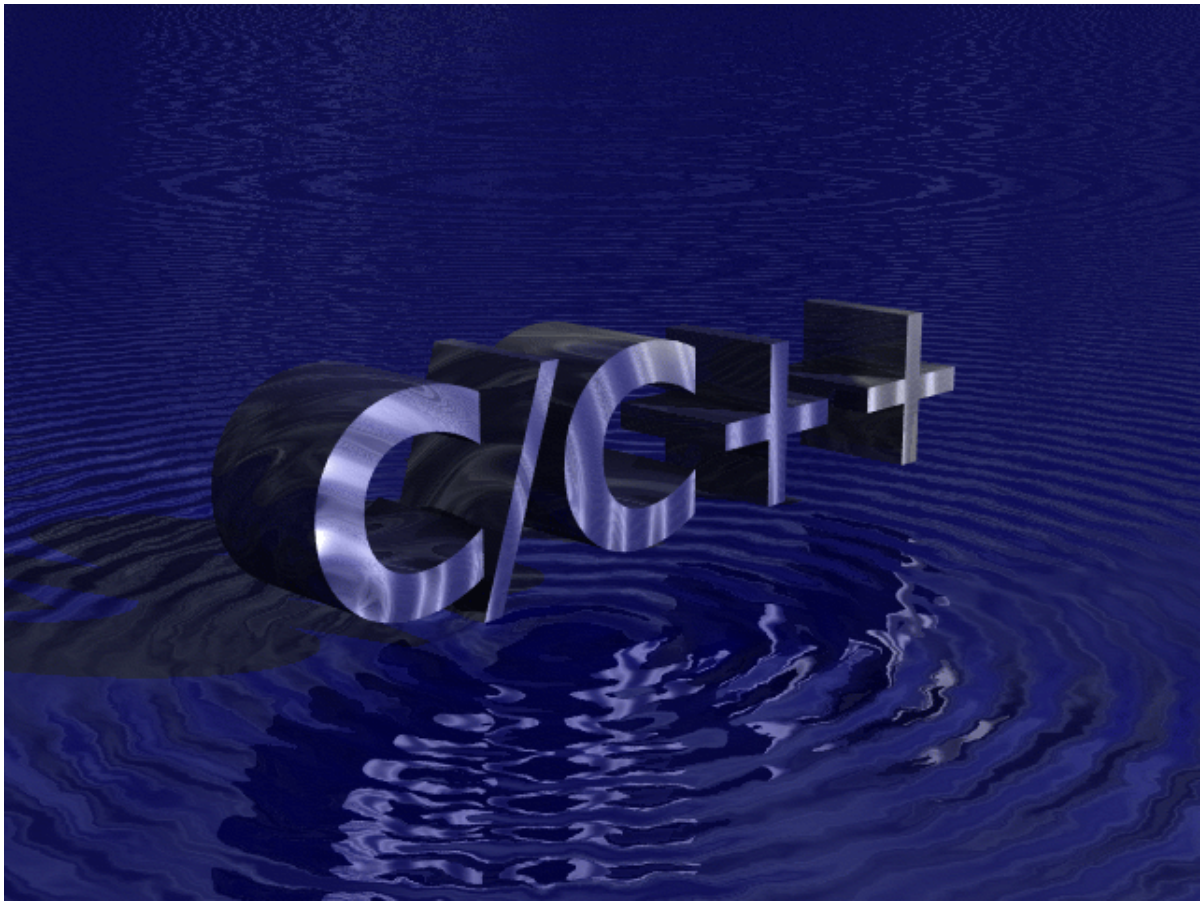
Programmieren in C/C++

von Michael Weitzel

Ein Kurs für Anfänger

Ausgabe vom 30. September 2001



Inhaltsverzeichnis

1 Vorwort

C++ ist DIE Standard-Programmiersprache. C++ bildet die Obermenge von C, Java, PHP, Perl und anderen Programmier- und Scriptsprachen. D.h., wenn man in C++ programmieren kann, ist es ein Katzensprung zu Java, zu dynamischen Internet-Seiten mit PHP und zu Perl, eine der beliebtesten und mächtigsten Script-Sprachen.

Je empfehle den Einsatz der GNU-Entwicklungsumgebung unter Unix/Linux bzw. die hervorragenden Windows32-Versionen wie DJGPP oder CygWin. Der große Vorteil dieser Entwicklungsumgebung ist zunächst die Tatsache, daß es sich um freie Programme handelt, die niemand von uns bezahlen braucht. Davon abgesehen ist der von uns verwendete C++ - Compiler einer der besten, der überhaupt erhältlich ist.

Dieser C/C++ - Kurs richtet sich an Anfänger, die möglicherweise noch nichts mit dem Thema Programmieren zu tun hatten. Dieses Skript ist das Begleitmaterial zu einem Abendkurs, den ich im Vereinsraum des LDKnet e.V. gehalten habe. Den Vereinsraum mußten wir aus finanziellen Gründen abgeben, weswegen sich das mit dem Kurs leider auch erledigt hatte. Ich will in Zukunft versuchen, noch ausstehende Punkte zu schreiben und Teile zu verbessern. Falls ein Sachverhalt zu knapp oder unverständlich dargestellt ist, bitte ich um Rückmeldung. Für Verbesserungen und Korrekturen bin ich dankbar. Da ich selbst schon etwas länger mit dem Thema zu tun habe, fehlt mir möglicherweise manchmal der richtige Blick für Probleme. Da ich kein Buch schreiben will (gute Bücher gibt's genug ;-), kann ich natürlich nicht auf die Details der C-Befehlsbibliothek oder des C++ STL eingehen - das würde den Rahmen sprengen. Man kann viel lernen, indem man sich einfach fertige Quellcodes anschaut. Linux-Benutzer, die eine Dokumentation zu einem Befehl der Befehlsbibliothek suchen, sollten einen Blick in die man-Pages werfen. Die Windows-Leute finden eine ähnliche Dokumentation vor (einfach im RHIDE mit dem Cursor mal auf den unbekannten Befehl gehen und Strg-F1 drücken). Der Nachteil hierbei ist, daß es diese Hilfe nicht in deutsch gibt. Wer kein Englisch kann, sollte sich vielleicht später ein Buch zulegen.

Bei manchen Themen, besonders bei den „Pointern", erwarte ich Verständnisprobleme. Das Programmieren eines Computers hat nicht allzuviel mit der „natürlichen Welt" zu tun. Als Anfänger braucht man eine gewisse Zeit, um sich die neue Denkweise anzueignen. Auch, wenn man möglicherweise durch diesen Kurs nicht zum Profi wird, wird man auf jeden Fall etwas Wichtiges mitnehmen: Einen erweiterten Horizont.

Viel Spaß!

1.1 Hinweis

2 Die Grundlagen: C

Bevor wir mit C++ beginnen, müssen wir etwas mit C vertraut werden. C bildet die Untermenge von C++. Bevor wir lernen C++ - Programme zu schreiben, benötigen wir die Grundlagen der Programmiersprache C. Das in diesem Abschnitt Gelernte läßt sich zu 100% in C++ wiederverwenden.

2.1 Das erste C - Programm Schritt für Schritt

Ein C - Programm ist normalerweise eine Ansammlung von Funktionen[1], Deklarationen und Definitionen. Aller Anfang ist leicht. Unser erstes Programm besteht aus nur wenigen Zeilen Quelltext[2] und ist zu nichts zu gebrauchen:

```
// Ich bin eine einzelne Kommentarzeile

void main(void)
{ // hier beginnt der Block

        /* hier steht
           normalerweise
           der Programm-Code.
           In diesem Fall ist
           das ein mehrzeiliges
           Kommentar */

} // Ende des Blocks
```

Geschweifte Klammern (**{**, **}**) umschließen einen *Block*. Innerhalb eines solchen Blockes stehen die eigentlichen Anweisungen. Die Kommentare sind wichig. Sie verbessern die Lesbarkeit des Quelltextes und beantworten die häufig gestellte, aber deswegen nicht weniger peinliche Frage: *„Was habe ich eigentlich damals da gemacht?“*. In C/C++ kann man auf zwei Arten ein Kommentar in seinen Quelltext aufnehmen:

1. Alle Zeichen, die hinter `//` stehen, sind Teil des Kommentars. Dieses Kommentar reicht nur bis zum Zeilenende.

2. Alle Zeichen, die hinter `/*` und vor `*/` stehen, sind Teil des Kommentars. Dieses Kommentar darf über mehrere Zeilen hinweg benutzt werden.

Methode Nr. 2 hat den Vorteil, daß sich mit ihr auch Kommentare des Typs 1[3] *auskommentieren* lassen. Man kann also einen mit `//` kommentierten Programmteil in `/*` und `*/` einschließen und damit komplett auskommentieren.

Wir erweitern unser erstes Programm und geben dem Block Inhalt:

[1] Funktionen kann man auch als Unterprogramme oder Prozeduren bezeichnen, wenn sie keinen *Rückgabewert* haben

[2] engl. *source code*

[3] genaugenommen ist die erste Methode nicht Teil des ISO/ANSI Standards und wird erst in C++ offiziell eingeführt. Ich habe bislang noch keinen C-Compiler gesehen, der nicht beide Methoden unterstützt

```
#include <stdio.h>

void main(void)
{
        printf("Hallo ");
        printf("Welt\n");
}
```

Dieses Programm schreibt *„Hallo Welt“* auf den Bildschirm. **printf** ist eine *Funktion*. Innerhalb der runden Klammern stehen die *Parameter*[4], die der Funktion **printf** übergeben werden. Eine Zeichenkette wie *“Hallo“* wird *string* genannt. Die Funktion **printf** wird zweimal aufgerufen, bis die Nachricht auf dem Bildschirm erscheint. **n**[5] repräsentiert ein Sonderzeichen, das einen Zeilenumbruch[6] auslöst. Eine Übersicht über die von C/C++ unterstützen Escape-Codes ist in der ASCII-Code-Tabelle (Tabelle 6, Spalte ESC) im Anhang zu finden.

Jede ordentliche Funktion muß irgendwo definiert werden. Auch **printf** ist nicht einfach vom Himmel gefallen, sondern wird in der *Headerdatei*[7] stdio.h definiert. Damit wir **printf** benutzen können, binden wir ganz oben in unserem Programm, mittels der *Präprozessordirektive* #**include**, die Definition dieser Funktion ein. Der Header stdio.h enthält die Definitionen für viele andere Funktionen, die zur Ein- und Ausgabe von Daten benutzt werden können. Bei Headern handelt es sich um normale C-Quelltexte. Es lohnt sich, einen Blick in diese Dateien zu werfen - sie befinden sich im Unterverzeichnis include (unter Linux: /usr/include). Der Linux-Benutzer erhält mit dem Kommando man 3 stdio eine Übersicht über die stdio-Funktionen.

2.2 Variablen und einfache Typen

Den Speicher eines Computers kann man sich wie einen riesigen Schrank mit beinahe unzählig vielen Schubladen vorstellen. Diese Schubladen sind Speicherbereiche, in denen man Zahlen, Buchstaben, Zeichenketten und Vieles mehr ablegen kann. Eine solche Schublade wollen wir in Zukunft als Variable bezeichnen, da man ihren Inhalt verändern[8] kann. Indem eine Variable deklariert wird, wird sie dem Programm bekannt gemacht. Wie dies geschieht, wird in einem Bespiel gezeigt.

Es ist in den meisten Programmiersprachen von entscheidender Bedeutung, welchen Typ (z.B. Zeichenkette, Zahl, ...) eine Variable hat. C/C++ verfügt über einen Satz von *eingebauten*[9] Typen (Tabelle 1). Die in der Tabelle angegebenen Größen sind mit Vorsicht zu genießen, da diese nicht im ANSI-Standard spezifiziert sind und je nach verwendetem Compiler oder Prozessor variieren.

---

[4] auch *Argumente* genannt
[5] sprich „Escape n“
[6] auch *line feed* oder kurz *LF* genannt; ASCII-Zeichen-Code 10
[7] auch kurz *Header* genannt
[8] *Variable = Veränderliche*
[9] engl. *intrinsic types*

Der Typ char und alle int-Typen können durch die vorangestellten Bezeichner unsigned (vorzeichenlos[10]) und signed (vorzeichenbehaftet[11]) ergänzt werden. Standardmäßig liegen char und int als vorzeichenbehaftet vor, ohne daß ein signed vorangestellt werden muß. double und float sind grundsätzlich vorzeichenbehaftet, was sich auch nicht ändern läßt.

Der Typ void nimmt eine Sonderstellung ein. *void* heißt soviel wie *leer* oder *nichts* und wird oft als Rückgabewert für Funktionen benutzt, die nichts zurückgeben sollen - sozusagen als Platzhalter. Der aufmerksame Leser hat *void* schon in unserem ersten Programm bemerkt. *void* stand dort direkt vor *main*. Wir haben dort - ohne es zu wissen - eine Funktion *main* definiert, deren Rückgabewert *void* war. Später dazu mehr... zunächst ein Beispiel:

```
#include <stdio.h>  // wir brauchen die Funktion "printf"

// Zwei vorzeichenbehaftete int-Variablen mit Namen sa und sb.
// Die sb bekommt gleich den Wert -3 zugewiesen
int sa, sb = -3;
// eine vorzeichenlose int-Variable usc
unsigned int usc;
// eine vorzeichenlose int-Variable usd
unsigned usd;

// eine float-Variable pi, vorbelegt mit dem Wert 3.1415
float pi = 3.1415;

void main(void)
{
        // eine weitere Variable, vorbelegt mit dem Wert von Variable
        // pi
```

[10] kann nur positive Werte annehmen
[11] kann positive und negative Werte annehmen

Typ	Abkürzung	Bedeutung	Größe
char	-	ein Zeichen / eine ganze Zahl	8 Bit
short int	short	kleiner Integer (ganze Zahl)	8 Bit
int	signed	Integer (ganze Zahl)	16/32 Bit
long int	long	großer Integer (ganze Zahl)	32 Bit
long long int	long long	sehr großer Integer (ganze Zahl)	64 Bit
float	-	rationale Zahl, einfacher Genauigkeit	32 Bit
double	-	rationale Zahl, doppelte Genauigkeit	64 Bit
long double	-	double mit doppelter Genauigkeit	80/96 Bit
void	-	Benutzung als pseudo-Rückgabewert oder universal-Zeiger (void*)	???

