String in den ersten. Der erste muss dabei genügend Platz für den zweiten bieten. |
| <code>size_t strlen(char *)</code> | Gibt die Länge des Strings zurück |

1.16.4 Speichermanipulation

malloc()

```
void *malloc(size_t);
```

Die Funktion `malloc` reserviert die übergebene Anzahl Bytes an Speicher und gibt eine Referenz vom Typ `void` darauf zurück. Die Anzahl wird oftmals mit dem `sizeof`-Operator ermittelt, die Rückgabe muss oft durch einen Cast in die entsprechende Form gebracht werden.

Beispiel:

```
char *chr = (char *)malloc(sizeof(char) * 5);
```

In obigem Beispiel wird ein Zeiger auf eine Variable des Datentyps `char` erstellt. Diesem werden 5 Speicherzellen mit jeweils der Grösse von `char` reserviert.

1.17 Anhang A – der Formatstring

Ein Formatstring setzt sich aus normalen Text und Platzhaltern zusammen. Platzhalter werden immer mit einem % eingeleitet, darauf folgt der Formatkennzeichner. Letzterer ist meistens abhängig vom Datentyp der Variable, die ausgegeben werden soll.

Hier die wichtigsten Formatkennzeichner:

| Formatkennzeichner | Argument-Typ | Zahlensystem |
|---------------------------|---------------------|-------------------------------------|
| d, i | int | dezimal |
| u | unsigned | dezimal |
| f | double / float | Gleitkommazahl, dezimal |
| c | char, int | Einzelnes Zeichen |
| s | String | Ausgabe des char-Arrays bis zu '\0' |

Es gibt zahlreiche Möglichkeiten, die Genauigkeit der Ausgabe von Fließkommazahlen zu beeinflussen, dies geht allerdings zu stark ins Detail und ist darum nicht prüfungsrelevant.

2 C#-Grundlagen

In diesem Teil werden die Grundlagen der Programmiersprache C# behandelt. Weiter wird auf die Einzelheiten eingegangen, die für die Prüfung laut VFI auch wirklich relevant sind. Viele Themen werden nur sehr kurz behandelt, auch hier besteht kein Anspruch auf Vollständigkeit.

2.1 Syntax

Die C#-Syntax ist sehr stark derer von C nachempfunden. Bei der OOP kann man auch starke Einflüsse aus der Programmiersprache C++ wie auch aus Visual Basic erkennen.

Die Grundlegenden Sprachelemente, wie Iterationen, Ausdrücke und Anweisungen sind gleich wie in C, darum werden diese hier nicht mehr weiter erläutert. Auf die jeweiligen Unterschiede wird in den dazugehörigen Abschnitten jeweils näher eingegangen.

Kommentare funktionieren noch immer gleich wie in C.

Die Syntaxteile, die jedoch in C noch nicht vorhanden sind (z.B. OOP) werden jedoch explizit in eigenen Abschnitten erklärt.

2.2 OOP

C# ist vollkommen objektorientiert aufgebaut. Da die *objektorientierte Programmierung*⁴ jedoch nicht der Schwerpunkt des Moduls 118 darstellt, wird diese nicht in allen Details beschrieben. Es wird nur auf die Teile eingegangen, die für die Verwendung von C# unbedingt notwendig sind.

2.2.1 Der strukturierte Ansatz

Strukturierte Programmiersprachen wie C sehen eine strikte Trennung von Daten und Anweisungen vor. Daten werden durch Variablen und Strukturen repräsentiert, Anweisungen können zu Funktionen zusammengefasst werden. Man hat also zwei Teile, die jedoch von einander abhängig sind, jedoch nicht als eine Einheit implementiert werden können.

2.2.2 Der objektorientierte Ansatz

Aus obiger Problematik ist die OOP entstanden. Daten und Anweisungen können nun in sog. Klassen zusammengehalten werden und bilden so nicht nur eine logische, sondern auch eine physische Einheit. Der Zugriff auf die Daten kann dabei eingeschränkt bzw. geschützt werden. Dies bezeichnet man als Datenkapselung. Hier bei gilt der folgende, wichtige Merksatz:

In einer objektorientierten Anwendung sollten alle Eigenschaften vor Zugriffen von aussen geschützt (private) sein und nur über die öffentlichen Methoden (Schnittstelle⁵) verändert werden können!

Man spricht dann von Eigenschaften oder Membervariablen (= Variablen die zu einem Objekt gehören) und Methoden oder Memberfunktionen (= Funktionen die zu einem Objekt gehören).

2.2.3 Klassen

Das wichtigste Konstrukt in der OOP ist die Klasse. In einer Klasse wird angegeben, welche Daten diese beinhaltet und wie diese verarbeitet werden können. Eine Klasse ist oftmals eine Abstraktion eines Sachverhalts aus der Umwelt. Mit Klassen kann man z.B. die Funktionalität eines Geräts oder einer Schnittstelle definieren, den Zugriff auf eine Datei besser verwalten oder auch Dinge aus der realen Welt wie z.B. Katzen und Autos besser abbilden.

Eine Klasse wird nicht direkt vom Programmierer verwendet, sondern ist nur ein Bauplan für Objekte, welche im folgenden Abschnitt erklärt werden.

⁴ Im folgenden Dokument als „OOP“ bezeichnet

⁵ Die Summe aller öffentlichen Elemente einer Klasse nennt man Schnittstelle.

In C# muss jede Anweisung innerhalb einer Klasse aufgeführt sein, man kann also nicht ohne Klassen programmieren!

2.2.4 Objekte

Damit ein Programmierer eine Klasse auch wirklich verwenden kann, muss aus dieser zuerst ein Objekt erstellt werden. Das erstellen eines Objekts aus einer Klasse wird dabei als Instanzierung bezeichnet – ein Objekt ist eine Instanz einer Klasse. Aus einer Klasse können beliebig viele Objekte erstellt werden, ein Objekt kann jedoch nur aus einer Klasse bestehen.

2.2.5 Objektstatus

Die Gesamtheit der Werte aller Attribute eines Objektes bezeichnen wird als dessen Status.

2.2.6 Eigenschaften

Die Daten eines Objekts werden durch Eigenschaften repräsentiert. Eine Eigenschaft ist dabei nichts anderes, als eine einfache Variable, welche von der ganzen Klasse verwendet werden kann und somit nicht auf eine einzige Funktion beschränkt ist.

Eine Eigenschaft kann gegen aussen geschützt werden, es ist jedoch auch möglich, dass verschiedene Klassen auf eine Eigenschaft einer Klasse zugreifen können.

2.2.7 Methoden

Um die Logik (also die Anweisungen) einer Klasse zu definieren, werden Methoden verwendet. Eine Methode ist nicht anderes, als eine Funktion, die innerhalb einer Klasse steht. Da sich aber in C# alles in Klassen abspielt, ist jede Funktion sogleich eine Methode.

Methoden können auch in ihrer Sichtbarkeit eingeschränkt werden, sodass diese nur von der Klasse aus sichtbar ist, in welcher sie auch tatsächlich steht.

2.2.8 Signatur

Die Signatur einer Methode besteht aus dem Methodennamen und der Liste der Parameter mit ihren Datentypen.

2.3 Programmaufbau

Da sich in C# alles in Klassen abspielt, ist der Programmaufbau dementsprechend etwas anders als bei C. In den folgenden Abschnitten werden diese Unterschiede erklärt.

2.3.1 Die Main-Methode

Jedes C#-Programm startet mit der Methode Main. Im Gegensatz zu C wird Main hier gross geschrieben. Die Methode sollte folgendermassen aussehen:

```
static void Main()
{
    ... // weiterer Programmcode
}
```

2.3.2 wohin mit Main?

Die Methode Main stellt einen Sonderfall dar, da sie nicht logisch einer bestimmten Klasse zugeordnet werden kann. In welcher Klasse sie steht ist letztendlich egal. In der Praxis steht Main immer da wo das ganze Programm „gemanaged“ wird.

Main gehört also einfach in eine beliebige Klasse:

```
namespace [Namensraum]
{
    public class [Klassenname]
    {
        static void Main()
        {
            ... // weiterer Programmcode
        }
    }
}
```

Ein Klassenname ist ein normaler Bezeichner und untersteht darum auch den entsprechenden Benennungsregeln. Grossschreibung ist aber aus Stilgründen zu empfehlen.

Es ist zu empfehlen, jede Klasse in einen Namespace zu packen. Dabei geht es um die Zugriffsrechte, weitere Details sind nicht prüfungsrelevant. Auch namespaces sind Bezeichner und folgend darum den gleichen Regeln wie Klassen.

2.4 Erweiterte Datentypen verwenden

In den folgenden Abschnitten werden vor allem Enumeratoren, Strukturen und Datenfelder behandelt.

2.4.1 Datentypen

| Typ | Byte | Bitgröße | Beschreibung |
|---------|-----------------------|---|--|
| byte | 1 | 0 bis 255 | Byte-Wert mit Vorzeichen |
| sbyte | 1 | -128 bis 127 | Byte-Wert ohne Vorzeichen |
| short | 2 | -32.768 bis 32.767 | Short-Wert mit Vorzeichen |
| ushort | 2 | 0 bis 65.535 | Short-Wert ohne Vorzeichen |
| int | 4 | -2.147.483.648 bis 2.147.483.647 | Integer-Wert mit Vorzeichen |
| uint | 4 | 0 bis 4.294.967.295 | Integer-Wert ohne Vorzeichen |
| long | 8 | -9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807 | Langer integer-Wert mit Vorzeichen |
| ulong | 8 | 0 bis 18.446.744.073.709.551.615 | Langer integer-Wert ohne Vorzeichen |
| float | 4 | Gültigkeitsbereich: $3.4 \cdot 10^{-45}$ bis $3.4 \cdot 10^{38}$ Genauigkeit: 7 stellig | Gleitkommazahl |
| double | 8 | Gültigkeitsbereich: $5 \cdot 10^{-324}$ bis $1.7 \cdot 10^{308}$ Genauigkeit: 15 bis 16 stellig | Gleitkommazahl mit doppelter Genauigkeit |
| decimal | 16 | Ohne Dezimalpunkt: 0 bis $79 \cdot 10^{27}$ Mit Dezimalpunkt: $1 \cdot 10^{-29}$ bis $7.9 \cdot 10^{27}$ | Zahl mit fester Genauigkeit |
| string | 10 plus 2 pro Zeichen | Max. Länge von ca. 2'000'000'000 Zeichen | Unicode-Zeichenfolge |
| char | 2 | Beliebig | Unicode-Zeichen |

| Typ | Byte | Bitgröße | Beschreibung |
|--------|------|---------------------------|-----------------|
| bool | 2 | Wahrheitswert(true,false) | Boolescher Wert |
| object | 4 | Universaler Datentyp | Object |

2.4.2 Schlüsselwörter

Schlüsselwörter sind vordefinierte reservierte Bezeichner und dürfen somit nicht für eigene Variablen, Klassen uws. benutzt werden. Falls doch ein Schlüsselwort für etwas verwenden will muss ein @ voransetzen (z.B. @checked). Hier eine Tabelle mit allen Schlüsselwörter:

| | | | | | |
|----------|------------|---------|----------|-----------|----------|
| abstract | as | base | bool | break | byte |
| case | catch | char | checked | class | const |
| continue | decimal | default | delegate | do | event |
| explicit | extern | false | finally | fixed | float |
| for | foreach | goto | if | implicit | in |
| int | interface | new | null | object | operator |
| out | override | params | private | protected | public |
| readonly | ref | return | sbyte | sealed | short |
| sizeof | stackalloc | static | string | struct | switch |
| this | throw | true | try | typeof | uint |
| ulong | unchecked | unsafe | ushort | using | virtual |
| volatile | void | while | | | |

2.4.3 Konstante

Eine Konstante wird wie folgt deklariert:

```
const int jahr = 2005;
```

Der Wert der Variable `jahr` kann so nicht mehr verändert werden.

2.4.4 Typsuffixe

Wenn die Datentypen `long`, `float` oder `decimal` genutzt werden, müssen die sog. Typsuffixe (L, F, M) verwendet werden. Ansonsten werden diese als Literale vom Typ `int` oder `double` behandelt. Beispiele:

```
float max = 99.99;      //Fehler!
float max = 99.99F;     //richtig
decimal geld = 300.50;  //Fehler!
decimal geld = 300.50M; //richtig
```

2.4.5 Casts

Man kann den aus C bekannten Cast-Operator auch in C# verwenden.

Eine Übersicht welche Datentypen zu welchen passen findet ihr im Buch „Grundlagen und Profiwissen - Visual C#.net“, Im Kapitel 3.4.2 auf der Seite 126/127.

2.5 Enumeratoren

Enumeratoren funktionieren im Grossen und Ganzen gleich wie in C, nur dass in C# noch zusätzlich ein Datentyp für die Werte definiert werden kann.

Die Deklaration des Enumerator-Datentyps kann innerhalb oder ausserhalb einer Klasse erfolgen, nicht aber in einer Methode.

Syntax:

```
enum Bezeichner : Datentyp
```

Beispiel:

```
enum Monate : byte
{
    Januar = 1, Februar, März, April, Mai, Juni, Juli, August, September,
    Oktober, November, Dezember
}
```

2.5.1 Zugriff

Wenn ein Enumtyp deklariert ist, können Sie ihn wie jeden anderen Datentyp verwenden.

Beispiel:

```
Monate ersterMonat;
ersterMonat = Monat.Januar;
```

Folgende Beispiele sind kompilierbar:

```
Monate m1, m2;
m1 = Monate.Januar;    // Ergebnis: m1 = Januar
m1 = (Monate)77;       // Ergebnis: m1 = 77
m1 = (Monate)4;        // Ergebnis: m1 = April

int temp = (int)m1;    // Ergebnis: temp = 4
m2 = m1 + 1;          // Ergebnis: m2 = Mai
m2 = m1 + 20;         // Ergebnis: m2 = 24
```

Diese Beispiele funktionieren allerdings nicht:

```
Monate m1;
m1 = April;                // Compilerfehler
m1 = 4;                    // Compilerfehler

byte btemp;
m1 = Monate.Januar;
btemp = m1;                // Compilerfehler
btemp = MaschinenZustand.November; // Compilerfehler
int temp = m1;             // Compilerfehler
```

2.6 Datenfelder (Arrays)

Datenfelder entsprechen einer bestimmten Anzahl Variablen des selben Datentyps, die unmittelbar hintereinander im Arbeitsspeicher liegen. Mittels Index kann auf die einzelnen Elemente zugegriffen werden.