Tabelle 1: Eingbaute Datentypen

```
        double blabla = pi;

        // sa den Wert 2000 zuweisen
        sa = 2000;
        // dem vorzeichenlosen usc den Wert -3 zuweisen *AUTSCH!*
        usc = sb;

        printf("Wir schreiben das Jahr %i.\n", sa);
        printf("In Variable sb steht der Wert %i.\n", sb);
        printf("Die Zahl Pi hat etwa dem Wert %f.\n", pi);
        printf("In Variable blabla steht der Wert %f.\n", blabla);
        printf("In Variable usc steht der Wert %u ... AUTSCH!\n", usc);

        { // wir oeffnen einen weiteren (sinnlosen) Block...
                // und deklarieren eine weitere Variable
                double e;

                e = 2.718;
                printf("Die Zahl e hat den Wert %f.\n", e);
        } // ende des Blocks
}

/*
Die Ausgabe des Programms:

Wir schreiben das Jahr 2000.
In Variable sb steht der Wert -3.
Die Zahl Pi hat etwa dem Wert 3.141500.
In Variable blabla steht der Wert 3.141500.
In Variable usc steht der Wert 4294967293 ... AUTSCH!
Die Zahl e hat den Wert 2.718000.
*/
```

Hier sehen wir zunächst den eigentlichen Verwendungszweck von **printf** - die formatierte Ausgabe von Text und Zahlen. Weiterhin sehen wir, wie man Variablen deklariert und ihnen bei Bedarf gleich einen Wert zuweist. In dem Programm wurden nur Variablen benutzt, die vorher deklariert wurden. In C/C++ dürfen Variablen erst dann benutzt werden, wenn sie vorher deklariert wurden. In C dürfen Variablen außerhalb von *main* (wie bei sa, sb, usc, usd und pi gezeigt) und zu Beginn jeden Blockes **vor** Anweisungen und Funktionsaufrufen deklariert werden. C++ gibt dem Programmierer hier etwas mehr Freiheit, da auch zwischen Funktionsaufrufen und Zuweisungen neue Variablen deklariert / erzeugt werden dürfen. Hätten wir nicht den weiteren, als *sinnlos* bezeichneten, Block im unteren Teil geöffnet, hätten wir an dieser Stelle die Variable e nicht deklarieren dürfen! Wichtig ist auch noch zu erwänen, daß in C/C++ Variablen nicht initialisiert[12] werden. Der Wert einer Variable ist also undefiniert, bis ihr ein Wert zugewiesen wird!

[12] mit einem Start-Wert belegt

2.2.1 Strukturen (structs)

Strukturen - besser vielleicht *structs* genannt - entstehen, wenn man mehrere Variablen gruppiert. Möchte man beispielsweise einen Schüler beschreiben, könnte man folgenden struct anlegen:

```
struct schueler
{
        char    name[100];
        int     alter;
        int     klasse;
        int     note;
} ein_schueler;

void main()
{
        struct schueler nochein_schueler;
        struct schueler *schueler_ptr;

        schueler_ptr = &nochein_schueler;

        ein_schueler.alter = 7;
        ein_schueler.klasse = 2;
        nochein_schueler.alter = 6;
        nochein_schueler.klasse = 1;
        schueler_ptr->alter = 8;
        schueler_ptr->klasse = 3;
        // ...
}
```

In diesem Beispiel wird der struct *schueler* definiert. Vor dem abschließenden Semikolon der Definition steht *ein_schueler*. Damit wird gleich ein struct von Typ *schueler* erzeugt. *ein_schueler* hätte auch weggelassen werden können. Im Hauptprogramm wird gezeigt, wie man weitere structs vom Typ *schueler* erzeugt und auf die intern gespeicherten Variablen zugreift. *schueler_ptr* ist ein *Pointer* auf einen den struct *schueler* - später mehr dazu.

2.2.2 Varianten (unions)

Wenn man von einem struct mit mehreren Variablen unterschiedlichen Typs mit Sicherheit weiß, daß man immer nur eine Variable benötigt, sind die restlichen Variablen redundant[13] und verschwenden Speicherplatz. Für diesen Spezialfall gibt es in C/C++ die union. Hierbei werden die verschiedenen Variablen im gleichen Speicherplatz abgelegt. Ändert man also eine Variable, ändern sich gleichzeitig alle anderen Variablen. Unions werden sehr selten verwendet - oft in der harwarenahen Programmierung, wo es gilt Speicherplatz zu sparen. Da die Einsparung von ein paar Bytes heute nicht mehr allzu wichtig ist (Speicherplatz ist billig), spielen unions kaum noch eine Rolle. Die Syntax

[13] d.h. überflüssig

ist analog zu den structs[14].

2.2.3 Aufzählungen

Mit einer Aufzählung[15] wird eine Menge von durchnummerierten Konstanten definiert. Beispiel:

```
...
enum wtage { MO, DI, MI, DO, FR, SA, SO };
...
```

Die einzelnen Konstanten MO bis SO haben einen int-ähnlichen Typ. Der Compiler weist den Konstanten eine bei 0 beginnende, aufsteigende Nummerierung zu - MO besitzt den Wert 0, SO den Wert 6. Der komplette Enumerationsausdruck stellt einen neuen Typen dar, mit dem Variablen definiert werden können. Diesen Variablen kann man Werte zuweisen, die innerhalb ihres Wertebereiches liegen - im Beispiel die Zahlen 0 bis 6, oder die Konstanten MO bis SO. Die Nummerierung läßt sich durch Zuweisungen beeinflussen:

```
enum wtage { MO, DI, MI, DO, FR, SA, SO }; // MO=0, DI=1, ..., SO=6

// Alle Konstanten definieren
enum temp { KALT=10, KUEHL=20, NORMAL=30, WARM=40, HEISS=50 };

enum { HEUTE=1, MORGEN }; // HEUTE=1, MORGEN=2

int main()
{
        enum wtage tag;          // tag ist von enum-Typ wtage

        tag = DO;                // tag den Wert 3 zuweisen
        tag = 6;                 // ok, weil SO=6
}
```

2.3 Felder

2.3.1 Eindimensionale Felder

Oft kann man mit einzelnen Variablen nicht allzu viel anfangen, weil man eine Datenstruktur benötigt, die viele Variablen des selben Typs speichern soll. Will man beispielsweise 100 float-Variablen speichern, greift man auf eine ganz einfache Datenstruktur zurück, die dies elegant ermöglicht - das Feld[16]. Das folgende Beispielsprogramm illustriert die Verwendung.

```
#include <stdio.h>

void main(void)
{
```

[14] anstelle des Schlüsselwortes **struct** wird das Schlüsselwort **union** verwendet
[15] engl. *enumeration*
[16] engl. *array*

```
        // deklariere ein Feld von 5 int-Zahlen
        int feld_int[5];

        // deklariere ein Feld von 5 float-Zahlen,
        // Eintraege 0 bis 4 werden vorbelegt
        float feld_float[] = {1.2, 3.4, 5.6, 6.7, 7.8};

        // deklariere ein Feld aus 6 Zeichen
        char feld_char1[6] = {'H', 'a', 'l', 'l', 'o', '\0'};

        // deklariere ein Feld aus 5 Zeichen
        char feld_char2[5] = "Welt";

        // belege feld_int mit den Werten 1, 2, 3, 4 und 5
        feld_int[0] = 1;        feld_int[1] = 2;
        feld_int[2] = 3;        feld_int[3] = 4;
        feld_int[4] = 5;

        printf("Der erste Eintrag von feld_float ist %f\n",
                feld_float[0]);

        printf("Der letzte Eintrag von feld_float ist %f\n",
                feld_float[4]);

        printf("Der beruehmte Satz lautet: %s %s!\n",
                feld_char1, feld_char2);
}

/*
Die Ausgabe des Programms:

Der erste Eintrag von feld_float ist 1.200000
Der letzte Eintrag von feld_float ist 7.800000
Der beruehmte Satz lautet: Hallo Welt!
*/
```

Die Syntax sollte klar sein. Ein Feld wird wie eine normale Variable deklariert - mit dem Unterschied, daß hinter der eigentlichen Bezeichnung in eckigen Klammern eine Zahl steht. Diese Zahl gibt die Größe des Feldes an. feld_int ist demnach ein Feld aus fünf Integer-Variablen.

Bei feld_float wird gezeigt, wie man die Einträge schon bei der Deklaration mit Werten vorbelegen kann. Die Angabe der Feldgröße kann hierbei weggelassen werden - der Compiler zählt die Einträge der nachfolgenden Initialisierungsliste und kennt deshalb die Größe von feld_float.

Wie in Tabelle 1 schon erwähnt, dient der Typ char zur Speicherung von Zeichen (z.B. einzelnen Buchstaben). feld_char1 ist ein sechselementiges char-Feld von Zeichen. Wenn

man ein Zeichen in einer char-Variablen speichern möchte, den ASCII-Code des Zeichens aber nicht parat hat, kann man das Zeichen direkt, eingefaßt in Abostrophen (z.B. 'H'), eingeben. Alternativ kann man auch die ASCII-Codes der Zeichen benutzen (die Zahlencodes sind der ASCII-Tabelle im Anhang entnommen). Das Resultat ist vollkommen gleichwertig.

```
...

// deklariere ein Feld aus 6 Zeichen
char feld_char1[6] = {72, 97, 108, 108, 111, 0};

...
```

feld_char2 zeigt eine weitere Alternative. Wir hatten das vorher *string* genannt. Es handelt sich um eine Zeichenkette, wie sie z.B. der printf-Funktion übergeben wird. Es ist nun leicht zu erraten, daß es sich auch bei feld_char1 um einen string gehandelt haben muß. Wer die Buchstaben des Wortes *„Welt“* nachzählt, wird zu dem Schluß kommen, daß das Feld mit fünf Elementen zu groß dimensioniert wurde. Dem ist aber nicht so: Wäre mein Feld nur vier Elemente groß, würde mein Programm möglicherweise abstürzen. Das letzte Element von char_feld1 ist '\0' bzw 0. Dieses Element wird die *Terminierung* des string genannt. Die Terminierung signalisiert das Ende des strings. Ohne diese Terminierung würde z.B. bei der Ausgabe des strings über das Ende hinausgelesen werden. Das Resultat wäre ein Bildschirm voller piepender Sonderzeichen und mit etwas Pech anschließend ein Programmabsturz. Wird ein string mit Hilfe der Anführungszeichen deklariert, wird die Terminierung von Compiler automatisch und stillschweigend angehängt. Auch wenn der Compiler so nett ist, dem Programmierer diese Arbeit abzunehmen, solle man niemals das unsichtbare '\0' am Ende des strings vergessen (es rächt sich bitter).

Bei feld_int wird gezeigt, wie man den Feld-Elementen einzeln, nach der Deklaration, Werte zuweisen kann. Man greift auf ein spezielles Element des Feldes zu, indem man seine Nummer in eckigen Klammern hinter dem Feldnamen angibt. Wichig ist hierbei, daß das erste Element nicht etwa Element 1 ist, sondern Element 0. In gleicher Weise kann man auch auf die Elemente / Zeichen eines strings zugreifen, da es sich ja bei einem string auch nur um ein Feld handelt.

2.3.2 Mehrdimensionale Felder

Die bisher behandelten Felder waren eindimensional. Für speziellere Aufgabenstellungen benötigt man manchmal mehrdimensionale Felder. Will man beispielsweise ein gescanntes Foto speichern, dann hat man es nicht mehr mit einem eindimensionalen Strang von Daten zu tun, sondern mit Bildpunkten, die in Zeilen und Spalten zusammengesetzt ein Bild ergeben. Dieses Bild hat zwei Dimensionen, eine Höhe und eine Breite. Jedes einzelne Pixel[17] hat eine Farbe, die sich wiederum aus einem Rot-, Grün- und Blauanteil zusammensetzt. Dieses farbige Bild hat drei Dimensionen - Höhe, Breite, Farbe. Das folgende Beispiel zeigt, wie man mehrdimensionale Felder benutzt.

[17] von engl. *picture element*; ein Bildpunkt

```
#include <stdio.h>

void main(void)
{
        int feld_2d[3][4] =
        {
                {  1,  2,  3,  4 },
                {  5,  6,  7,  8 },
                {  9, 10, 11, 12 }
        };
        int feld_3d[2][3][2] =
        {
                {   { 1, 1 }, { 2, 2 }, { 3, 3 }   },
                {   { 4, 4 }, { 5, 5 }, { 6, 6 }   }
        };
        int zeile, spalte;

        zeile = 1;
        spalte = 2;
        printf("Eintrag %i ist %i !\n", 7,
                feld_2d[zeile][spalte]);
        feld_2d[zeile][spalte] = 42;
        printf("Eintrag %i ist jetzt %i !\n", 7,
                feld_2d[zeile][spalte]);
}
/*
Die Ausgabe des Programms:

Eintrag 7 ist 7 !
Eintrag 7 ist jetzt 42 !
*/
```

2.4 Bedingungsabfragen mit if

Sehr häufig benötigt man eine Konstruktion, mit der man Befehle in Abhängigkeit einer Bedingung ausführen kann.

```
wenn (Bedingung ist erfuellt) dann
        fuehre_Befehl(e)_aus...
```

...oder...

```
wenn (Bedingung ist erfuellt) dann
        fuehre_Befehl(e)_aus...
ansonsten
        fuehre_andere(n)_Befehl(e)_aus...
```

Eine solche Konstruktion sieht in C/C++ folgendermaßen aus.

```
...
if (a > 0)
        printf("%i ist groesser als 0\n", a);
...
```

...oder...

```
...
if (a > b)
        printf("%i ist groesser als %i\n", a, b);
else
        printf("%i ist kleiner gleich %i\n", a, b);
...
```

Sollen anstatt des einzelnen printf mehrere Befehle ausgeführt werden, faßt man sie zu einem Block zusammen, indem man sie in geschweifte Klammern einschließt.

2.5 Abfragen mit switch & case

Wenn man in Abhängigkeit eines Variablenwertes irgendwelche Anweisungen ausführen will und dabei mehrere mögliche Variablenwerte berücksichtigen muß, kann man das mit mehreren aneinandergehängten if-Bedingungsabfragen erledigen:

```
if (X=='A')                     // wenn X gleich 'A' ist ...
        [...Anweisungen...];
else if (X=='B')                // wenn X gleich 'B' ist ...
        [...Anweisungen...];
else if (X=='C')                // wenn X gleich 'C' ist ...
        [...Anweisungen...];
else
        [...Anweisungen fuer X weder 'A' noch 'B' noch 'C'...];
```

Diese Beispiel ist noch überschaubar. Wenn die Zahl der if-Bedingungsabfragen weiter zunimmt, wird die Sache allerdings unschön und fehleranfällig. Um Abfragen dieser Art etwas übersichtlicher und pflegeleichter zu gestalten, gibt es die switch-Anweisung:

```
switch (X)
{
case 'A':
        [...Anweisungen...];
        break;
case 'B':
        [...Anweisungen...];
        break;
case 'C':
        [...Anweisungen...];
        break;
default:
        [...Anweisungen fuer X weder 'A' noch 'B' noch 'C'...];
}
```

Wie man sieht, entsprechen die Anweisungen hinter dem „Label“ „default:“ den Anweisungen des letzten else-Teils. Die anderen Labels werden in Abhängigkeit des Variablenwertes von X angesprungen. Wichtig ist hierbei, daß man hinter den Anweisungen eines Labels die break-Anweisung nicht vergißt. Wird beispielsweise Label 'B' angesprungen und das break nach den Anweisungen von 'B' wurde weggelassen, geht die Verarbeitung der Befehle nicht etwa hinter der switch-Anweisung weiter, sondern bei den Anweisungen des Labels 'C'.

2.6 Operatoren

Die Operatoren sind die „Rechenzeichen“ von C/C++. Allgemein kann man die logischen (boole'schen) Operatoren, die bitweise logischen Operatoren und die Arithmetik-Operatoren unterscheiden. Einen Überblick über die Arithmetik-Operatoren gibt Tabelle 2. Ein Beispiel soll die Verwendung verdeutlichen.

Operator	Bedeutung	Erläuterung
+	Addition (Plus)	-
-	Subtraktion (Minus)	-
*	Multiplikation (Mal)	-
/	Division (Geteilt)	-
%	Modulo (Rest)	liefert den Rest nach Division

Tabelle 2: Arithmetik-Operatoren

2.6.1 Logische Operatoren

Logische Operatoren finden Verwendung beim Vergleichen und bei Bedingungsabfragen. Eine Verknüpfung dieser Operatoren liefert nur zwei Ergebnisse: 1 oder 0 für „wahr“ oder „falsch“. Im Gegensatz zu anderen Programmiersprachen kennt C keine eigenen Typen für die Aussagen „wahr“ und „falsch“. Auch die Zahlen-Typen, wie z.B. int, können zusammen mit diesen logischen Operatoren benutzt werden. Eine int-Variable, die einen Wert größer 0 hat, ist per Definition „wahr“. Dementsprachen sind Zahlenwerte kleiner oder gleich 0 „falsch“. Eine Übersicht gibt Tabelle 3.

2.6.2 Bitweise logische Operatoren

Die Variablen der Zahlen-Typen (signed/unsigned) char, short int, int und long int bestehen, je nach Typ, aus einer Reihe von Bits. Diese einzelnen Bits stellen atomare „wahr/falsch-Entscheidungen“ dar. Mit Hilfe der bitweise logischen Operatoren lassen sich die einzelnen Bits einer solchen Variable beeinflussen. Tabelle 4 zeigt eine Übersicht. Abbildung 1 erläutert die Funktionsweise anhand eines Beispiels.

```
              11010100                      11010100
bitweise UND  10010111      bitweise ODER   10010111
--------------------------  --------------------------
              10010100                      11010111

              11010100
bitweise XOR  10010111     bitweise NICHT   10010111
--------------------------  --------------------------
              01000011                      01101000


Shift << 2    10010111       Shift >> 2     10010111
--------------------------  --------------------------
              01011100                      00100101
```

Abbildung 1: Bitweise-UND, -ODER, -NICHT, -XOR, Shiftopterationen

Operator	Bedeutung	Erläuterung
!	logisches NICHT (Invertierung)	„nicht a“
&&	logisches UND (Konjunktion)	„a und b“
||	logisches ODER (Disjunktion)	„a oder b“
==	Gleichheit (Äquivalenz)	„a gleich b“
!=	Ungleichheit (Ambivalenz)	„a ungleich b“ bzw. „nicht ’a gleich b’“
$\langle$	kleiner	„a kleiner b“
$\rangle$	größer	„a größer b“
$\langle=$	kleiner gleich	„a kleiner gleich b“
$\rangle=$	größer gleich	„a größer gleich b“

Tabelle 3: Logische Operatoren

Operator	Bedeutung	Erläuterung
$\sim$	bitweise Invertierung	-
&	bitweise UND	-
|	bitweise ODER	-
$\wedge$	bitweise Ambivalenz (XOR)	„ungleich“-Operation
$\langle\langle$	links-Shift	Bits nach links schieben, rechts mit Nullen auffüllen.
$\rangle\rangle$	rechts-Shift	Bits nach rechts schieben, links mit Nullen auffüllen.

Tabelle 4: bitweise Operatoren

2.6.3 Abkürzungen

Häufig benutzte Kombinationen von Operatoren lassen sich in C/C++ (und Perl, Java, PHP, ...) abkürzen. Tabelle 5 zeigt einige dieser Abkürzungen[18] und den *ternären Operator* „?“.

Operation	Abkürzung	Erläuterung
a = a + 1;	a++;	Post-Inkrement
a = a + 1;	++a;	Pre-Inkrement
a = a − 1;	a−−;	Post-Dekrement
a = a − 1;	−−a;	Pre-Dekrement
a = a + b;	a+=b;	Addition mit Zuweisung
a = a − b;	a−=b;	Subtraktion mit Zuweisung
a = a ∗ b;	a∗=b;	Multiplikation mit Zuw.
a = a / b;	a/=b;	Division mit Zuweisung
if (x==y) a=b; else a=c;	a=(x==y) ? b : c;	ternärer Operator

Tabelle 5: Abkürzungen für häufige Operationen

2.6.4 Ein Beispiel

```
#include <stdio.h>

void main()
{
        char i, j, k, l;

        // (1) bitweise UND zwischen k und l
        // (2) logisches UND zwischen j und (k & l)
        i = j && (k & l);
        j = 1; k = 2; l = 3;

        if (j==1 && (k==2 || k==42))
        {
                printf ("k ist entweder 2 oder 42. Auf jeden Fall hat ");
                printf ("j den Wert 1\n");
        }

        i++;                // i = i + 1;
        i*=42;              // i = i * 42;
        i%=4;               // i = i % 4; (Divisionsrest)

        // Unterschied zwischen Pre-/Post-Inkrement
        // (Pre-/Post-Dekrement funktioniert in gleicher Weise)
```

[18] Es gibt noch einige andere, die aber alle dem Schema VARIABLE *OPERATOR*= VARIABLE folgen

```
        i = 42;
        printf ("i ist %i\n", ++i);       // 'i ist 43'
        printf ("i ist %i\n", i++);       // 'i ist 43'
        printf ("i ist %i\n", i);         // 'i ist 44'

        i = 42; j = 1; k = 2; l = 3;

        i = (j>k) ? k : l; // i wird der Wert 3 zugewiesen
}
```

2.7 Schleifen

Schleifen sind die wichtigsten und elementarsten Bestandteile einer Programmiersprache. Schleifen ermöglichen die mehrmalige, an Bedingungen gebundene Ausführung von Befehlen. C/C++ kennt drei Arten von Schleifen. Alle drei Arten sind äquivalent und können sich gegenseitig ersetzen. Diese *Redundanz* soll dem Programmierer das Leben leichter machen.

2.7.1 Die for - Schleife

Die Syntax der for-Schleife ist:

```
for ([Initialisierung];[Lauf-Bedinung];[Inkrement])
        [Befehl oder Block]
```

Im Initialisierungsteil können z.B. Laufvariablen initialisiert[19] werden. Der Befehl oder der Befehlsblock wird nur ausgeführt, wenn die Lauf-Bedingung *wahr* ist. Nach dem ersten und nach jedem weiteren Schleifendurchlauf wird der Inkrement-Teil ausgewertet. Im Inkrement-Teil werden z.B. Variablen herauf- oder heruntergezählt.

2.7.2 Die while - Schleife

Die Syntax der while-Schleife ist:

```
while ([Lauf-Bedinung])
        [Befehl oder Block]
```

Der Befehl oder der Befehlsblock wird solange ausgeführt, solange die Lauf-Bedingung den Wert *wahr* hat. Hat die Lauf-Bedingung schon vor dem ersten Durchlauf den Wert *falsch*, so wird der Befehl bzw. der Befehlsblock nicht ausgeführt.

2.7.3 Die do...while - Schleife

Die Syntax der do...while-Schleife ist:

```
do
        [Befehl oder Block]
while ([Lauf-Bedingung])
```

[19] mit einem Start-Wert belegen

Der Befehl / Befehlsblock wird hier ausgeführt, bevor die Lauf-Bedingung geprüft wird. Hat die Lauf-Bedingung schon vor dem ersten Durchlauf den Wert *falsch*, so wird der Befehl bzw. Befehlsblock **mindestens einmal** ausgeführt.

2.7.4 break und continue

Innerhalb eines Befehlsblockes einer Schleife dürfen die Schlüsselwörter break[20] und continue[21] verwendet werden. Ein Aufruf von break; unterbricht die gerade laufende Schleife. Eine andere Anwendung haben wir bereits bei der switch-Anweisung gesehen. continue; bewirkt, daß die aktuelle Abarbeitung des Schleifenkörpers abgebrochen und mit der nächsten Abarbeitung begonnen wird.

2.7.5 Ein Beispiel

Das folgende Beispiel implementiert den effizienten *Shell-Sort* Sortier-Algorithmus. Alle drei Schleifenarten finden Verwendung. Bei diesem Beispiel handelt es sich um eine Funktion. Funktionen werden im nächsten Abschnitt erläutert.

```
void Shell (int* feld, int elemente) // "Funktionskopf"
{
        int p=elemente, f, i, j, m, tmp;

        while (p>1)
        {
                p/=2;
                m=elemente-p;
                do
                {
                        f=0;
                        for (j=0; j<m; j++)
                        {
                                i=j+p;
                                if (feld[j]>feld[i])
                                {
                                        // Vertausche feld[i]
                                        // mit feld[j]
                                        tmp=feld[i];
                                        feld[i]=feld[j];
                                        feld[j]=tmp;

                                        f=1;
                                }
                        } // Ende der for-Schleife
                }
                while (f>0); // Ende der do... while-Schleife
        } // Ende der while-Schleife
```

[20] engl. *to break = unterbrechen*
[21] engl. *to continue = fortsetzen*

```
} // Ende von void Shell (int*, int)
```

2.7.6 Das goto

Die goto-Anweisung ermöglicht es, an *irgendeine* Stelle im Programm zu springen und die Abarbeitung dort fortzusetzen. Mit der goto-Anweisung läßt sich prinzipiell jede Art von Schleife oder Funktionsaufruf ersetzen. Davon sollte man jedoch möglichst absehen, weil die Verwendung von goto Programme nicht gerade übersichtlich macht (man spricht dann von „Spaghetti-Code"). Mit goto läßt sich jede saubere Strukturierung durchbrechen. Viele Programmierer verdammen deshalb das goto als „Tabu-Anweisung" und verwenden es nicht. In der relativ jungen Programmiersprache Java wurde es aus diesem Grund garnicht erst aufgenommen[22]. Ich persönlich sehe das nicht so extrem und verwende es - wenn auch sehr dosiert und selten. Eine goto-Anweisung besteht immer aus dem eigentlichen goto und einer Sprungmarke:

```
...
goto SPRUNGMARKE;
...
// verschiedene Anweisungen, die uebersprungen werden
...
SPRUNGMARKE:
...
```

2.8 Funktionen

Eine Funktion ist zunächst eine mathematische Konstruktion[23]. C richtet sich im Prinzip nach dieser mathematischen Vorgabe; erweitert diese aber, wodurch dem Programmierer weit mehr Freiheit gegeben wird. Funktionen in C haben die folgenden Eigenschaften:

- Funktionen sind Unterprogramme
- Funktionen verarbeiten eine Reihe übergebener Variablen (Argumente, Parameter) und globaler Variablen zu einem Funktionswert (Rückgabewert), den sie zurückgeben.
- Jede Funktion hat einen Funktionskopf, in dem Argumente und Rückgabewert definiert werden. Dem Funktionskopf folgt ein Block, der den Funktionsrumpf bildet:

  ```
  [Typ des Rueckgabewertes] Funktionsname ( [x1], [x2], ..., [xn] )
  {
  ```

[22] obwohl goto merkwürdigerweise ein Schlüsselwort in Java ist, d.h. ein reservierter Ausdruck, der ansonsten nicht verwendet werden darf.

[23] Nur zur Abschreckung: Wenn x und y zwei variable Größen sind und wenn sich einem gegebenen x-Wert genau ein y-Wert zuordnen läßt, dann nennt man y eine $Funktion\ von\ x$ und schreibt $y = f(x)$. Die veränderliche Größe x heißt *unabhängige Variable* oder *Argument* der Funktion y. Alle x-Werte denen sich y-Werte zuordnen lassen, bilden den $Definitionsbereich\ D$ der Funktion $f(x)$. Die veränderliche Größe y heißt *abhängige Variable*; alle y-Werte bilden den $Wertebereich\ W$ der Funktion $f(x)$.

```
        [Anweisungen, die x1 bis xn verarbeiten...
         ... und den Rueckgabewert bilden]
        return [Rueckgabewert];
}
```

$x1$ bis xn sind durch Kommata abgetrennte Variablendeklarationen der Form:

```
[Typ] [Name]
```

- Zu jeder Funktion kann man einen *Prototypen* angeben:

  ```
  [Typ des Rueckgabewertes] Funktionsname ( [y1], [y2], ..., [yn] );
  ```

  Funktions-Prototypen sehen aus wie Funktionsköpfe. Der einzige Unterschied ist in der Argumentenliste zu finden: $y1$ bis yn dürfen bei Prototypen in zwei verschiedenen Formen angegeben werden:

  ```
  [Typ] [Name]
  ```

  oder...

  ```
  [Typ]
  ```

  Prototypen dienen der Definition der Schnittstelle der Funktion. Den Compiler interessieren nur Rückgabetyp, Funktionsname und Typ und Reihenfolge der Argumente. Wird eine Funktion in einer anderen Funktion aufgerufen und die Definition der *aufgerufenen* Funktion steht (im Quelltext) *hinter* der Definition der *aufrufenden* Funktion, so wird ein Prototyp benötigt, der der *aufrufenden* Funktion vorangeht und die *aufgerufene* der *aufrufenden* Funktion bekanntmacht[24]. Indem wir früher die Headerdatei stdio.h mittels #include eingebunden haben, um printf benutzen zu können, haben wir gleichzeitig den Prototypen der Funktion printf in unser Programm eingebunden. Nicht ohne Grund haben wir die Headerdatei ganz oben eingebunden: Hätten wir sie unterhalb von main eingebunden, wäre sie in main nicht bekannt gewesen!

- Auch main ist eine Funktion. main nimmt allerdings eine Sonderstellung ein, indem es das Hauptprogramm bildet, von dem die Unterprogramme (Funktionen) aufgerufen werden.

- Der Rückgabetyp von main ist void oder int. Bei void spucken allerdings viele Compiler eine Warnung aus, **womit sie vollkommen im Recht sind**: Jedes Programm, daß unter einem Betriebssystem ausgeführt wird, **muß einen Status-Code zurückgeben**. Wenn der Rückgabewert bisher immer void war, so hat der Compiler diesen stets auf int geändert. Der ISO-Standard definiert den Rückgabewert von main als int, weshalb in Zukunft unsere Programme immer den Rückgabewert int haben.

[24] Das ist möglicher nicht so leicht zu verstehen. Wenn man Schwierigkeiten damit hat, sollte man sich das Beispiel genauer ansehen.

- Auch main kann Argumente haben. Diese Argumente dienen zur Übergabe von Kommandozeilenparametern. Der erste Parameter ist eine int-Variable, die die Anzahl der übergebenen Kommandozeilenparameter enthält (im Beispiel argc). Der zweite Parameter ist ein Feld, in dem die Werte der einzelnen Kommandozeilenparameter gespeichert werden (im Beispiel argv). Benötigt man in seinem Programm keine Kommandozeilenparameter, läßt man die Argumentenliste einfach leer, oder schreibt int main(void).
- Die Argumente dürfen innerhalb der Funktion verändert werden. Damit werden allerdings nicht die übergebenen Variablen verändert, da diese vorher gesichert werden (es wird mit Kopien der übergebenen Variablen gearbeitet).