2.6.1 Deklaration

Die Deklaration eines Arrays ist dieselbe wie bei einer normalen Variable, lediglich nach dem Datentypbezeichner folgt ein eckiges Klammerpärchen [].

Die Grösse des Arrays wird bei der Deklaration noch nicht angegeben, denn es existiert noch keine Instanz dieses Objekts, lediglich eine Referenz auf null wird angelegt.

Syntax:

```
Datentyp[] arrVariable;
```

Beispiel:

```
int[] a;
```

2.6.2 Instanziierung

Die Array-Variable zeigt nun noch ins Leere und muss zuerst noch instanziiert werden.

Syntax:

```
arrVariable = new Datentyp[Dimension(en)];
```

Beispiel:

```
a = new int[4];
```

Die Deklaration und die Instanziierung können auch in einer Anweisung erledigt werden

Syntax:

```
Datentyp[] arrVariable = new Datentyp[Dimension(en)];
```

Beispiel:

```
int[] a = new int[4];
```

2.6.3 Initialisierung

Arrays werden Erzeugen mit Standardwerten initialisiert (einfache Datentypen mit 0, boolesche Elemente mit false, Verweistypen mit null). Um andere Werte zuzuweisen gibt es zwei Varianten.

Einmal mit der new – Anweisung

Syntax:

```
Datentyp[] arrVariable;  
arrVariable = new Datentyp[Dimension(en)] {wert1, wert2, ... };
```

Beispiel:

```
int[] a; //Deklarieren  
a = new int[4] {10,9,8,7}; //Erzeugen und Initialisieren
```

oder ohne new – Anweisung.

Syntax:

```
Datentyp[] arrVariable = {wert1, wert2, ... };
```

Beispiel:

```
int[] a = {10,9,8,7}; //Deklarieren, Erzeugen und Initialisieren
```

2.6.4 Zugriff mit for-Schleife

Ein kurzes Beispiel wie man ein int-Array mit der Hilfe einer for-Schleife mit den Zahlen 0-99 füllt.

Beispiel:


```
int[] a = new int[100];
for (int i = 0; i != a.Length; i++)
    a[i] = i;
```

2.6.5 Zugriff mit foreach-Schleife

Ein kurzes Beispiel wie man ein `int`-Array mit der Hilfe einer `foreach`-Schleife mit den Zahlen 0-99 ausgibt.

Beispiel:

```
foreach (int feld in a)
    MessageBox.Show(feld.ToString());
```

Zu beachten ist: `feld` ist nur eine lokale Kopie. Schreibvorgänge auf `feld` würden auf das Array keine Auswirkung haben.

2.6.6 Mehrdimensionale Arrays

Ein Array darf auch mehr als nur eine Dimension haben. Die einzelnen Dimensionen werden beim Deklarieren bzw. Zugriff mit einem Komma getrennt.

Syntax:

```
Datentyp[,] arrVariable = new Datentyp[Dimension1, Dimension2];
```

Beispiel:

```
float[,] c = new float[10, 20]; //c(0,0) ... c(9,19)
```

Im folgendem Programmabschnitt wird ein zweidimensionales Array durwegs mit dem wert 5.5 belegt. (Für die Bestimmung der Länge ist die Eigenschaft `Length` ungeeignet, es sollte stattdessen die `GetLength`- bzw. `GetUpperBound`- Methode verwendet werden):

```
for (int i = 0; i < c.GetLength(0); i++) // für erste Dimension
{
    for (int j = 0; j < c.GetLength(1); j++) // für zweite Dimension
    {
        c[i,j] = 5.5F;
    }
}
```

2.6.7 Initialisierung

Auch Mehrdimensionale Arrays lassen sich mit Anfangswerten initialisieren, die Werte der einzelnen Dimensionen werden in geschweiften Klammerpaaren hintereinander aufgeführt.

Beispiel:

```
String[,] personen = {{"Meier", "Berlin"}, {"Schultze", "Leipzig"},
{"Krause", "Bonn"}};

string s = personen[2,1];
MessageBox.Show(s); // zeigt "Bonn"
```

2.6.8 Zuweisen von Arrays

Ebenso wie man den Inhalt einer Variablen einer anderen Variablen zuweisen kann, z.B. `a = b`, kann man auch einem Array eine anderes Array zuweisen. Da aber Arrays in allen .NET-Programmiersprachen Referenztypen sind, speichern Array-Variablen also direkt keinen Wert, sondern nur Referenzen (Zeiger) auf einen bestimmten Speicherbereich.

Das gegenseitige Zuweisen von Array-Variablen führt also nicht zu einem Kopieren der Werte, sondern lediglich zum Zuweisen einer neuen Speicheradresse.

Beispiel:

```
int[] a = {1, 2, 3};
int[] b = new int[a.Length]; // kein Kopieren, sondern nur Referenzieren!
b = a;
b[0] = 5;
MessageBox.Show(b[0].ToString()); // zeigt erwartungsgemäss "5" an
MessageBox.Show(a[0].ToString()); // zeigt nicht "1", sondern "5" an!
```

2.6.9 Leeren von Arrays

Um die Felder eines Arrays zu löschen (sie werden nicht entfernt!), verwenden man die Clear-Methode. Es handelt sich hierbei um eine statische Methode der Array-Klasse.

Syntax:

`Array.Clear(Array, Startindex, Anzahl zu löschende Felder);`

Beispiel:

```
int[] a = {5, 6, 7, 8, 9};
MessageBox.Show(a[3].ToString()); // viertes Element hat Wert 8
Array.Clear(a, 3, 1); // ab dem 4. Element von a wird ein Element gelöscht
MessageBox.Show(a[3].ToString()); // viertes Element hat Wert 0
```

Um eine komplettes Array zu entfernen, muss man der Array-Variablen den Wert null zuweisen (dereferenzieren).

Beispiel:

```
int[] a = new int[4]{5, 6, 7, 8};
a = null; // Dereferenzieren
```

2.6.10 Eigenschaften und Methoden

| Eigenschaft/Methode | Beschreibung |
|----------------------------|--|
| Length | liefert Gesamtanzahl der Array-Elemente |
| Rank | liefert die Anzahl der Array-Dimension |
| Clone() | kopiert alle Elemente aus Quell- in Ziel-Array |
| CopyTo(Array, Index) | kopiert Elemente ab Index von Quell- in Ziel-Array |
| GetLength | liefert die Anzahl Elemente einer bestimmten Dimension |
| Initialize() | initialisiert alle Array-Elemente auf Standardwert |

Beispiel

```
int[] a = {6, 8, 7, 9};
int[] b;
b = (int[])a.Clone();
MessageBox.Show(b[1].ToString());
```

Wichtige Klassenmethoden:

| Methode | Beschreibung |
|--|---|
| Clear(Array, Startindex, Anzahl zu löschende Felder) | löscht die Werte der Elemente ab index für length-Elemente (die Elemente bleiben erhalten!) |

| <i>Methode</i> | <i>Beschreibung</i> |
|------------------------------------|---|
| <code>IndexOf(Array, Value)</code> | dient zur (langsamen) sequenziellen Such in einem Array |
| <code>Sort(Array)</code> | dient zur vollständigen bzw. teilweisen Sortierung eines Arrays |

```
int[] a = {5, 8, 7, 6, 9};
Array.Sort(a); // Arrayinhalt = {5, 6, 7, 8, 9}
```

2.7 Datenfelder

In diesem Block werden statische, dynamische (einfache & strukturierte) Datenfelder und verkettete Listen behandelt.