2.8.1 Ein Beispiel

Das Beispielprogramm führt eine Berechnung des Schaltjahres durch. Die Jahreszahl wird dem Programm als Kommandozeilenparameter übergeben.

```
#include <stdio.h>

/*
 * Funktionsprototyp der Funktion schaltjahr. Ohne diesen Prototyp
 * waere schaltjahr in main unbekannt und duerfte nicht benutzt
 * werden.
 */
int schaltjahr (int jahr); // alternativ: int schaltjahr (int);

int main (int argc, char *argv[])
{
        int jahr, ergebnis;

        // Programm wird ohne Parameter aufgerufen -> argc = 1
        // Programm wird mit einem Paramter aufgerufen -> argc = 2
        // usw. ...
        if (argc!=2)
        {
                // argv[0] enthaelt den Namen dieses Programms
                fprintf(stderr, "Fehler:\n\tAufruf mit %s [Jahr]\n\n",
                        argv[0]);
                return 2; // mit Exit-Code 2 aus main aussteigen
        }

        // argv[1] enthaelt den ersten Kommandozeilenparameter
        // in diesem Fall das Jahr in Form einer Zeichenkette
        // sscanf wandelt die Zeichenkette in eine int-Zahl um.
        sscanf(argv[1], "%d", &jahr);

        printf("Das Jahr %i ist .... ", jahr);

        // Aufruf der Funktion schaltjahr. Der Rueckgabewert wird in
```

```
        // der int-Variable ergebnis gespeichert
        ergebnis = schaltjahr(jahr);

        if (ergebnis) // aequivalent zu: if (ergebnis>0)
                printf ("ein Schaltjahr HURRA!!!");
        else
                // implizit gilt im else-Teil: ergebnis<=0
                printf ("kein Schaltjahr :-(");

        printf("\n"); // macht die Ausgabe huebscher...

        // Das Hauptprogramm gibt einen Wert an das Betriebssystem
        // bzw. den Kommandonterpreter/Shell zurueck
        return ergebnis;
}

/*
 * Diese Funktion gibt 1 zurueck, wenn das uebergebene Jahr
 * ein Schaltjahr war; ansonsten wird 0 zurueckgegeben.
 *
 * Algorithmus:
 *   Ein Jahr ist ein Schaltjahr, wenn es glatt durch 4 teilbar ist
 *   und es NICHT glatt durch 100, oder glatt durch 400 teilbar ist.
 */
int schaltjahr (int jahr)
{
        if (!(jahr%4) && (jahr%100 || !(jahr % 400)))
                return 1;
        else
                return 0;
}
```

2.9 Pointer

Bei Pointern handelt es sich um das wichtigste Handwerkszeug eines C/C++ - Programmierers. In Java gibt es sie nicht - zumindest merkt der Programmierer nichts oder nicht viel davon, obwohl er sie implizit ständig benutzt. Pointer sind Zeiger auf Variablentypen. Sie speichern nicht selbst den Wert einer Variablen, sondern *zeigen* auf eine Variable, die einen Wert speichert. Wie funktioniert dieses *Zeigen*? Ganz einfach: Ein Pointer speichert die Speicheradresse der Variablen, auf die er zeigt. Da jede Variable, die wir in unseren Programmen deklarieren irgendwo im Speicher des Computer liegt, hat sie auch eine eindeutige Speicheradresse, die man in einem Pointer ablegen könnte.

Für einen Variablentyp T ist $T*$ ein „Zeiger“ oder „Pointer“ auf T. Das heißt, eine Variable vom Typ $T*$ kann die Speicheradresse (kurz: Adresse) eines Objektes vom Typ T speichern. Ein Beispiel:

```
        char  c = 'a';  // c speichert das Zeichen 'a'
```

```
        char* p = &c;    // p speichert die Adresse von c
        char *p = &c;    // aequivalente Schreibweise
```

Das &-Zeichen vor dem Variablennamen kann man als Adressoperator bezeichnen, denn es liefert in Verbindung mit einer Variablen die Adresse, unter der die Variable im Speicher zu finden ist. Da in einem laufenden C - Programm so ziemlich alles im Speicher liegt, läßt sich auch für so ziemlich alles eine Speicheradresse angeben. Man kann auf Variablen jeden Typs zeigen; auf Felder von Variablen, Strukturen und sogar auf Funktionen und Zeiger selbst:

```
int quadrat(int x)
{
        return x*x;
}

int main()
{
        int      z = 1;
        char     zk[10] = "HalloWelt";

        int*     zahl;                  // Zeiger auf eine int-Zahl
        char*    str;                   // Zeiger auf einen String
        char*    zeigerfeld[10];        // Feld aus 10 char*
        char**   zeichenketten;         // Zeiger auf einen Zeiger, der
                                        // auf ein Zeichen zeigt.
        int      (*funktion)(int);      // Zeiger auf eine Funktion mit
                                        // int-Argument, die int zurueck-
                                        // liefert

        zahl = &z;
        funktion = quadrat;
        str = zk;
}
```

Um von einem Pointer - der Speicheradresse einer Variablen - wieder zur Variablen zu gelangen, muß dessen Adresse zurückverfolgt und der Wert an der Speicherstelle ausgelesen werden. Diesen Vorgang bezeichnet man als *Dereferenzierung*. Durch Dereferenzierung wird erreicht, daß man beispielsweise von einem *int**-Pointer zum Wert der *int*-Variable gelangt, auf die der Pointer zeigt. Auch hierzu ein Beispiel:

```
        char  c = 'a';   // c speichert das Zeichen 'a'
        char* p = &c;    // p speichert die Adresse von c
        char* q = p;     // q speichert jetzt auch die Adresse von c
        char  d = *p;    // p wird dereferenziert und d speichert jetzt
                         // das Zeichen 'a'
        char  e = *q     // q wird dereferenziert und e speichert jetzt
                         // das Zeichen 'a'
        *p = 'b';        // p wird dereferenziert. Der char-Variablen, auf
                         // die p zeigt, wird der Wert 'b' zugewiesen
                         // (die Var. c erhaelt den Wert 'b')
        *q = 'x';        // q wird dereferenziert. Der char-Variablen, auf
```

```
// die q zeigt, wird der Wert 'x' zugewiesen
// (die Var. c erhaelt den Wert 'x')
```

2.9.1 Pointer auf Funktionen

Pointer können nicht nur auf normale Variablen oder structs zeigen, sondern auch auf Funktionen. Diese speziellen Zeiger lassen sich wie alle anderen Pointer einander zuweisen und in Datenstrukturen speichern. Bei der Deklaration eines Funktionspointers müssen neben dem Variablenname die Funktionsargumente und der Rückgabewert angegeben werden:

```
[Rueckgabetyp] (*[Variablenname])([Typenliste der Argumente])
```

Die Benutzung und Zuweisung eines Funktionspointers ist etwas haarig, da es mehrere erlaubte und vom ANSI-Standard tolerierte Möglichkeiten gibt. Beispiel:

```
#include <stdio.h>

int quad(int z)
{
        return z*z;
}

int main()
{
        int a, b;
        int (*funktionA)(int);
        int (*funktionB)(int);

        /* 1. Alternative der Zuweisung */
        funktionA = quad;
        funktionB = quad;

        /* 2. Alternative der Zuweisung */
        funktionA = &quad;
        funktionB = &quad;

        /* 1. Alternative des Aufrufs */
        a = funktionA(9);
        b = funktionB(12);

        /* 2. Alternative des Aufrufs */
        a = (*funktionA)(9);
        b = (*funktionB)(12);

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

Funktionspointer werden eher selten benutzt. Ein typisches Beispiel sind *threads*. Ein *thread* ist ein von einem Programm aufgerufener Subprozeß, der gleichzeitig neben dem

Hauptprogramm und evtl. anderen Subprozessen läuft[25]. Bei der Erzeugung eines solchen Thread wird einer Systemfunktion ein Funktionspointer eines Unterprogramms übergeben, welches den Subprozeß bildet.

Eine andere wichtige Anwendung sind *callback*-Funktionen, die häufig in Verbindung mit GUIs[26] benutzt werden. Wird beispielsweise eine Schaltfläche erzeugt, übergibt man ihr mit Hilfe eines Funktionspointers eine *callback*-Funktion, die aufgerufen wird, sobald die Schaltfläche geklickt wird.

2.10 Typenkonvertierung

Wird einer Variablen ein Wert einer anderen Variablen gleichen Typs zugewiesen, stellt das kein Problem dar, denn beide Variablen verfügen über die gleichen Eigenschaften und die gleiche Speicherkapazität. Schwieriger wird es, wenn man Variablen unterschiedlichen Typs einander zuweist. Um aus einer solchen Zuweisung ein sinnvolles Ergebnis zu ziehen, muß der Compiler eine Typenkonvertierung[27] vornehmen, bevor er die eigentliche Zuweisung ausführt. Prinzipiell lassen sich die folgenden Situationen unterscheiden:

- Die Zuweisung funktioniert ohne Zutun des Programmierers. Der Compiler macht eine *implizite Typenkonvertierung* und es sind keine Probleme zu erwarten. Beispiel: Eine char-Variable wird einer int-Variable zugewiesen. Die 16/32 Bit große int-Variable ist auf jeden Fall groß genug, eine 8 Bit große char-Variable aufzunehmen. Eine Zuweisung ist problemlos möglich.

- Die Zuweisung funktioniert zwar, aber der Compiler spuckt eine Warnung aus. Dies ist z.B. der Fall, wenn man eine float-Variable[28] einer int-Variable zuweist. Bei der Zuweisung werden die Nachkommastellen rücksichtslos abgeschnitten (es wird nicht gerundet!) und ein Teil der Eigenschaften geht durch die Zuweisung verloren. Um den Sourcecode lesbarer zu machen und um die Absicht zu dieser Typenkonvertierung zu bestätigen, verewigt man seine Absicht im Quellcode, worauf auch der Compiler nicht mehr warnt:

  ```
  int   intvariable;
  float floatvariable=3.1415;

  intvariable = (int)floatvariable;
  ```

  Diese Schreibweise bedeutet soviel wie „konvertiere floatvariable nach int, bevor sie intvariable zugewiesen wird“. Hier findet also eine *explizite Konvertierung* statt.

[25] Als Beispiel kann man einen Web-Browser nennen, in dem man in mehreren Fenstern gleichzeitig arbeiten kann.

[26] GUI = *graphical user interface*; eine graphische Bedienoberfläche

[27] engl. *type casting*

[28] also eine rationale Zahl, d.h. „Kommazahl“

- Die Konvertierung funktioniert nicht und der Compiler bricht die Übersetzung ab - selbst, wenn man eine explizite Konvertierung vornimmt. In diesem Fall hat man etwas Fürchterliches versucht - z.B. wollte man eine *float*-Variable in einen int-Pointer konvertieren. Wenn man mit einer solchen Fehlermeldung konfrontiert wird, weiß man, daß man entweder einen Flüchtigkeitsfehler oder einen Denkfehler gemacht hat.

2.11 Modulare Programmierung

Demjenigen, der sich die Quellcodes eines C - Programms etwas genauer angesehen hat, wird nicht entgangen sein, daß er es mit mehreren Dateien zu tun hatte. Diese Dateien stehen in folgendem Zusammenhang:

- Es gibt mehrere .c-Dateien, in denen sich fertig implementierte Funktionen befinden. Diese fertigen Funktionen stehen in thematischem Zusammenhang. Der Autor hat diesen Dateien sinnvolle Namen gegeben, die den Verwendungszweck der enthaltenen Funktionen erahnen lassen.

- Jede .c-Datei bindet eine oder mehrere .h-Dateien (Headerdateien) über die #include-Direktive ein. Zu fast jeder .c-Datei gibt es eine gleichnamige Header-Datei. In dieser Header-Datei werden weitere Header-Dateien eingebunden und es gibt zu manchen oder allen Funktionen der .c-Datei einen Funktionsprototypen in der Headerdatei.

- Makros und Konstanten werden praktisch nur in Headerdateien definiert. Oft findet man eine Art „Konfigurations-Headerdatei“, die voll von irgendwelchen Makro-Definitionen ist. Diese Headerdatei wird von vielen anderen Headerdateien oder .c-Dateien eingebunden, wodurch sie Zugriff auf die Konfigurationsdaten haben.

- Ein Paar aus Header-Datei und .c-Datei kann man ein Code-Modul nennen.

- Funktionen nutzen andere Funktionen aus anderen Code-Moduln. Um dies zu ermöglichen, binden sie die Header-Datei der anderen Code-Moduln ein. Daraus folgt, daß alle Funktionen, die von anderen Code-Moduln benutzt werden sollen, einen Prototypen in ihrer Headerdatei haben müssen. Funktionen, die nur von anderen Funktionen im selben Code-Modul benutzt werden, brauchen also keinen Funktionsprototypen in der Headerdatei. Man könnte hier also zwischen „öffentlichen“ und „privaten“ Funktionen unterscheiden.

- Man findet eine Datei, die den Namen *Makefile* trägt. Diese Datei wird von einem Programm namens *make* ausgewertet. Sie enthält die Regeln, wie die einzelnen Quellcode-Dateien mit Hilfe des Compilers zunächst in .o-Dateien (Object-Dateien) übersetzt werden können und wie diese .o-Dateien entweder zu einem lauffähigen Programm „ge-linkt“[29], oder zu einer Bibliothek zusammengefaßt werden sollen.

[29] d.h. „gebunden“

2.12 Gültigkeitsbereiche von Variablen

Wenn man eine Variable anlegt, sollte man sich darüber bewußt sein, wo diese Variable gültig ist, d.h. benutzt werden kann. Man kann in diesem Zusammenhang zwischen *lokalen Variablen* und *globalen Variablen* unterscheiden.

2.12.1 Lokale Variablen

Eine Variable ist eine lokale Variable, wenn sie in einer Funktion deklariert wurde. Eine solche Variable kann nur innerhalb ihrer Funktion benutzt werden und ist nach außen unsichtbar[30]. Wird eine Funktion nach der Abarbeitung wieder verlassen, werden die lokalen Variablen gelöscht. Eine Ausnahme bilden die Variablen, die mit dem Schlüsselwort static gekennzeichnet wurden. Diese static-Variablen werden bei ihrer Erzeugung in einem speziellen Speicherbereich abgelegt; sie liegen also genauergesagt außerhalb der Funktion. Beim Verlassen ihres Gültigkeitsbereiches werden sie nicht gelöscht, sondern behalten ihren Wert. Daß diese Variablen nicht gelöscht werden, heißt allerdings nicht, daß sie außerhalb der Funktion sichtbar sind - sie sind und bleiben lokale Variablen. Beispiel:

```
#include <stdio.h>

int a = 42;       // Diese globale Variable a hat NICHTS mit dem a aus
                  // aus der Funktion zaehler zu tun!

void zaehler()
{
        static int a=1;          // statische lokale Variable
        int        b=1;          // nicht-statische lokale Variable

        printf("Zaehlerstand a=%i, b=%i\n", a, b);
        a++;
        b++;
        // Die Funktion wird jetzt verlassen und b wird zerstoert.
        // Da a als static gekennzeichnet ist, wird a nicht zerstoert.
}

int main()
{
        zaehler(); // Ausgabe: Zaehlerstand a=1, b=1

        // hier wird  die globale Variable a um 10 erhoeht.
        // man hat keinen Zugriff auf die statische Variable
        // der Funktion zaehler, da diese eine lokale Variable
        // ist!
        a += 10;

        zaehler(); // Ausgabe: Zaehlerstand a=2, b=1
```

[30] Fremde Funktionen können nicht auf diese *lokalen Variablen* zugreifen

```
        // Einen neuen Block oeffnen: Es duerfen wieder Variablen
        // deklariert werden.
        {
                // Deklariere ein lokales a. Diese Deklaration
                // verdeckt das eben benutzte, globale a.
                int a = 111;
                printf("a=%i\n", a); // Ausgabe: a=111
                // Im Block deklariertes a wird jetzt wieder
                // zerstoert.
        }
        printf("a=%i\n", a); // Ausgabe: a=52

        zaehler(); // Ausgabe: Zaehlerstand a=3, b=1
}
```

2.12.2 Globale Variablen

Alle Variablen, die außerhalb von Funktionen deklariert werden, sind globale Variablen. Globale Variablen können in jeder Funktion ausgelesen und verändert werden. Sie werdem beim Start des Programms erzeugt und beim Beenden des Programms zerstört. Wie wir im Teil „Modulare Programmierung“ gesehen haben, kann ein C/C++-Programm aus mehreren Code-Moduln bestehen. Es ist möglich, globale Variablen über mehrere Code-Moduln hinweg zu benutzten. Bei dieser Art Benutzung macht der Anfänger oft folgenden Fehler:

Wird die globale Variable in jedem Code-Modul deklariert, lassen sich die einzelnen Code-Moduln problemlos und ohne Warnung übersetzen. Will man nun die einzelnen Moduln zu einem Programm „verlinken“ kracht es:

```
michael@flame:[~/]$ gcc -c fehler1.c
michael@flame:[~/]$ gcc -c fehler2.c
michael@flame:[~/]$ gcc -o fehler fehler1.o fehler2.o
fehler2.o(.data+0x0): multiple definition of `globaleVariable'
fehler1.o(.data+0x0): first defined here
collect2: ld returned 1 exit status
```

Der Compiler sagt hier, daß die Variable *globaleVariable* mehrfach definiert wurde. Die erste Definition wurde in fehler1.o gefunden - das Duplikat in fehler2.o. Der Fehler ist logisch: Die selbe Variable wurde in fehler1.c und fehler2.c deklariert und benutzt. Aus Sicht des Linkers handelt es sich um zwei unterschiedliche Variablen, die beide den Namen *globaleVariable* tragen - das ist ein Fehler.

Man müßte also die Variable in genau einem Modul definieren und dem Compiler in den anderen Moduln mitteilen, daß man sich auf eine bereits in einem anderen Modul definierte Variable bezieht. Genau das geschieht mit dem Schlüsselwort `extern`, das man der

Variablendefinition voranstellt. extern [typ] [variablenname] ist also nur eine Art Verweis auf eine in einem anderen Modul definierte/deklarierte Variable.

2.12.3 static bei Funktionen und globalen Variablen

Funkionen oder Variablen dürfen im globalen Kontext[31] als static deklariert werden. Die Bedeuting dieser static-Deklaration ist allerdings eine Andere: Diese static-Funktionen und -Variablen dürfen nur innerhalb ihres Code-Moduls benutzt werden. Wer versucht, aus einem anderen Code-Modul auf eine static-Funktion oder -Variable zuzugreifen, wird vom Linker mit einer Fehlermeldung bestraft.

2.13 Eigene Typen definieren

Mit Hilfe des Schlüsselwortes typedef kann man eigene Typen definieren, oder vorhandene Typen umdefinieren:

```
typedef int* meinintzeigertyp;
meinintzeigertyp a;
```

Der int-Pointer ist nach so einer Definition fortan als meinintzeigertyp bekannt. meinintzeigertyp bleibt angenehmerweise kompatibel zu einem echten int-Zeiger:

```
...
meinintzeigertyp b=NULL;
int *c;

c = b;
...
```

Ob so eine Definition sinnvoll ist, bleibt zu bezeifeln. Man definiert sich einen neuen Typen, der ein Pointer ist, aber nicht sofort als Pointer zu erkennen ist, da das Sternchen fehlt. So richtigen Sinn macht typedef erst bei structs. Ein Beispiel aus dem Leben[32]:

```
...
struct node_t
{
        char    *name;
        struct  node_t *left, *right;
};

typedef struct node_t node;
...
```

Hier wird der Typ struct node_t als node definiert, was Schreibarbeit erspart. An der Benutzung der Variablen ändert sich nichts - sie funktionieren immernoch wie structs.

[31] d.h. außerhalb von Funktionen oder Klassen (siehe C++ - Teil)

[32] Bei diesem einfachen struct handelt es sich um einen „Knoten“ einer doppelt verketteten Liste oder eine „Astgabel“ eines Binärbaums. *left* und *right* sind Zeiger auf andere Knoten.

2.14 Makros und Präprozessordirektiven

Präprozessordirektiven sind Anweisungen, die vor der eigentlichen Übersetzung vom Präprozessor[33] verarbeitet werden. Man kann sie an der Raute/Hash „#“ am Zeilenanfang erkennen. Die #include-Anweisung ist auch eine solche Präprozessordirektive, die die Einbindung einer Headerdatei veranlaßt. Mit der Direktive #define lassen sich Makros definieren. Andere Direktiven steuern die Übersetzung auf verschiedenen Plattformen. Wer einen Blick in die Headerdateien seines Compilers wirft, wird ein (organisiertes) „Makro-Chaos“ vorfinden. Ein Beispiel:

```
...
#define VORNAME  "Michael"
#define NACHNAME "Weitzel"
#define ALTER    22

// Abfrage, ob ein Makro definiert ist
#ifdef VORNAME
printf("Sein Vorname ist %s.\n", VORNAME);
// Definition von VORNAME wieder aufheben
#undef VORNAME
#else
printf("Dieser Mensch hat keinen Vornamen!\n");
#endif

// Abfrage, ob ein Makro nicht definiert ist
#ifndef NACHNAME
printf("Er hat keinen Nachnamen :-(\n");
#else
printf("Sein Nachname ist %s.\n", NACHNAME);
#endif

// logische Verknuepfungen von Makros I
#if defined VORNAME && defined NACHNAME
printf("Er hat einen Vornamen und einen Nachnamen:-)\n");
#endif

// logische Verknuepfungen von Makros II
#if defined VORNAME || ( defined NACHNAME && defined ALTER)
printf("Er hat einen Vornamen oder ");
printf("einen Nachnamen und ein Alter\n");
#endif

// Makros mit Parametern
#define MAXIMUM(x,y) ((x>y)?(x):(y))

printf("Maximum von %i und %i ist %i.\n",
       3, 42, MAXIMUM(3,42));
```

[33] wer hätte das gedacht ;-)

```
...
```

Soweit möglich sollte man Makros vermeiden, da sie sich nicht debuggen lassen und die Fehleranfälligkeit erhöhen. Wenn man Makros verwendet, sollte man stets Großbuchstaben verwenden, um damit zu verdeutlichen, daß es sich um ein solches handelt. Headerdateien sollten grundsätzlich mit sogenannten „include-Wächtern“ ausgestattet werden. Diese „Wächter“ verhindern, daß ein Programm, das aus mehreren Moduln besteht, eine Headerdatei mehrmals einbindet. Würden Headerdateien mehrmals eingebunden, würden auch die enthaltenen Variablen mehrmals definiert, was der Compiler als Fehler anzeigt. Ein Beispiel zeigt, wie diese „Include-Wächter“ funktionieren:

```
#ifndef MODULNAME_H
#define MODULNAME_H

/*
 * Hier steht die eigentliche
 * Headerdatei
 */

#endif
```

Ausdrücklich weise ich an dieser Stelle nocheinmal auf die korrekte Verwendung von globalen Variablen hin: Um sicherzustellen, daß eine Variable genau einmal definiert wird, sollte man sie in der .cc bzw. .c - Datei definieren und sie in der zurgehörigen Headerdatei als extern kennzeichnen. Jede andere Quelltext-Datei, die die Headerdatei einbindet, bindet auch die als extern gekennzeichnete Variablendefinition ein. Diese Vorgehensweise vermeidet Fehler und erspart unnötige Sucherei!

2.15 Dynamische Speicherverwaltung

Die Programme, die mit den im Quelltext definierten Variablen auskommen, sind sicher die Ausnahme. Werden beispielsweise irgendwelche Benutzereingaben oder Datensätze eingelesen, deren Anzahl zunächst noch unbekannt ist und sich erst während der Verarbeitung ergibt, hat man zwei Möglichkeiten:

1. Man schreibt die Datensätze in ein Feld, das in jedem Fall groß genug gewählt ist.
2. Man fordert für jeden neuen Datensatz einen Speicherbereich an und speichert in dort.

Methode 1 ist sicher nicht unproblematisch:

- Wie groß ist *groß genug*? Wenn das Feld überläuft, weil seine Kapazität erschöpft ist, kommt es gnadenlos zum Programmabsturz (die Ursache der berühmten *Schreibschutzverletzung* unter Windows oder des *segmentation fault* unter Linux & Co)
- *In jedem Fall groß genug* ist gleichbedeutend mit *im Normalfall zu groß*. Die fehlende Flexibilität wird mit Speicherverschwendung bestraft.

Dagegen hat Methode 2 gewisse Vorteile

- Die maximale Datenmenge hängt nur vom verfügbaren Speicherplatz ab. Unflexibele und kritische Datenstrukturen, wie z.B. Felder, gibt es nicht.
- Der Speicher wird optimal ausgenutzt.

2.15.1 Wie funktioniert dynamische Speicherverwaltung in C?

In C wird in der Headerdatei *stdlib.h* die Funktion malloc definiert. Diese Funktion dient zum *Allokieren*[34] von Speicherplatz. Der Funktionsprototyp der malloc-Funktion hat folgendes Aussehen:

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

Dazu läßt sich folgendes sagen:

- Der Rückgabewert ist vom Typ void*. Dieser Typ ist ein Universalpointer, der sich in jeden anderen Pointertyp konvertieren läßt (über eine explizite Typenkonvertierung).
- Das Argument size ist eine ganze Zahl und gibt die Größe in Bytes des zu reservierenden Speichbereichs an. Der Typ size_t ist Plattform- und Compiler-abhängig. Anstelle von size_t kann man sich etwa unsigned long int denken.

Da es in C keine *Garbage Collection*[35] gibt, muß der reservierte Speicher auch wieder freigegeben werden. Dies geschieht mit der Funktion free. Ein Beispiel soll die Verwendung verdeutlichen:

```
#include <string.h> // fuer strlen, strcpy
#include <stdlib.h> // fuer malloc

/*
 * Funktion zum Duplizieren eines Strings
 */
char * StringDuplikat(const char *todup)
{
        char *kopie;
        int laenge = strlen(todup);

        // Platz fuer die Zeichenkette und den
        // Terminator '\0' anfordern.
        kopie = (char*)malloc(laenge+1);

        // wenn malloc fehlschlaegt ist das
        // Resultat NULL (der Null-Zeiger)
        if (kopie==NULL) return NULL;
```

[34] d.h. zum Anfordern & Zuweisen

[35] Eine Einrichtung, die es z.B. in Java gibt: Reservierter Speicher wird automatisch freigegeben, wenn er nicht mehr benutzt wird.

```
        // den String aus todup in kopie kopieren
        strcpy(kopie, todup);

        // kopie zurueckgeben
        return kopie;
}

int main()
{
        char alt[] = "Hallo Welt!";
        char *neu = StringDuplikat(alt);

        // [mit neu arbeiten]
        // neu wieder freigeben
        free(neu);
        // ...
}
```

Wirklich *dynamisch* war das allerdings jetzt nicht: Wir hatten bereits eine Pointer-Variable, der wir dann nur noch den durch malloc reservierten Speicher zugewiesen haben. Um z.B. eine beliebige Anzahl Datensätze speichern zu können, müssen wir uns schon etwas Besseres einfallen lassen - wie z.B. eine *verkettete Liste* (wir besprechen das in der Übung).

Jetzt stellt sich vielleicht die Frage: *„Woher soll ich wissen, wieviel Bytes ich für eine Variable eines Typs xy belegen muß?“*. Zur Beantwortung dieser Frage gibt es in C den sizeof()-Operator. Diesem Operator wird ein Typ übergeben für diesem er dann die Größe einer Variable dieses Typs ermittelt. Beispiel (Allokation eines int-Feldes mit 256 Einträgen - egal, ob nun 16 oder 32 Bit Integer verwendet werden):

```
int * int_feld = (int *)malloc(256 * sizeof(int));
```

sizeof() sollte immer dann verwendet werden, wenn man sich nicht 100%ig sicher ist, wie groß eine Variable eines Typs ist oder wenn es sich bei dem Typ um einen *zusammengesetzten* Typ handelt (also struct oder union).

3 Eine Generation weiter: C++

Wie der Name schon erahnen läßt, ist C++ eine Weiterentwicklung der Programmiersprache C. C++ ist dabei voll abwärtskompatibel, d.h., ein guter C++ - Compiler sollte jedes C - Programm übersetzen können. C++ ist nicht „besser“ als C, sondern erweitert es um einige sehr grundlegende „Werkzeuge“, die für die Umsetzung einer neuen Denkweise (oder sogar Weltanschauung), die der Objektorientierung, von Bedeutung sind.

3.1 Die Grundideen der Objektorientierung

3.1.1 Objekte und Klassen

In der realen Welt hat man es mit Objekten zu tun. Ein Beispiel: Ein Fernseher ist ein solches Objekt. Wenn ich mir einen Film ansehen möchte schalte ich ihn über die Fernbedienung ein, stelle das Programm ein und genieße. Wie so ein Fernseher eigentlich funktioniert ist mir total egal. Um mit einem Fernseher glücklich zu werden, muß ich nur wissen, wie man ihn bedient. Ein Fernseher hat eine fest definierte Funktion. Ich erwarte nicht, daß er gekühlte Getränke serviert, oder mir die Füße massiert. Weil mich seine Bauteile und die Details, wie er es mit ihnen schafft mir den Film zu zeigen, nicht interessieren, handelt es sich bei diesen Dingen um die Privatsache meines Fernsehers (das *Prinzip der Geheimhaltung*).

Die Mattscheibe und die Fernbedienung stellen die Schnittstelle[36] meines Fernseher-Objektes dar. Über Fernbedienung und Mattscheibe tauscht der Fernseher mit der Außenwelt (mit mir) Informationen aus. Diese Schnittstelle ist die einzige Möglichkeit mit dem Fernseher-Objekt zu kommunizieren.

Selbstredent kann man so einen Fernseher immer wieder verwenden, ohne ihn für jeden neuen Film umbauen zu müssen (das *Prinzip der Wiederverwendung*). Wenn er richtig funktioniert und ich mit seiner Schnittstelle zufrieden bin, brauche ich ihn niemals umzubauen. Wenn ich genug gespart habe, kann ich mir das nächst größere Modell zulegen. Das Luxusmodell hat eine große Bildröhre, perfekten Klang und bessere Farben. Die Schnittstelle ändert sich dabei nicht: Ich sehe weiterhin das Bild über die Mattscheibe und bediene eine Fernbedienung.

In der Fabrik werden Fernseher am Fließband produziert. Es wird nicht jeder Fernseher neu entwickelt, sondern nach einem vorher festgelegten Bauplan konstruiert. Diesen „Bauplan“ wollen wir Klasse[37] nennen. Mit dem Bauplan, den uns die Klasse liefert, konstruieren wir Objekte. Mein Fernseher (-Objekt) wird dann als *Instanz*[38] der Klasse „Fernseher“ bezeichnet.

[36] engl. *interface*
[37] engl. *class*
[38] engl. *instance*

3.1.2 Vererbungshierarchien

In der Objektorientierung wird ständig klassifiziert und in Hierarchien eingeordnet. Über einen Tiger könnte man zum Beispiel sagen, daß er eine Raubkatze ist. Eine Raubkatze ist ein Wirbeltier und ein Fleischfresser. Ein Wirbeltier ist ein Tier und ein Tier ist ein Lebewesen. Ein Tiger ist also eine spezielle Raubkatze. Eine Raubkatze ist ein spezielles Wirbeltier und ein Wirbeltier ist ein spezielles Lebewesen - und irgendwie ist auch ein Tiger ein spezielles Lebewesen[39].

Wenn wir eine Klasse (einen „Bauplan“) für ein Wirbeltier zusammenstellen sollen, fragen wir zunächst, welche Eigenschaften (Attribute[40]) ein Wirbeltier hat, die wir benötigen. Es interessieren uns wirklich **nur** die unbedingt benötigten Eigenschaften. Eigenschaften, die wir für unseren Verwendungszweck nicht benötigen, stellen nur unsinnigen Ballast dar und werden weggelassen - selbst, wenn die Beschreibung eines Wirbeltiers dadurch vollständiger werden würde.

Wenn wir nun eine Wirbeltier-Klasse haben und sollen als nächste Aufgabe eine Klasse für eine Raubkatze zusammenstellen, bietet es sich an, die schon vorhandene Wirbeltier-Klasse in die neue Raubkatzen-Klasse einzubetten, da es sich ja bei einer Raubkatze um ein spezielles Wirbeltier handelt (d.h., eine Raubkatze ein Wirbeltier ist und folglich alle Eigenschaften eines Wirbeltiers besitzt). Diesen Vorgang der „Einbettung“ bezeichnet man als Vererbung[41]. Die Raubkatze erbt alle Attribute des Wirbeltiers und bringt zusätzlich neue Attribute mit, die nur eine Raubkatze hat.

3.2 Objektorientierung und C++

Nun gilt es, dieses allgemeine *Raubkatzen-Fernseher-Blabla* in einen formalen C++ - Code umzusetzen. Diese Abstrahierung ist ein kreativer Vorgang. Es kommt nicht darauf an, eine Klasse möglichst realistisch aufzubauen. Entscheidend für den Aufbau einer Klasse ist nur, was man von der Klasse erwartet und überflüssige Details sollten weggelassen werden. Man sollte möglichst „schöne Schnittstellen“ zur Kommunikation mit der Klasse bereitstellen, die die Benutzung der Objekte, die man von der Klasse erzeugt, vereinfacht. Schnittstellen sollten allerdings auch knapp gehalten werden.

Man sollte sich beim Entwerfen von Klassen an das *Prinzip Geheimhaltung* halten. Die Schnittstelle der Klasse stellt den öffentlichen Teil dar. Alles, was nicht zur Schnittstelle gehört, sollte als privat gekennzeichnet werden. Auf diese Privatsachen kann von außen niemand zugreifen. Wenn Variablen innerhalb einer Klasse definiert werden, nennt man diese Attribute. Der Zugriff von außen auf diese Attribute sollte strikt verboten werden, d.h. Attribute sollten als privat gekennzeichnet werden. Eine Wertänderung dieser Attribute darf nur über die Schnittstelle erfolgen. Die Schnittstelle setzt sich aus Funktionen zusammen, die in der Klasse definiert wurden. Da diese speziellen Funktionen sich im-

[39] Stichwort „Polymorphie“
[40] engl. *attributes*
[41] engl. *inheritance*

mer nur auf die Klasse bzw. das Objekt beziehen, nennt man sie Methoden. Man greift also nicht direkt auf die Attribute eines Objektes zu, sondern teilt dem Objekt über die Methoden seiner Schnittstelle mit, daß es sich verändern soll. Damit gesteht man einem Objekt also ein gewisses Eigenleben zu.

Neben den öffentlichen und privaten Teilen einer Klasse gibt es noch die „geschützten" Teile. Geschützte Teile sind eigentlich privat, erlauben jedoch den Zugriff durch die „Verwandtschaft", d.h. Klassen der selben Vererbungshierarchie.

Klassen, die in einer Vererbungshierarchie weit „oben" stehen, sind naturgemäß weniger spezialisiert als die Klassen, die weiter „unten" stehen. Oft macht es keinen Sinn, ein Objekt einer wenig spezialisierten Klasse zu erzeugen, weil eben die Eigenschaften fehlen, die man benutzen will. Diese Klassen sollte man als „virtuell" kennzeichnen und damit signalisieren, daß sie sich nur zur Vererbung eignen, nicht aber zur Erzeugung von Objekten.

3.2.1 Klassen in C++

Klassen werden in C++ und Java mit dem speziellen Schlüsselwort class erzeugt. Die Syntax für eine einfache Klasse, die von keiner anderen Klasse erbt, ist wie folgt:

```
class KlassenName
{
private:
        [private Attribute und -Methoden];
protected:
        [geschuetzte Attribute und -Methoden];
public:
        [die Schnittstelle: oeffentliche Methoden (u. Attribute)];
public:
        // die spezielle Constructor-Methode ohne Rueckgabewert:
        KlassenName([Argumentenliste])
        {
                [...Anweisungen...]
        }

        // die spezielle Destructor-Methode ohne
        // Rueckgabewert und Argumentenliste
        ~KlassenName()
        {
                [...Anweisungen...]
        }
};
```

Zunächst fallen die bisher verschwiegenen „Constructor"- und „Destructor"-Methoden auf. Diese beiden Methoden haben keinen Rückgabewert. Constructor und Destructor

tragen den Namen der Klasse (hier: KlassenName). Um den Destructor vom Constructor zu unterscheiden, wird der Methodenname des Destructors von einer Tilde („∼“) angeführt. Der Constructor wird automatisch bei Erzeugung eines neuen Objektes der Klasse aufgerufen und dient der Initialisierung der Attribute. Der Destructor dient der Deinitialisierung des Objekts und wird automatisch aufgerufen, wenn das Objekt von außen zerstört wird (also spätestens bei Beendigung des Programms). Constructor und Destructor lassen sich nicht direkt aufrufen.

Da Klassendefinitionen typischerweise in Headerdateien stehen, werden die Methoden normalerweise nicht vollständig implementiert, sondern durch Prototypen ersetzt. Die Implenentierung findet in der .cc-Datei statt. Das folgende Beispiel eines einfachen Klasse soll Klarheit schaffen. Zunächst die Header-Datei *Simple.h*:

```
#ifndef SIMPLE_H // der "Include-Waechter"
#define SIMPLE_H

class Simple
{
private:
        int privater_int;
public:
        // zwei oeffentliche Methoden
        void setzeWert(int wert);
        int leseWert();

        // Prototyp des Contructors und eines
        // alternativen Contructors ohne Arg.
        Simple(int privater_int);
        Simple();

        // Prototyp des Destructors
        ~Simple();
};

#endif // SIMPLE_H ... vom "Include-Waechter"
```

Die Headerdatei beinhaltet die vollständige Definition der Klasse *Simple*. Man sieht hier, daß die Klasse zwei Constructor-Methoden besitzt, die sich nur in den Argumenten unterscheiden. Diese Art der Definition ist vollkommen legal. In Abhängigkeit der Argumente, die bei der Objekterzeugung übergeben werden wird der passende Constructor ausgewählt und ausgeführt. Diese Vorgehensweise wird „Überladen“ genannt. Es lassen sich nicht nur Constructors überladen, sondern auch Methoden und normale Funktionen (später dazu mehr). Wird in der Klassendefinition kein Constructor angegeben, so besitzt die Klasse einen leeren, argumentenlosen *default*-Constructor. Was weiterhin auffällt ist, daß das Argument des ersten Constructors den gleichen Namen trägt, wie die private int-Variable. Führt das zu Namenskonflikten? - wir werden sehen. Zunächst zu *Simple.cc*, d.h. zur Implementierung der Methoden:

```
/*
 * Einbinden der Klassendefinition
```

```
 */
#include "Simple.h"
/*
 * Weil im Destructor mit der printf-Funktion ein alberner Hilferuf
 * losgelassen wird, muss die C++-Version der Headerdatei stdio.h ein-
 * gebunden werden.
 */
#include <cstdio>

void Simple::setzeWert(int wert)
{
        this->privater_int = wert;
        // alternativ:
        // privater_int = wert;
}

int Simple::leseWert()
{
        return this->privater_int;
        // alternativ:
        // return privater_int;
}

// Der Constructor ...
Simple::Simple(int privater_int)
{
        this->privater_int = privater_int;
}

// Der alternative Constructor ...
Simple::Simple()
{
        this->privater_int = 0;
        // alternativ:
        // privater_int = 0;
}

// Der Destructor ...
Simple::~Simple()
{
        printf("Neiiiiiiinnnn Hilfe !!!\n");
}
```

Wie man sieht, wurde dem Compiler mit Simple:: klar gemacht, daß es sich bei den Funktionen in der .cc-Datei um Methoden der Klasse Simple handelt. Vor den beiden Doppelpunkten steht der Klassenname. Gesetzt den Fall, wir hätten in der Klasse *Simple* eine Methode printf definiert und wollten aus dieser oder einer anderen Methode mit der stdio.h-Funktion printf etwas auf den Bildschirm schreiben, müßten wir dem Compiler

irgendwie klarmachen, welche Funktion wir meinen - Methode oder stdio.h-Funktion. Wenn wir die stdio.h-Funktion meinen, müßten wir ihr, da sie sich in keiner Klasse befindet, einfach nur die zwei Doppelpunkte voranstellen. Neu und vielleicht auch etwas ungewöhnlich ist das Schlüsselwort this. this ist in diesem Fall vom Typ Simple* - also ein Pointer auf ein Objekt der Klasse Simple. Hierbei handelt es sich aber nicht um *irgendein* Simple-Objekt, sondern immer um das aktuelle Objekt. this-⟩privater_int bezieht sich also auf die Variable privater_int des aktuellen Objektes. Die Syntax ist die gleiche, wie bei der Dereferenzierung von struct-Pointern. Die Benutzung des this-Pointers ist optional (verbessert vielleicht die Lesbarkeit) - mit einer Ausnahme: Der Constructor mit dem unglücklich gewählten int-Argument. Da die Argument-Variable und die private Klassen-Variable den gleichen Namen tragen, muß dem Compiler klar gemacht werden, auf welche der beiden man sich bezieht. Die Schreibweise this-⟩privater_int = privater_int ist eindeutig. Die Argument-Variable wird der privaten Klassenvariable zugewiesen.

3.2.2 Der Copy-Constructor

Jede Klasse besitzt, ohne daß ihn der Programmierer explizit angibt, einen *Copy - Constructor*. Mit Hilfe dieses speziellen, impliziten Constructors lassen sich Kopien eines Objekts anlegen - genauer gesagt wird eine Kopie eines Objekts erzeugt, indem ein neues Objekt mit den Elementen eines bestehenden Objekts initialisiert wird. Gesetzt den Fall, wir haben ein Objekt mit Namen simpleobj der Klasse Simple:

```
Simple simpleobj();
```

Jetzt soll eine 1:1-Kopie mit Namen simpleobj_kopie erzeugt werden:

```
Simple simpleobj_kopie(simpleobj);
```

Hier sieht man direkt, wie das Objekt simpleobj dem Copy - Constructor des neuen Objekts simpleobj_kopie übergeben wird. Folgende Schreibweise ist äquivalent und beschreibt die Kopier-Aktion intuitiv vielleicht besser:

```
Simple simpleobj_kopie = simpleobj;
```

Der Copy - Constructor von Simple ist wiefolgt definiert und kann, wenn das Standard-Verhalten (Kopieren aller Elemente) nicht gewünscht ist, vom Programmierer selbst implementiert werden:

```
Simple::Simple(const Simple&);
```

Das zu kopiende Objekt wird dem neuen Objekt als Referenz (siehe dazu auch *call-by-reference*) übergeben. Das Schlüsselwort const deutet hier an, daß das *Quell-Objekt* als konstant anzusehen ist und nicht verändert werden darf. Die Zuweisung im zweiten, äquivalenten Beispiel wurde über den implizit definierten Zuweisungsoperator operator= ermöglicht. Über das Definieren von Operatoren gibt es einen gesonderten Abschnitt (siehe dazu *Überladen von Operatoren*).

3.2.3 Der Default-Constructor

Ein weiterer impliziter Constructor ist der *Default - Constructor*. Dieser Constructor besitzt keine Parameter und existiert nur für solche Klassen, die keinen eigenen, explizit angegebenen Constructor haben - d.h. sobald man einen Constructor für eine Klasse programmiert hat, darf der *Default - Constructor* nicht mehr verwendet werden.

3.2.4 Vererbung in C++

Das folgende Beispiel zeigt eine Klasse Tiger, die eine Klasse Raubkatze *beerbt*. Durch diesen Vorgang der Vererbung wird die Klasse Tiger selbst zu einer Raubkatze:

```
#include <cstdio>

class Raubkatze
{
private:
        int alter;
public:
        int schnurrhaare;
        int beine;
        bool hungrig;
public:
        void setzeAlter(int a)
        {
                alter = a;
        }
        int leseAlter()
        {
                return alter;
        }
public:
        /*
         * Die constructor-Methode
         */
        Raubkatze(int sh)
        {
                schnurrhaare = sh;
                printf("Ich bin eine Raubkatze und habe ");
                printf("%i Schnurrhaare.\n", schnurrhaare);
        }
        /*
         * Die destructor-Methode
         */
        ~Raubkatze()
        {
                printf("Die Raubkatze verabschiedet sich ...\n");
        }
};
```

```
class Tiger : public Raubkatze
{
public:
        bool tigerpfote;
        bool tigerfell;
public:
        void anschleichen()
        {
                // ...
        }
public:
        /*
         * Die constructor-Methode:
         *      Diese Tieger haben alle 76 Schnurrhaare und 4 Beine:
         *
         *      Raubkatze(76)
         *              ruft den constructor der Raubkatzen-Klasse
         *              auf, wo schnurrhaare mit dem Wert 76
         *              initialisiert wird.
         *      tigerpfote(true)
         *              initialisiert das Tiger-Attribut tigerpfote
         *              mit dem Wert true.
         */
        Tiger() : Raubkatze(76), tigerpfote(true)
        {
                tigerfell = true;

                // Ein Tiger ist staendig hungrig.
                // hungrig war urspruenglich ein Attribut
                // der Klasse Raubkatze:
                hungrig = true;
                // genauso beim Raubkatzen-Attribut beine:
                beine = 4;

                setzeAlter(10);

                printf("Ich bin ein Tiger mit tigerpfote und ");
                printf("tigerfell.\nIch als Tiger habe ");
                printf("%i Schnurrhaare.\n", schnurrhaare);
                printf("Ich bin %i Jahre alt.\n", leseAlter());
        }

        ~Tiger()
        {
                printf("Der Tiger verabschiedet sich ...\n");
        }
};
```

```
int main()
{
        Tiger *tiger_objekt;

        // einen neuen Tiger dynamisch erzeugen...
        tiger_objekt = new Tiger();