2.7.1 Statische Datenfelder

Wenn man mehrere Variablen eines bestimmten Datentyps braucht (z.B. 100) kann man diese mit einem einzigen Befehl erstellen.

Beispiel mit C `:int iFeld[100];`

Beispiel mit C# `:int[] iFeld = new int[100];`

Die so erzeugten 1000 Variablen liegen immer unmittelbar hintereinander im Speicher.

2.7.2 Dynamische Datenfelder

Mit den statischen Datenfeldern haben wir aber leider ein Problem. Wir haben nun Speicher für 100 Variablen reserviert. Was geschieht nun, wenn wir aber plötzlich 150 Variablen speichern müssten?

Eine Möglichkeit wäre es, dass man den schlimmsten Fall annimmt und die Arrays immer mit der grösst möglichen Grösse erstellt. Aber auch dies ist keine gute Idee, da einem so schnell der Speicher ausgeht. Da die Elemente eines Datenfeldes bekanntlich unmittelbar hintereinander im Speicher liegen müssen. Wenn das Betriebssystem nicht einen derart grossen Platz mehr frei hat, beginnt es die Speicherseiten zu "swappen" (vorübergehend auf Festplatte auslagern) und wenn das auch nicht mehr geht, gibt es eine Memory-Allocation Fehlermeldung "Speicherplatz konnte nicht reserviert werden". Also muss nach einer anderen Variante gesucht werden.

Wäre es nicht toll, wenn wir nicht bereits zu Beginn die Grösse des Arrays festlegen müssten, sondern diese zur Laufzeit ändern könnten. So was gibt es und man nennt es **dynamische Datenfelder**. Dynamische Datenfelder sind also Datenfelder, die zur Laufzeit ihre Grösse ändern können.

Leider können wir nicht einfach einem Array eine neue Grösse geben, analog zu dem folgenden Beispiel:

```
int[] Fled = new int[5];
Feld = new int[10];
```

Mit dem Befehl `Feld = new int[10]` deklarieren wir für das Array `Feld` neu (mit dem Speicherplatz von 10).

2.7.3 Erster dynamischer Ansatz

Wie könnte ein Array vergrössert werden, ohne dass wir den Inhalt verlieren? Dies ist schon möglich allerdings benötigen wir hierzu ein Temporäres Array, in welches wir die Daten zwischenspeichern können.

Beispiel:

```
int[] Feld = {6, 8, 7, 9, 5};
int[] tempFeld = new int[Feld.Length + 1]; /* Array, das einen Wert mehr
                                             aufnehmen kann */
```

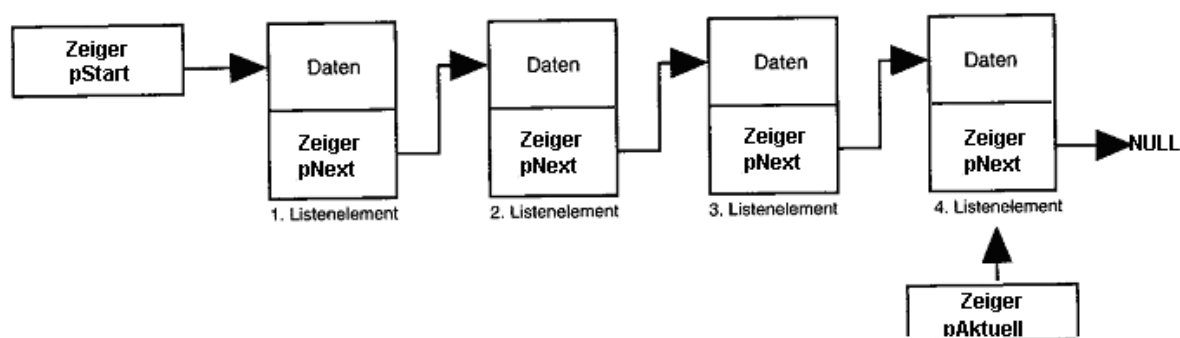
```
Feld.CopyTo(tempFeld,0); // Array wird kopiert
Feld = tempFeld; // Feld wird mit von temp Feld überschrieben
```

Somit haben wir zwar ein dynamisches Array, jedoch ist diese Lösung sehr Ressourcen fressend und deshalb auch nicht die richtige Lösung. Ein guter erster Ansatz ist es dennoch, da das Datenfeld nun dynamisch ist - theoretisch könnte so eine Applikation erstellt werden.

2.8 Verkettete Dynamische Datenfelder

Ein grosser Nachteil von Dynamischen Datenfeldern (ohne Verkettung) ist, dass die Daten bei Erweiterungen immer hin- und herkopiert werden müssen. Das ist überhaupt nicht optimal hinsichtlich der Ressourcen (Speicher und Verarbeitungsgeschwindigkeit). Eine viel bessere Lösung diesbezüglich bieten die verketteten Listen.

Eine verkettete Liste ist eine Folge von Variablen, welche aufeinander Verweisen. Das heisst, eine Variable hat nicht nur seine Daten (z.B. Pers-Nr, Namen, etc.) sondern auch noch einen Zeiger auf die nächste Variable:



Prinzipiell unterscheidet man folgende Formen von verketteten Listen:
Unterscheidung nach Struktur:

- einfach-verkettete Listen (haben nur einen Verweis zum Nachfolger)
- doppelt-verkettete Listen (haben einen Verweis zum Nachfolger UND zum Vorgänger)

Unterscheidung nach Füllverhalten:

- unsortierte-verkettete Listen (Die Daten werden in der eingegebenen Sequenz gespeichert)
- sortierte- verkettete Listen (Die Daten werden in sortierter Reihenfolge gespeichert)

Vorteile von verkettete Listen gegenüber den dynamischen Datenfelder sind:

- Sehr einfache Erweiterung der Liste (ohne Kopieren des Datenfeldes).
- Die Daten müssen nicht mehr am Stück hintereinander im Speicher liegen (Das Betriebssystem kann sich das aufwändige allozieren und verschieben von grossen Speicherbereichen sparen).

2.9 ArrayList

Mit C# haben wir die Möglichkeit verkettete Listen abstrahiert mit einer ArrayList zu lösen. Bei dieser Form der Datenspeicherung können einzelne Elemente flexibler hinzugefügt und entfernt werden als bei der Verwendung von Arrays. Für viele Aktionen wird im Unterschied zu Arrays kein Index benötigt.

Bei Aufzählungen und Auflistungen kann eine spezielle Schleife angewendet werden, die ohne Lauf- bzw. Zählvariable und ohne Index auskommt.

ArrayLists nehmen als Daten alle Instanzen der Klasse `object` auf, also alles was in C# einen Wert aufnehmen kann.

Syntax:

```
foreach (Datentyp Temp-Variable in Auflistung)
{
    ...
}
```

Beispiel:

```
ArrayList myList = new ArrayList();
int i = 5;
myList.Add(i);
i = 3;
myList.Add(i);
```

```
foreach(int tmp in myList)
{
    MessageBox.Show(tmp);
}
```

Die Verwendung dieser foreach-Schleife bietet sich an, wenn alle in einer Liste vorhandenen Elemente angesprochen werden sollen. Zur Adressierung ausgewählter Elemente eignet sie sich nicht. Auch diese Schleife kann mit `break` vorzeitig verlassen oder mit `continue` beim nächsten Schleifendurchlauf fortgesetzt werden.

2.9.1 Eigenschaften und Methoden

| Eigenschaft/Methode | Beschreibung |
|----------------------------|--|
| Count | Gibt die Anzahl der Listenelemente zurück |
| Capacity | Ruft die maximale Anzahl der Listenelemente ab oder legt sie fest |
| Sort() | Sortiert die Elemente |
| Add() | Fügt der Liste ein neues Element hinzu |
| Insert() | Fügt an einer bestimmten Indexposition ein neues Element ein |
| Clear() | Löscht die Liste |
| Remove() | Sucht und entfernt ein Element, das mit einem übergebenen Suchmuster übereinstimmt |
| RemoveAt() | Löscht ein Element an der übergebenen Indexposition |

3 Strukturierte Diagramme

Viele zu realisierende Applikationen sind umfangreich und kompliziert. Damit man nun vor dem eigentlichen Programmieren trotzdem den Überblick nicht verliert, ist es wichtig sich einen systematisch aufgebauten Plan zu erstellen, der den Lösungsweg beschreibt. Hier kommen die sogenannten strukturierten Diagramme wie z.B.

- Programmablaufpläne (Flussdiagramm)
- Struktogramme (Nassi-Shneidermann)


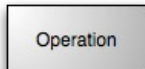
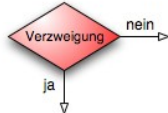
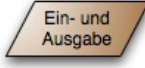
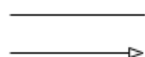
zum Einsatz. Sie beschreiben den Lösungsweg graphisch übersichtlich gegliedert. Strukturierte Diagramme haben u.a. den Vorteil, dass sie auch von einem Kunden, der sich mit Programmcode weniger auskennt, rasch verstanden werden und ihm einen Überblick verschaffen, wie seine Problemstellung nach und nach in die Tat umgesetzt wird.

Grob zusammengefasst sind strukturierte Diagramme Hilfsmittel bei der Programmentwicklung, bei denen die Programmstruktur sich bildhaft als Ablauf darstellen lässt.

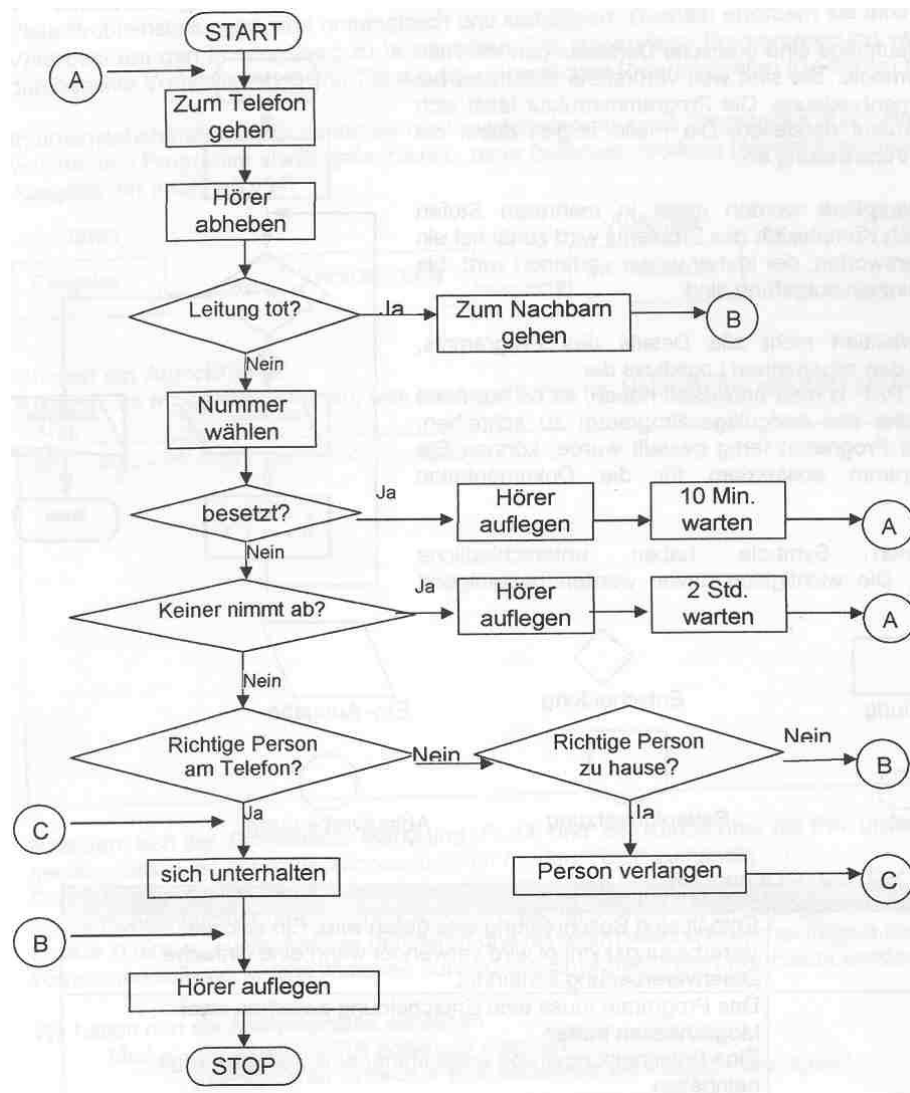
3.1 Das Flussdiagramm

Ein Flussdiagramm ist nicht so detailliert aufgebaut wie das eigentliche Programm, das aus ihm resultiert – er stellt lediglich den allgemeinen Logikfluss dar.

Aufgebaut ist das Flussdiagramm aus einzelnen, genormten (DIN 66001) Symbolen, welche unterschiedliche Bedeutungen haben. Sie werden in der nachfolgenden Tabelle genauer erläutert:

| Bezeichnung | Symbol | Beschreibung |
|----------------|---|--|
| Terminal |  | Wird verwendet für den Beginn, sowie das Ende des Programms. |
| Verarbeitung |  | Wird verwendet, wenn eine einfache Datenverarbeitung stattfindet. Z.B. bei Variablendeklarationen, Rechnungen, Zuweisungen, usw. |
| Entscheidung |  | Das Programm muss eine Entscheidung zwischen zwei Möglichkeiten treffen. |
| Ein- Ausgabe |  | Wird verwendet, wenn z.B. etwas am Bildschirm ausgegeben, bzw. vom Benutzer eingegeben werden muss. |
| Pfeile, Linien |  | Zur Verbindung zum nächstfolgenden Element. |


Beispiel:



3.2 Struktogramme

Ein Struktogramm ist die grafische Darstellung eines Programmablaufs in Form eines geschlossenen Blocks. Dieser kann, entsprechend den einzelnen, logischen Grundstrukturen in verschiedene untergeordnete Blöcke aufgeteilt werden.