        // den neuen Tiger gleich wieder loeschen:
        delete tiger_objekt;
}

/*
 * DIE AUSGABE DES PROGRAMMS:
 *      Ich bin eine Raubkatze und habe 76 Schnurrhaare.
 *      Ich bin ein Tiger mit tigerpfote und tigerfell.
 *      Ich als Tiger habe 76 Schnurrhaare.
 *      Ich bin 10 Jahre alt.
 *      Der Tiger verabschiedet sich ...
 *      Die Raubkatze verabschiedet sich...
 */
```

Wie die Vererbung mit der Klasse Raubkatze funktioniert, ist hoffentlich offensichtlich. Folgende Punkte sollten auffallen:

- Mit class Tiger : public Raubkatze wird angedeutet, daß die Klasse Tiger alle öffentlichen Attribute und Methoden von der Klasse Raubkatze übernimmt.
- Innerhalb der Tiger-Klasse kann nicht auf das als privat gekennzeichnete Attribut alter der Raubkatze zugegriffen werden - was allerdings nicht heißt, daß Tiger das Attribut alter nicht hat - ein Zugriff über die Raubkatzen-Methode *setzeAlter / leseAlter* funktioniert. Das Gleiche gilt für privat gekennzeichnete Methoden.
- Die Klasse Tiger erweitert die Raubkatze um die beiden Attribute tigerfell und tigerpfote, sowie die Methode anschleichen().
- Da es sich um eine *public-Vererbung* handelt, kann in Tiger direkt auf die öffentlichen Attribute von Raubkatze zugegriffen werden (siehe schnurrhaare, beine und hungrig).
- Der constructor von Raubkatze verlangt einen int-Wert zur Initialisierung. Die *Raubkatze im Tiger* wird in Tigers constructor-Methode mit Tiger() : Raubkatze(76) initialisiert.
- Tigers constructor zeigt eine alternative Möglichkeit um Klassenattribute zu initialisieren: Tiger() : tigerpfote(true). tigerpfote wird der Wert true zugewiesen. Mehrere Attribut-Initialisierungen werden mit Kommata abgetrennt.
- Die angehängte Ausgabe des Programms zeigt, in welcher Reihenfolge die constructors / destructors aufgerufen werden.

- C++ bietet im Gegensatz zu Java die Möglichkeit zur Mehrfachvererbung. Bei dieser (hier nicht gezeigten) Art der Vererbung kann eine Klasse von mehreren anderen Klassen erben. Die Klassennamen werden hierbei einfach durch Kommata abgetrennt an das class-Schlüsselwort angehängt: class Tiger : public Raubkatze, public Ente. Die klasseninterne Handhabung gleicht der der Einfachvererbung.

3.2.5 Abstrakte Klassen und virtuelle Vererbung

Einen Tiger oder einen Löwen kann man als Raubkatze bezeichnen. Es macht Sinn, eine Klasse Tiger anzulegen, die die Eigenschaften der Klasse Raubkatze erbt. Gleiches gilt für eine Klasse Loewe. Macht es aber Sinn ein Objekt der Klasse Raubkatze anzulegen? Raubkatze beschreibt die Gemeinsamkeiten aller raubkatzenartigen Tiere - es fehlen jedoch Raubkatze wichtige Eigenschaften, damit es Sinn machen würde Objekte davon anzulegen (z.B. *anschleichen*, um Beute zu schlagen). Diese wichtigen Eigenschaften werden erst durch die Vererbung eingebaut - nur bekommt das *Ding* dann einen anderen Namen: Tiger.

Es macht also keinen Sinn direkt von Raubkatze Objekte anzulegen, weil Raubkatze selbst zu *abstrakt* formuliert ist. Raubketze ist eine Basisklasse, die nur im Zusammenhang der Vererbung ihre Daseinsberechtigung hat. Wenn die Objekterzeugung aber keinen Sinn macht, sollte sie ausdrücklich verboten werden. Jemand, der versucht ein Objekt der Klasse Raubkatze zu erzeugen zeigt damit, daß er die grundlegenden Gedanken des Programmierers von Raubkatze nicht verstanden hat! Der Compiler sollte darauf hinweisen.

Es ist klar, daß jede Raubkatze irgendwie jagen muß, um zu überleben. Die verschiedenen Arten unterscheiden sich aber in der Art, wie sie jagen. Die *Anlage* zur Jagd sollte in jeder Raubkatze vorhanden sein. Um diese Anlage zu verankern erhält die Klasse Raubkatze eine *virtuelle* Methode jagen:

```
class Raubkatze
{
private:
        int alter;
public:
        int schnurrhaare;
        int beine;
        bool hungrig;
public:
        void setzeAlter(int a)
        {
                alter = a;
        }
        int leseAlter()
        {
                return alter;
        }
```

```
        /*
         * Die neue virtuelle Methode jagen()
         */
        virtual bool jagen(Beute *b) =0;
public:
        /*
         * Die constructor-Methode
         */
        Raubkatze(int sh)
        {
                schnurrhaare = sh;
                printf("Ich bin eine Raubkatze und habe ");
                printf("%i Schnurrhaare.\n", schnurrhaare);
        }
        /*
         * Die destructor-Methode
         */
        ~Raubkatze()
        {
                printf("Die Raubkatze verabschiedet sich ...\n");
        }
};
```

Die neue virtuelle Methode hat keinen Methodenrumpf - was ja auch keinen Sinn machen würde, da jede Raubkatze andere Jagdtechniken einsetzt. Wichtig ist das Schlüsselwort virtual, mit dem die Methode als virtuell gekennzeichnet wird. Wichtig ist auch das „=0“, mit dem die Methode als *rein virtuell* gekennzeichnet wird, was den Compiler mitteilt, daß diese Methode keinen Methodenrumpf hat, sondern nur einen Teil der Schnittstelle spezifiziert. Der Rückgabewert der Methode ist bool, was andeuten soll, daß die Jagd entweder erfolgreich war, oder nicht. Gejagt wird natürlich eine Beute (ein Beute-Objekt einer irgendwo definierten Beute-Klasse) - der einzige Parameter der Methode.

Da die Klasse eine rein virtuelle Methode enthält, wird sie selbst zu einer *rein virtuellen* oder *abstrakten* Klasse, von der sich keine Objekte mehr anlegen lassen. Jede Klasse, die unsere neue Version der Klasse Raubkatze beerbt, wird dazu gezwungen, die Methode jagen zu implementieren. Wegen unserer Änderung in Raubkatze müssen wir nun auch den Tiger ändern und eine (der in der *abstrakten Basisklasse* Raubkatze spezifizierten Schnittstelle entsprechende) Methode jagen implementieren:

```
// Die Klassendefinition von Raubkatze einbinden
#include <Raubkatze.h>

class Tiger : public Raubkatze
{
public:
        bool tigerpfote;
        bool tigerfell;
public:
```

```
        void anschleichen()
        {
                // ...
        }

        /*
         * WICHTIG: Die Implementierung von jagen. Die Schnittstelle
         * muss der in Raubkatze spezifizierten Schnittstelle entsprechen
         */
        bool jagen(Beute *b)
        {
                bool jagd_erfolgreich = false;

                anschleichen();
                // ...
                if (jagd_erfolgreich)
                {
                        hungrig = false;
                        return true; // Jagd war erfolgreich
                }
                else
                {
                        hungrig = true;
                        return false; // Jagd war erfolglos
                }
        }
public:
        Tiger() : Raubkatze(76)
        {
                // ...
        }

        ~Tiger()
        {
                // ...
        }
};
```

An diesem Punkt ahnt man schon, daß es einen Unterschied zwischen *virtuell* und *rein virtuell* gibt: Wird das „=0“ weggelassen, ist die Methode *virtuell* und benötigt einen Methodenrumpf. Sobald die Klasse Raubkatze keine rein virtuellen Methoden mehr enthält, können von ihr wieder Objekte erzeugt werden. Damit handelt es sich nicht mehr um eine abstrakte Klasse. In einem Methodenrumpf einer virtuellen Methode könnten irgendwelche Standard-Operationen erledigt werden, die abgearbeitet werden sollen, **falls** eine beerbende Klasse - z.B. Tiger - die virtuellen Methoden nicht mit eigenen Methoden *überschreibt* (wie im obigen Quelltext der neuen Tiger-Klasse geschehen).

Man sollte sich genau überlegen, ob es Sinn macht eine Klasse mit derart virtuellen

Methoden zu haben, von der sich Objekte anlegen lassen. Bei dem Raubkatzen-Beispiel macht es sicher keinen Sinn, da die Raubkatze von Anfang an als abstrakte Klasse (mit rein virtuellen Methoden) geplant war.

3.2.6 Polymorphie

Im vorangegangenen Beispiel hat eine Raubkatze eine Beute gejagt. Bei einer solchen Beute kann es sich um eine konkrete, alleinstehende Klasse Beute gehandelt haben, oder aber um eine Vererbungshierarchie wie wir sie von der Klasse Raubkatze kennen: Beute ist möglicherweise eine (möglicherweise) abstrakte Basisklasse für eine Klasse Zebra, eine Klasse Gnu oder sogar eine „angehängte“ Klassenhierarchie Antilope[42]. Zum Erlegen einer Beute kann ein Tiger auf alle Eigenschaften/Methoden zurückgreifen, die ihm die (möglicherweise abstrakte) Klasse Beute bietet. Den Umstand, daß man eine abgeleitete Klasse wie eine Basisklasse benutzen kann, bezeichnet man als *Polymorphie*[43]. Ein Beispiel:

```
/*
 * Klassendefinition der abstrakten Basisklasse Raubkatze
 */
class Raubkatze
{
        // ...
        virtual bool jagen(Beute *b) =0;
        // ...
};

/*
 * Klassendefinition der von Raubkatze abgeleiteten Klasse Tiger
 */
class Tiger : public Raubkatze
{
        // ...
        /*
         * jagen wird vom Tiger implementiert
         */
        bool jagen(Beute *b)
        {
                // ...
        }
        // ...
}

/*
 * Klassendefinition der (moeglicherweise abstrakten) Basisklasse
 * Beute
 */
```

[42] es gibt ja verschiedene Antilopenarten

[43] Polymorphie auf deutsch: *Vielgestaltigkeit*

```
class Beute
{
        // ...
public:
        bool konnte_entkommen()
        {
                // ...
        }
        bool wurde_gefressen()
        {
                // wenn die Beute nicht entkommen konnte
                // wurde sie gefressen
                return !konnte_entkommen();
        }
        // ...
};

/*
 * Klassendefinition einer Klasse fuer pferdeartige Tiere
 */
class PferdeTier
{
        // ...
public:
        void die_flucht_ergreifen()
        {
                // ...
        }
        // ...
};

/*
 * Klassendefinition der abgeleiteten Klasse Zebra. Hier handelt es
 * sich wieder um Mehrfachvererbung:
 *   - Zebra erbt die Beute-Eigenschaft von der Basisklasse Beute
 *   - Zebra erbt die PferdeTier-Eigenschaft von der Basisklasse
 *     PferdeTier
 */
class Zebra : public PferdeTier, public Beute
{
        // ...
};

/*
 * Ein Hauptprogramm mit polymorphen Aufrufen
 */

#include <cstdio>
```

```
int main()
{
        Tiger *theobald = new Tiger();
        Zebra *mathilde = new Zebra();

        // Der Tiger theobald jagt
        if (theobald->jagen(mathilde))
        {
                printf("der Tiger theobald hat das Zebra");
                printf(" mathilde erlegt.\n");
        }
        else
        {
                printf("das Zebra mathilde ist dem Tiger");
                printf(" theobald entkommen.\n");
        }

        // die Methode wurde_gefressen() hat das Zebra von
        // der Basisklasse Beute geerbt.
        if (mathilde->wurde_gefressen())
        {
                // mathilde ist tot und wird geloescht.
                delete mathilde;
        }
        else
        {
                printf("theobald ist jetzt hungrig.\n");
                printf("mathilde ergreift jetzt die Flucht\n");
                mathilde->die_flucht_ergreifen();
        }

        // ...

        return 0;
}
```

3.2.7 Friends

In seltenen Fällen kann es beim Entwurf einer Klasse durchaus sinnvoll sein, einer Funktion oder einer anderen Klasse Zugriff auf die privaten Teile (also private Methoden oder private Attribute) zu erlauben. Wenn dies gewünscht wird, muß in der Klassendefinition vermerkt werden, daß es sich bei der anderen Klasse oder der Funktion um einen *friend*[44] handelt. Dem *Freund* wird dann der Zugriff auf die privaten Teile der Klasse gestattet - er hat also die gleichen Zugriffsrechte wie eine echte Methode der Klasse.

[44] engl. *Freund*

Wichtig ist, daß sich ein *friend* nicht vererben läßt. Eine Klasse, die von einer anderen Klasse mit *friend* abgeleitet wird, erbt **nicht** den *friend*! Weiterhin ist Eigenschaft einen *friend* zu haben nicht transitiv - das bedeutet, daß die *friend*s meiner *friend*-Klasse nicht automatisch meine *friend*s sind[45]. Beispiel:

```
class Simple
{
        // die Funktion useSimpleObj ist ein friend:
        friend void useSimpleObj(Simple &sobj);
        friend class SimplesFriend;
private:
        void privateMethode()
        {
                // sonstwas
        }

        /* ... */
};

void useSimpleObj(Simple &sobj)
{
        // useSimpleObj ist friend von Simple
        // und darf die private Methode "privateMethode"
        // aufrufen:
        sobj.privateMethode();
}

class SimplesFriend
{
        /* ... */
public:
        void nutzeSimple(Simple &sobj)
        {
                // SimplesFriend::nutzeSimple darf die
                // die private Methode "privateMethode"
                // aufrufen, weil die Klasse SimplesFriend
                // ein Freund von Simple ist.
                sobj.privateMethode();
        }

        /* ... */
};
```

[45] d.h. die Freunde meiner Freunde nicht nicht automatisch meine Freunde

3.2.8 Namespaces

Namespaces[46] stellen eine Möglichkeit zur Zusammenfassung / Gruppierung von Klassen, Funktionen und Variablen dar. Der Einsatz von Namespaces ist in den folgenden Fällen zu empfehlen:

- Mehrere Klassen beschäftigen sich mit dem selben Themenkomplex - z.B. mit graphischen Benutzerschnittstellen oder Datenbankzugriffen. Man faßt diese Klassen in einem Namespace zusammen, um ihre thematische Verwandtschaft zu unterstreichen.
- Bei der Programmierung eines größeren Programms lassen sich spontan wichtige Programmteile oder Komponenten unterscheiden. Jede dieser Komponenten erhält einen eigenen Namensbereich.
- Nicht für jedes Problem gibt es eine saubere, objektorientierte Lösung in Form von Klassen und Objekten. Manchmal kann es besser sein, auf Klassen zu verzichten[47]. In diesem Fall ist es besser, Funktionen und Variablen eines Themenkomplexes der sich gegen Objektorientierung *wehrt*, in einem Namespace zu verkapseln.

Wird auf eine in einem Namespace verkapselte Funktion, Variable oder Klasse zugegriffen, muß man dem Compiler irgendwie mitteilen, daß man sich auf die Funktion, Variable oder Klasse beziehnt, die im Namespace deklariert wurde. Dazu gibt es zwei Möglichkeiten. Im den folgenden Beispielen greift eine main-Funktion auf die im Namespace *Finanz* deklarierte Funktion *berechneZinsesZins(float, int, float)* zu. Die erste Möglichkeit benutzt die using-Direktive:

```
using namespace Finanz;

int main()
{
        // ...
        double zs = berechneZinsesZins(2000.0, 20, 5.2);
        // ...
}
```

Durch die using-Direktive wird der Compiler veranlaßt, den angegebenen Namespace *Finanz* nach einer Funktion *berechneZinsesZins* abzusuchen. Es geht aber auch anders, indem man dem Compiler direkt mitteilt, daß man die Funktion *berechneZinsesZins* aus dem Namespace *Finanz* meint:

```
int main()
{
        // ...
        zs = Finanz::berechneZinsesZins(2000.0, 20, 5.2);
        // ...
}
```

[46] engl. *Namensbereiche*
[47] das ist zumindest meine ganz persönliche Meinung

Unterschlagen wurde bislang, wie man Funktionen, Klassen und Variablen in Namespaces einträgt. Für die Header-Datei schlage ich folgende Schreibweise vor:

```
namespace Finanz
{

// die Klassendefinition der Klasse Finanz::Kredit
class Kredit { /* ...die Klassendefinition... */ };

// der Funktionsprototyp von Finanz::berechneZinsesZins
double berechneZinsesZins(float, int, float);

// die extern-Deklaration der Variable zinssatz:
extern float zinssatz;

}; // Ende des Namespace Finanz

// der bisherige Namespace Finanz wird nun um eine
// Klassendefinition erweitert ...
namespace Finanz
{

// den Namespace Finanz um die Klasse Steuererklaerung erweitern
class Steuererklaerung { /* ...die Klassendefinition... */ };

}; // Ende des Namespace Finanz
```

Wie man sieht, läßt sich ein einmal geschlossener Namespace wieder öffnen und um weitere Einträge erweitern (im Beispiel die Klassendefinition der Klasse *Steuererklaerung*). In der .cc-Datei könnte dann stehen ...