- Der **Vorteil** eines Nassi-Shneidermann-Diagrammes gegenüber dem Flussdiagramm ist, dass die Darstellung dem zu programmierenden Algorithmus viel ähnlicher ist.
- Der **Nachteil** ist aber, dass diese Darstellung nur von Fachmännern gelesen werden kann. Für Leute mit geringen Informatikkenntnissen ist sie ungeeignet.

| Bezeichnung | Darstellung | Verwendung |
|----------------|---|-------------------------------------|
| Prozedurrahmen |  | Rahmen für alle anderen Komponenten |

| Bezeichnung | Darstellung | Verwendung | | | | | | | | | | | | |
|---------------------|---|-------------|-----------------|-------------|-------------------|---------------------------|---------|---------|----|-----|-----|-----|-----|--------------------------------|
| Anweisung | <table><tr><td>Anweisung 1</td></tr><tr><td>.....</td></tr><tr><td>Anweisung n</td></tr></table> | Anweisung 1 | | Anweisung n | Normale Anweisung | | | | | | | | | |
| Anweisung 1 | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | |
| Anweisung n | | | | | | | | | | | | | | |
| Einfachverzweigung | <table><tr><td colspan="2">Bedingung</td></tr><tr><td>wahr</td><td>falsch</td></tr><tr><td>Block 1</td><td>Block 2</td></tr></table> | Bedingung | | wahr | falsch | Block 1 | Block 2 | IF/ELSE | | | | | | |
| Bedingung | | | | | | | | | | | | | | |
| wahr | falsch | | | | | | | | | | | | | |
| Block 1 | Block 2 | | | | | | | | | | | | | |
| Mehrfachverzweigung | <table><tr><td colspan="4">Bedingung</td></tr><tr><td>W1</td><td>W2</td><td>...</td><td>Wn</td></tr><tr><td>B 1</td><td>B 2</td><td>...</td><td>B n</td></tr></table> | Bedingung | | | | W1 | W2 | ... | Wn | B 1 | B 2 | ... | B n | IF/ELSE-IF/ELSE SWITCH-CASE |
| Bedingung | | | | | | | | | | | | | | |
| W1 | W2 | ... | Wn | | | | | | | | | | | |
| B 1 | B 2 | ... | B n | | | | | | | | | | | |
| Verweis | <table><tr><td>Verweis</td></tr></table> | Verweis | Funktionsaufruf | | | | | | | | | | | |
| Verweis | | | | | | | | | | | | | | |
| Schleifen | <table><tr><td>Bedingung</td></tr><tr><td>Block</td></tr></table> <table><tr><td>Block</td></tr><tr><td>Bedingung</td></tr></table> | Bedingung | Block | Block | Bedingung | FOR/WHILE DO-WHILE | | | | | | | | |
| Bedingung | | | | | | | | | | | | | | |
| Block | | | | | | | | | | | | | | |
| Block | | | | | | | | | | | | | | |
| Bedingung | | | | | | | | | | | | | | |

Dazu ein Beispiel:

Mittagessenauswahl

| | | | |
|-----------------------------|-----------------------------------|---|---------------|
| Ein Stück Brot kaufen | | | |
| Brot = knusprig? | | | |
| J | N | | |
| Einen Schokoriegel kaufen | | 2 dl Pouletsauce kaufen | |
| Brot und Schokoriegel essen | | Brot und 2dl Pouletsauce essen | |
| Milch vorhanden? | | | |
| J | N | | |
| Ein Glas Milch kaufen | | Am Brunnen einen Schluck Wasser trinken | |
| heiss | Milch = ? | | |
| Einen Beutel Zucker kaufen | lauwarm | | kalt |
| | Die Milch einem Kollegen schenken | | Milch trinken |
| 10 Minuten Pause machen | | | |

4 Projekte

Diese Zusammenfassung beinhaltet eine Beschreibung der verschiedenen Phasen, die eine gelungene Projektdokumentation benötigt. Es ist zu beachten, dass eine einfache Aufgabe nicht gleich als „Projekt“ bezeichnet wird, doch was versteht man eigentlich unter einem Projekt:

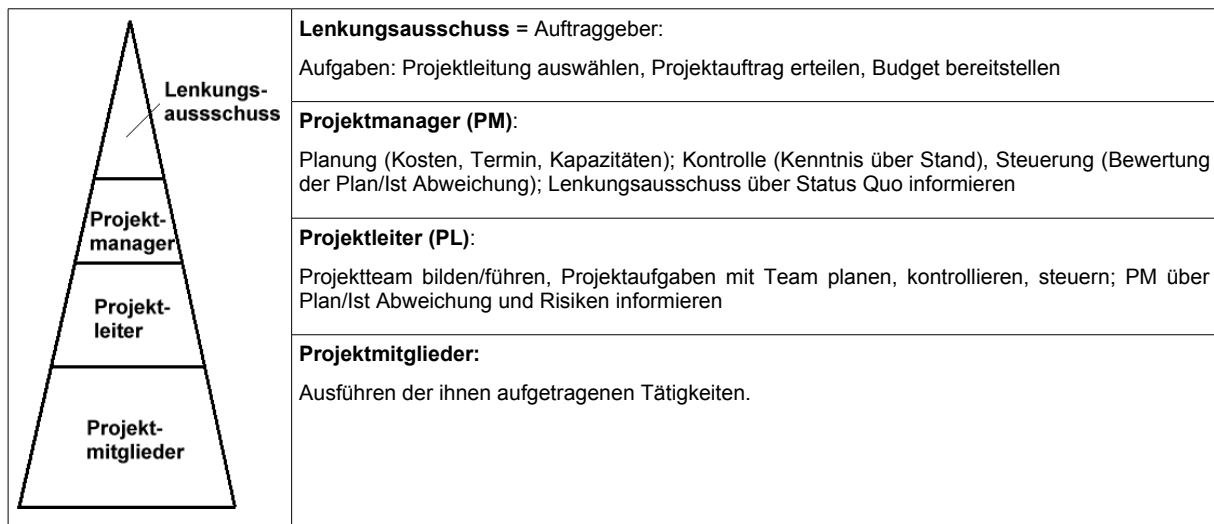
Ein Projekt ist ein Vorhaben, das im Wesentlichen durch Einmaligkeit der Bedingungen gekennzeichnet ist, wie z.B. Zielvorgabe, zeitliche, personelle und andere Abgrenzung gegenüber anderen Vorhaben.

Was macht also eine einfache Aufgabe zu einem Projekt:

- wenn eine inhaltlich konkretisierte Aufgabe vorliegt
- wenn der Aufwand mehr als 50 Personentage übersteigt
- wenn das Kostenvolumen mehr als 50'000 € beträgt
- wenn eine Laufzeit über drei Monate vorliegt

4.1 Projektorganisation

Hinter jedem Projekt steht eine gegliederte Projektorganisation, welche sich in Form einer Pyramide darstellen lässt:



4.2 Phasenkonzepte

Projekte werden in logisch und zeitlich abgrenzbare Schritte (= Phasen) gegliedert. Ein Phasenkonzept ist eine Definition, wie ein Projekt in einzelnen Phasen abgearbeitet werden soll. Sie stellen ein Raster zur Strukturierung des Projektablaufs in inhaltlicher und zeitlicher Hinsicht dar. Da die Bezeichnung der einzelnen Phasen eines Projektes in der Praxis sehr unterschiedlich ist, schafft die folgende Übersicht über die verschiedenen Projektphasennamen:

| Phasenkonzept 1 | Phasenkonzept 2 | Phasenkonzept 3 |
|-----------------|--|---|
| Projektstart | <ul style="list-style-type: none"> • Initialisierung • Vorstudie | <ul style="list-style-type: none"> • Projektvorbereitung |
| Analyse | <ul style="list-style-type: none"> • Grobkonzept • Evaluation | <ul style="list-style-type: none"> • Anforderungserfassung • Variantenanalyse |

| <i>Phasenkonzept 1</i> | <i>Phasenkonzept 2</i> | <i>Phasenkonzept 3</i> |
|------------------------|---|--|
| | <ul style="list-style-type: none"> • Vertragswesen | |
| Design | <ul style="list-style-type: none"> • Detailkonzept | <ul style="list-style-type: none"> • Definition und Planung • Integration und Test |
| Implementation & Test | <ul style="list-style-type: none"> • Realisieren | <ul style="list-style-type: none"> • Realisierung |
| Einführung | <ul style="list-style-type: none"> • Einführung • Nutzung | <ul style="list-style-type: none"> • Markteinführung |
| Projektabschluss | | <ul style="list-style-type: none"> • Projektabschluss |

4.3 Beispiel

4.3.1 Initialisierung & Vorstudie

Initialisierung = Zeitspanne zwischen dem Auftreten einer Idee und dem Entschluss etwas zu unternehmen.

Ergebnis der Initialisierungsphase ist der Projektauftrag:

| Projektauftrag | |
|--|---|
| Projektnummer | P-10198767-3 |
| Projektbezeichnung | Gobret Service Tool 3.0 |
| Aktuelle Situation, Hintergrund | Zur Zeit wird ein neues Feldbussystem (Konnex) entwickelt, das zum neuen europäischen Standard, und somit ein Gegenstück zum amerikanischen CAN-Bussystem, wird. Unsere Firma entwickelt bereits Zentralen und Regler, die auf diesem neuen Bussystem basieren. Den Servicetechniker steht für die neuen Geräte kein Servicetool (PC-Software) zur Verfügung, mit der die Konfiguration der neuen Anlagen bewerkstelligt werden kann. |
| kurze Beschreibung des Projektinhaltes | Die bestehende Service-Applikation Gobret Service Tool 2.0 soll derart erweitert werden, dass damit auch die Inbetriebnahme der neuen NESCOR-Anlagen mittels PC-Software erledigt werden kann. |
| Ziele | Produktziele: <ul style="list-style-type: none"> • Die ganze Funktionalität der Gobret Service Software 2.0 soll auch bei den neuen Konnex-Anlagen zur Verfügung stehen. • etc. • Gobret muss um die automatische Adressvergabe erweitert werden Projektziele: <ul style="list-style-type: none"> • Termin und kostengerechte Umsetzung des Projektes |
| Projektumfang | Das Erstellen der neuen Interface-Box und die Erweiterungen der bestehenden PC-Software gehören vollumfänglich zum Projekt |
| Projektorganisation | GPL: Dennis Jöhr PL-T(Software): Meier Hansjürg. PL-T(Interface): Weber Hanspeter. PL-M: Werner Fuchs |
| Termine / Meilensteine | Projektstart: Schritt 1 "Projektinitialisierung" 02.2003 Schritt 2 "Anforderungserfassung" 04.2003 Schritt 3 "Variantenanalyse" 06.2003 Schritt 4 "Definition (Design / Detailentwurf) " 10.2003 Schritt 5 "Implementation " 02.2004 Schritt 6 "Integration & Systemtest " 04.2004 Projektende 05.2004 Einführung: 08.2004 |
| Kosten | Software (25% Extern) : 2003=2MY; 2004=1,5 MY → 630 kCHF Interface : 2003=2MY; → 310 kCHF <div style="text-align: right;">Total → 940 kCHF</div> Diese Kosten Schätzungen basieren auf folgenden Annahmen: <input type="checkbox"/> Interne MY Kosten = 155kCHF <input type="checkbox"/> Externe MY Kosten = 250kCHF <input type="checkbox"/> Entwicklung durch qualifizierte SW-Entwicklungsingenieure mit Prozess Know how |
| Risiken | Das Standardisierungsgremium Kongre, das die Definition des neuen Konnex-Feldbussystems erarbeitet könnte die bestehenden Definitionen (z.B. Kommunikation) ändern. Durch die hohe Fluktuationsrate in unserer SW-Entwicklungsabteilung könnten Ressourcenengpässe entstehen. |
| Freigabeantrag | GPL: _____ Auftraggeber: _____ _____ |

In der Vorstudie (= erste formell geregelte Projektphase) werden Unklarheiten auf ihre Ursache hin untersucht. Erste Konzeptvorstellungen aus den Ideen der Initialisierungsphase, grob: „Was kostet es?“, „Was nützt es?“

Ebenso wird das weitere Vorgehen geplant, die Projektorganisation festgelegt, Situationsanalysen durchgeführt, Zielformulierungen festgelegt, Lösungsansätze erarbeitet, die Wirtschaftlichkeitsvorschau bewertet sowie der Vorgehensplan für die nächste Phase erstellt.

Das Ergebnis der Vorstudie ist der Vorstudienbericht.

4.3.2 Grobkonzept

EDV-Gesamtkonzept soll erarbeitet werden. man kann danach entscheiden, ob das Projekt durchgeführt oder eingestellt wird. Pflichtenheft (= Grundlage für die Ausschreibung).

Der Aufbau des Pflichtenheftes sieht folgendermassen aus:

- Kundenfirma heute (grob)
- Kundenfirma Zukunft (grob)
- Informatik-Systemkonzept (detailliert): Funktionen und –umfang, -abläufe, Datenmodell, EDV-Plattform, etc.
- Fragen zur Lieferfirma/zu deren Vertragspartnern (Aussehen)
- Fragen zum angebotenen Produkt
- Referenzen
- Fragen zu Preisen/Terminen
- Vertrags- und Lieferbedingungen
- Service und Unterhalt
- Einführungsunterstützung

Funktionsumfang der EDV-Lösung, grundsätzliche Funktionsweise, Hard- und Software-Konzept, Konzept der Datenorganisation, Aufwand, Nutzen, Wirtschaftlichkeit, notwendige Voraussetzungen, Konsequenzen abschätzen und Prioritäten festlegen.

Die Anforderungen sind der wichtigste Bestandteil des Grobkonzept-Dokuments. Das Pflichtenheft kann aus den Informationen des Grobkonzept-Dokumentes erstellt werden, falls Offerten einzuholen sind.

Grob klärt das Grobkonzept-Dokument z.B. die folgenden Fragen:

„WAS... kann die künftige Applikation?“

„WIE... soll die Applikation aussehen?“

Die Anforderungserfassung stellt die wichtigste Aufgabe dieser Phase dar, hier werden v.a. strategische geschäfts- und marktpolitische Ziele identifiziert und abgegrenzt.

Haupttätigkeiten und Kernaufgaben der Anforderungserfassung:

- Marktabklärung, aktuelle Marktsituation veranschaulichen
- Durchführen von Konkurrenzbeachtungen
- Positionierung im Sortiment
- Festlegen der Produkteziele, Ablöseziele, Terminziele, Verkaufsziele
- Durchführen von Konzeptstudien zur Realisierung
- Abklären und aufzeigen der Machbarkeit

Daraus resultieren die folgenden Dokumente:

3. Marktabstimmung
4. Produkterfordernisse
5. Reviewbericht der Produkterfordernisse

Weiterer Bestandteil der Grobkonzeptphase ist die Präferenzmatrix, durch welche man eine Zielgewichtung der Anforderungen an die künftige Applikation festlegen kann.

Die Dokumentation bzw. der Aufbau des Grobkonzeptes vollzieht sich folgendermassen:

- Zusammenfassung: knappe Skizzierung der Ausgangsbasis, Zielsetzung des Projektes, Zweck des vorliegenden Berichts, Vorschlag und weiteres Vorgehen.
- Einleitung/Übersicht: Aufbau des Berichtes, Anleitung zum Lesen, verwendete Unterlagen.
- Situationsdarstellung: Ist-Darstellung der Firma, Firmenstruktur, Materialflüsse, Informationsflüsse, Stärken, Schwächen, Mängelkatalog.
- Zielprojektion: Soll-Darstellung der Firma
- Informatik-Systemkonzepte: Varianten, Übersicht, Applikationsbeschreibungen, erforderliche Technik, Einführung, Betrieb und Unterhalt.
- Beurteilung der Varianten und Empfehlung
- Weiteres Vorgehen bei Annahme der Empfehlung
- Liste von Firmen, die zur Offertenabgabe eingeladen werden sollen
- Beilagen

4.3.3 Evaluation (Variantenanalyse)

Aus den eingegangenen Angeboten (Offerten) sind jene Anbieter auszuwählen, welche die im Pflichtenheft beschriebenen Bedürfnisse und Forderungen am Besten erfüllen.

Als Ergebnis dieser Phase resultiert eine begründete Variantenauswahl des besten Lösungsansatzes/ des besten Anbieters, welche folgendes beinhaltet:

- Erste Darstellung des Systems
- Zusammenfassung der bei der Lösungsfindung verfolgten Alternativen, beschreibt Kriterien, die zur Auswahl einer spezifischen Lösung geführt haben.
- Aussagen zu kritischen Punkten der gewählten Lösung, Machbarkeit, Beschränkungen, Risiken.
- Zusammenfassung der Patentabklärung, Patentwürdigkeit eigener Erfindungen, Massnahmen zur Vermeidung von Verletzung bestehender Patente.
- Ergebnisse der Make-or-Buy-Evaluation (= ist es besser eine Applikation intern selbst zu entwickeln oder eine bereits bestehende Applikation zu nutzen).
- Empfehlung eines Anbieters, falls Offerten eingeholt wurden.

4.3.4 Vertragswesen

Vertragsformulierung, Vertragsverhandlungen und Vertragsabschluss.

4.3.5 Detailkonzept

Es beginnt die *Design-Phase* („WIE...?“).

Gesamtkonzept auf der Basis einer Systementscheidung detaillieren, spezifizieren.

Ergebnisse:

- Systemhandbuch (Systembeschreibung):
 - Inhaltsverzeichnis

- Definitionen
- Kurze Vorstellung des Gesamtkonzeptes, Funktion des Detailkonzeptes in diesem Rahmen
- Detail-Datenflusspläne
- Hard- und Softwareanforderungen
- Eingabe
- Verarbeitung
- Ausgabe
- Sicherheitskonzept
- Verarbeitungstermine und – häufigkeiten
- Testkonzept
- Programmvorgaben:
 - Ablaufstrukturen und verbale Programmbeschreibungen für jede Applikation
 - Betroffene Datenbestände und deren Struktur
 - Definition der Schnittstellen zu anderen Programmen
 - Besondere Verarbeitungsvorschriften, Rechenformeln, Plausibilitäts- und Prüfvorschriften
 - Entwurf der Bildschirmmasken (GUI) und Listenbilder
 - Testbeispiele, Testdaten und Testanforderungen
 - Normen und Anforderungen and die Programmdokumentation, etc.
- (Grundlagen für Rahmenorganisation).

4.3.6 Realisieren & Testen

Installation der Hardware, Programmierung, Erstellen, Testen, Dokumentieren.

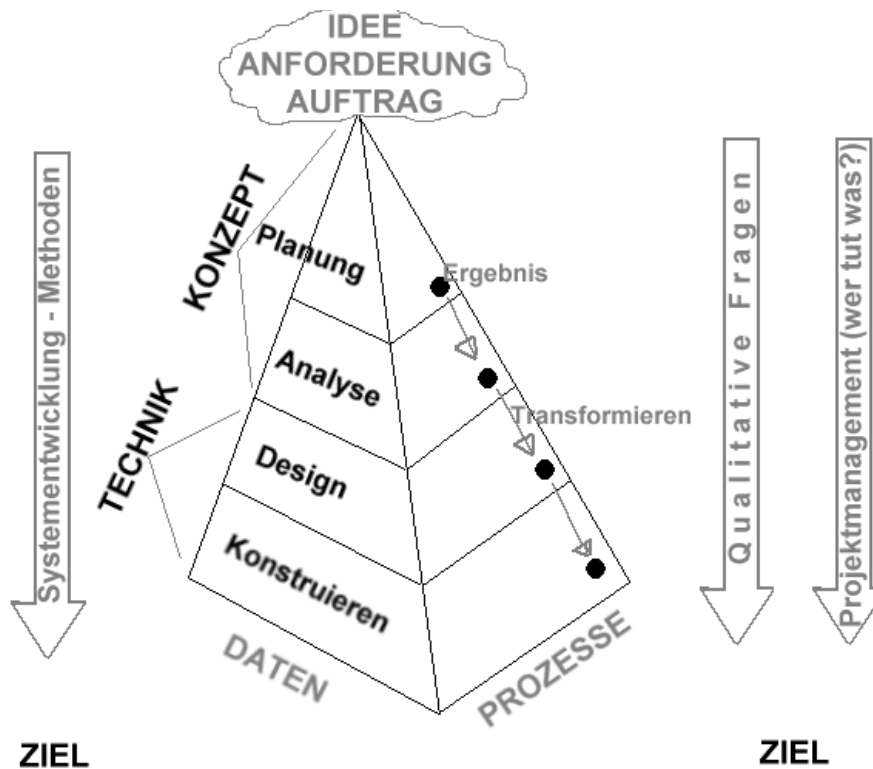
4.3.7 Einführung

Übergabe der funktionierenden EDV-Lösung.

Das Phasenkonzept ist eine Empfehlung, die den gegebenen Umständen aber angepasst werden muss.

1.3.2 Problemlösungspyramide

Die Problemlösungspyramide zeigt, dass die Daten und Prozesse im Laufe der Entwicklung eines Projektes immer breiter werden. Sämtliche Phasen setzen ein daraus resultierendes Ergebnis voraus, bevor zur nächsten Phase/Schicht vorangeschritten wird.



Damit ein Projekt überhaupt in die nächste Entwicklungsphase gelangen kann, muss eine Freigabe der aktuellen Entwicklungsphase erfolgen. Für jede Freigabe wird die Erstellung bzw. die Nachführung folgender Dokumente verlangt.

3. Freigabeantrag
4. Projektstatusbericht
5. Freigabeprotokoll

4.4 Vorteile des phasenweisen Vorgehens

- Überblick wird gewahrt und die Zusammenhänge sichergestellt.
- Das Risiko einer Fehlentwicklung wird verringert.
- Der Benutzer muss mitarbeiten und Stellung beziehen.
- Die Entscheidungsfreiheit des zukünftigen Anwenders wird gewahrt.
- Der EDV-Spezialist soll schrittweise mehr Verantwortung tragen.

5 Referenzen

Für die Erstellung dieses Dokuments haben folgende Bücher als Referenz gedient.

| <i>Titel</i> | <i>Verlag</i> | <i>ISBN</i> | <i>Website</i> |
|--|---------------|---------------|--|
| C – kurz & gut | O'Reilly | 3-89721-238 | www.oreilly.de |
| Visual C#.Net – Grundlagen und Profiwissen | Hanser | 3-446-22021-6 | www.hanser.de |