```
namespace Finanz
{

// eine Methode der Klasse Finanz::Kredit ...
bool Kredit::zurueckZahlen()
{
        // ... irgendwas ...
}

// [...weitere Methoden der Klasse Finanz::Kredit...]

// die Implementierung von Finanz::berechneZinsesZins
double berechneZinsesZins(float a, int b, float c)
{
        // ... irgendwas ...
```

```
}

// die Variable zinssatz
float zinssatz;

// eine Methode der Klasse Finanz::Steuererklaerung
int Steuererklaerung::drucken()
{
       // ... irgendwas ...
}

// [...weitere Methoden der Klasse Finanz::Steuererklaerung...]

}; // Ende des Namespace Finanz
```

3.2.9 Statische Attribute und Methoden

Methoden (in Klassen deklarierte Funktionen) und Attribute (in Klassen deklarierte Variablen) können mit dem Schlüsselwort static versehen werden.

Wird eine Variable mit static versehen, wird damit eine globale Variable erzeugt, die allen Objekten der Klasse gehört. Auch, wenn mehrere Objekte der Klasse erzeugt werden, gibt es nur eine einzige Ausgabe dieser static-Variable. Da alle Objekte einer Klasse sich dieselbe static-Variable *teilen*, stellt sich die Frage, welchem der Objekte die Ehre zukommt sie zu erzeugen. Die Antwort ist: Keines der Objekte erzeugt die static-Variable. Sie muß vom Programmierer außerhalb der Klasse erzeugt werden. Wird das Erzeugen der Variable vergessen, meldet der Linker beim Binden des Programms eine *undefined reference*. Beispiel:

```
class Person
{
        ...
private:
        ...
        static Kraftwagen MammasPolo("rot", "45 PS", "");  // FALSCH!
        static int a = 5;  // FALSCH!
        ...
        static Kraftwagen PappasBenz;  // richtig! :-)
        static int b;  // richtig :-)
        ...
};

...
// Erzeugung der static-Variablen:
Kraftwagen Person::PappasBenz("blau", "200 PS", "Anhaengerkupplung");
```

```
// Person::b erzeugen und mit 5 vorbelegen
int Person::b = 5;
...

int main()
{
        ...
        Person Erich();  //  Erich, August und Willi
        Person August(); //  teilen sich
        Person Willi();  //  PappasBenz
        ...
}
```

Wird eine Methode mit static versehen, ist das Verhalten ähnlich: Die Methode - die in diesem Fall auch *statische Elementfunktion* genannt wird - ist auch dann aufrufbar, wenn (noch) kein Objekt der Klasse erzeugt wurde. Bei einem Aufruf ohne ein existierendes Objekt muß die Methode durch den Klassennamen (hier *KlassenName*) qualifiziert werden:

```
int i = KlassenName::dieStaticMethode(die, argumenten, liste);
```

Das Schlüsselwort static steht nur in der Klassendefinition. Wird die Methode in der .cc-Datei der Klasse implementiert, darf das static vor dem Rückgabewert der Methode nicht mehr mit angegeben werden.

static vor klassenlosen Funktionen hat die gleiche Wirkung wie in C: Die static-Funktionen sind nur innerhalb des Code-Moduls sichtbar.

3.2.10 Konstante Methoden

Es ist sicher nicht wünschenswert, daß eine static-Methode die Attribute einer Klasse verändert, da sich ja bei einem Aufruf über den Klassennamen auf kein konkretes Objekt bezogen wird[48]. Um dies zu garantieren, sollte man hinter der Argrumentenliste der Methode das Schlüsselwort const angeben. Durch dieses Schlüsselwort wird dem Compiler mitgeteilt, daß es sich um eine konstante Methode handelt, die die Attribute der Klasse nicht verändern darf.

Es ist guter Stil, wenn man allen Methoden, die das Objekt nicht verändern (und z.B. nur irgendwelche Attribut-Werte nach außen durchreichen) das Schlüsselwort const nachstellt.

3.2.11 Ausnahmebehandlung

Fehler können immer und überall auftreten. Es ist äußerst schwierig fehlerfreie Programme zu schreiben. Jede Nebensache, an die der Programmierer nicht gedacht hat (oder

[48] es sei denn, die Attribute sind selbst static

nicht denken wollte), kann sich irgendwann rächen und im schlimmsten Fall zu einem Programmabsturz führen.

C++ und Java unterstützen mit ihrem *exception handling*[49] eine saubere Art mit Fehlern umzugehen. Ziel ist es, einen aufgetretenen Fehler zu erkennen und zu behandeln. Nicht jeder kleine Fehler muß unbedingt einen Abbruch des Programms zur Folge haben. Viele Fehler lassen sich, sobald sie erkannt sind, korrigieren. Bevor wertvolle Daten infolge eines Programmabsturzes verschwinden, wird im Zweifelsfall der Benutzer um Rat gefragt.

Der Kern des *exception handling* wird von den Schlüsselwörtern try[50], throw[51] und catch[52] gebildet. Wird irgendwo in einem Teil des Programms ein Fehler erkannt, wird an dieser Stelle eine Exception mit einer Nachricht über die Art des Fehlers „geworfen“, wodurch die gerade laufende Funktion abgebrochen wird. Wird in der Funktion, die die fehlerverursachende Funktion aufgerufen hatte, die „geworfene Exception“ nicht mit einem catch-Block „aufgefangen“, wird auch diese aufrufende Funktion beendet und die Exception wird „weitergeworfen“. Dieser Vorgang setzt sich bis in die main-Funtion fort, wenn nicht irgendwo ein catch-Block die Exception auffängt und behandelt, wodurch das Programm fortgesetzt werden kann (oder auch nicht, falls der Fehler nicht korrigierbar ist). Fehlt allerdings dieser catch-Block auch in der main-Funktion, wird das komplette Programm in letzter Konsequenz abgebrochen - für den Programmier sicher ein Grund sich zu schämen.

Was wirft man als Exception? C++ erlaubt es ein beliebiges Objekt und alle eingebauten Typen zu werfen. Sinnvollerweise programmiert man sich eine spezielle Exception-Klasse, die alle noetigen Informationen aufnehmen kann (etwa Funktions-/Klassenname, Quellcode-Datei, Zeilennummer, Art des Fehlers). Eine einfache Exception-Klasse könnte so aussehen:

```
...

class MyException
{
private:
        char *error_msg;
public:
        MyException(char *msg)
        {
                error_msg = new char[strlen(msg)+1];
                strcpy(error_msg, msg);
        }

        ~MyException() { delete error_msg; }
```

[49] engl. *Ausnahmebehandlung*
[50] engl. *probieren, versuchen*
[51] engl. *werfen*
[52] engl. *fangen*

```
        const char *errstr()
        {
                return error_msg;
        }

        // ... weitere Fehlerinformationen
};

...
```

Ein Beispiel zeigt, wie man die Ausnahmebehandlung benutzt:

```
/*
 * Diese Funktion wirft als Exception ein Objekt der
 * Klasse MyException
 */
int rechneirgendwas(int a, int b, int anzahl) throw (MyException)
{
        // [...]
        /*
         * Ein Fehler ist aufgetreten, der die Fortsetzung
         * der Funktion sinnlos macht - es wird jetzt eine
         * Exception geworfen:
         */
        if (fehler)
                throw (MyException(FEHLERMELDUNG));
        // [...]
}

/*
 * Diese Funktion wirft im Falle einer Exception ein
 * Objekt der Klasse SonstwasException
 */
int sonstwas() throw (SonstwasException)
{
        // ...
        /*
         * try bedeutet: "In diesem Block koennen Exceptions auftreten
         * die ich in einem catch behandeln moechte"
         */
        try
        {
                rechneirgendwas(a,b,x);
        }
        catch (MyException fehler) // fange die Exception in fehler
        {
                /*
                 * In diesem Block wird der Fehler behandelt und die
                 * Funktion muss nicht beendet werden, weil dieser
```

```
                    * Block existiert.
                    */
                   if (fehler.behebbar())
                           behebe_fehler();

                   /*
                    * bei unbekanntem Fehler eine andere Exception
                    * weiterwerfen (eine von vielen Moeglichkeiten)
                    */
                   if (fehler.unbekannt())
                           throw (SonstwasException(...));
           }
           // ...
}
```

Wie man sieht, habe ich den Funktionskopf um throw (ExceptionKlasse) ergänzt. Diese Ergänzung ist keine Pflicht und dient nur der Übersichtlichkeit. Man ist nicht schlecht beraten, wenn man diese Ergänzung in seine eigenen Programme aufnimmt. Ansonsten kann es schnell passieren, daß eine ungefangene Exception zu einem scheinbar unerklärlichen Programmabbruch führt (ohne Hinweis oder Fehlermeldung!).

3.3 Andere Erweiterungen von C++

3.3.1 Dynamische Speicherverwaltung

C++ stellt für die dynamische Speicherverwaltung die beiden neuen Operatoren new und delete zur Verfügung. new belegt einen neuen Speicherplatz vom übergebenen Typ und gibt einen Zeiger auf das neu erstellte Objekt zurück. new ersetzt damit die unter C verwendete malloc()-Funktion:

```
TypName * objektname = new TypName;
```

Da es sich bei den Typen in C++ i.d.R. um Klassen handelt, die einen Constructor besitzen, der bei Objekterzeugung aufgerufen werden soll, kann man die Constructor-Parameter bei Objekterzeugung mit new übergeben:

```
KlassenName * objektname = new KlassenName([Liste der Constructorparam.]);
```

Implizit wurde auch im ersten Beispiel ein Constructor aufgerufen. Dabei handelte es sich jedoch um den *Default*-Constructor, den jede Klasse implizit besitzt (in diesem Fall TypeName::TypName()). Der new-Operator hat den Vorteil, daß der zurückgegebene Speicherplatz automatisch vom richtigen Typ ist - eine Typenkonvertierung, wie sie bei malloc() erforderlich war, entfällt. Ein weiterer Vorteil ist, daß sich der Programmierer nicht um die Größe des zu belegenden Speicherplatzes kümmern muß - new ermittelt automatisch die erforderliche Speicherplatzgröße für ein Objekt des übergebenen Typs[53].

[53] in C mußte diese Größe mit dem sizeof()-Operator ermittelt werden

Der delete-Operator übernimmt die Funktion von free() und gibt einen mit new belegten Speicherplatz wieder frei. Der Unterschied zu free() ist allerdings, daß bei Objekten die Destructor-Methode aufgerufen wird, bevor das eigentliche Objekt freigegeben wird.

```
delete objektname;
```

Von new und delete gibt es Varianten für die Allokation von Feldern:

```
int * int_feld = new int[256];
...
delete [] int_feld;
```

Wenn eine Speicheranforderung mittels new fehltschlägt, wird eine Exception vom Typ bad_alloc geworfen. In Systemen, wo es abzusehen ist, daß der Speicher ausgehen wird, ist es sinnvoll diese Exception zu fangen und zu behandeln, da eine unbehandelte Exception das laufende Programm abbrechen wird (vielleicht ist es guter Stil, dies grundsätzlich zu machen ... aber niemand macht es, weil jeder davon ausgeht, daß genügend Speicher für sein Programm vorhanden ist ;-).

3.3.2 default-Werte für Funktionsparameter

Relativ häufig übergibt man einer Funktion immer wieder die selben Werte. Einzelne Parameter verändern sich vielleicht nur sehr selten - trotzdem muß man sie immer wieder angeben. In C++ hat man die Möglichkeit *default*-Werte[54] für einzelne Parameter festzulegen. Werden diese Parameter beim Funktionsaufruf weggelassen, nimmt der Compiler an, daß die default-Werte für die fehlenden Parameter eingesetzt werden sollen. Default-Parameter werden in den Funktions-Prototypen bzw. in den Prototypen der Methoden in der Klassendefinition festgelegt. Die default-Parameter sind grundsätzlich die letzten Parameter der Parameterliste. Eine Parameterliste kann dabei mehrere default-Parameter enthalten. Bei einer Parameterliste mit mehreren default-Parametern dürfen die letzten default-Parameter weggelassen werden. Der Compiler muß in der Lage sein, die richtigen Parameterzuweisungen und Funktionsnamen auflösen zu können. Werden gleichzeitig Funktionen überladen, kann leicht die Eindeutigkeit verloren gehen und der Compiler wird den Übersetzungsvorgang abbrechen. Ein Beispiel für die Verwendung von default-Parametern:

```
/* BeispielKlasse.h */
class BeispielKlasse
{
        ...
        int eineMethode(int a, int b, int c=42, int d=43);
        ...
};

/* Hauptprogramm */
int main()
```

[54] auch *Standard-Werte* gennannt

```
{
        BeispielKlasse beispiel();
        beispiel.eineMethode(1,2,3,4); // a<-1, b<-2, c<- 3, d<- 4
        beispiel.eineMethode(1,2,3);   // a<-1, b<-2, c<- 3, d<-42
        beispiel.eineMethode(1,2);     // a<-1, b<-2, c<-42, c<-43
}
```

3.3.3 Überladen von Funktionen und Methoden

Wenn Funktionen[55] überladen werden, hat man zwei oder mehreren Funktionen zu tun, die den gleichen Namen tragen, sich aber in ihrer Parameterliste und/oder ihrem Rückgabewert unterscheiden. Welche der gleichnamigen Funktionen verwendet wird, entscheidet der Compiler über die beim Aufruf verwendete Variablenanzahl und Variablentypen der übergebenen Parameter.

3.3.4 Überladen von Operatoren

(nicht fertig)

3.3.5 Call By Reference

C++ führt eine neue Technik ein - den „call by reference“. Ziel dieser neuen Technik ist es, die fehleranfällige Handhabung von Pointern zu vermeiden. Eine Referenz ist ein Verweis auf ein Objekt einer Klasse oder eine Variable eines Typs, der sich wie das Objekt oder die Variable benutzen läßt. Um dem Compiler klarzumachen, daß es sich um eine Referenz handelt, hängt man dem Typ oder der Klasse ein &-Zeichen an[56]. Beispiel:

```
/*
 * Vertauscht zwei int-Variablen miteinander. Die Variablen werden
 * per Referenz uebergeben.
 */
void swap (int& a, int& b)
{
        int temp = a;
        a = b;
        b = temp;
}

int main()
{
        int  a = 10;
        int& b = a;      // b ist eine Referenz auf a
        int  c = b;      // c ist ein neuer int, dem b und damit a
                         // zugewiesen wird.
        b = 2;           // b wird 2 zugewiesen - damit wird auch a
```

[55] das folgende gilt auch für Methoden

[56] ähnlich wie das * bei den Pointern

```
                                // 2 zugewiesen, da b eine Referenz auf a ist.

        int x=1, y=2;
        swap(x, y);         // swap speichert x in y und y in x
}
```

Referenzen machen nur Sinn und sind auch nur erlaubt, wenn ihnen direkt ein Objekt zugewiesen wird (was auch im Funktionskopf sichergestellt ist). Ansonsten könnte man etwas anlegen, was sich wie ein Objekt verhält, aber weder auf ein Objekt verweist, noch ein Objekt ist. Etwas wie ...

```
        ...
        int& a;
        ...
```

... stellt also einen Fehler dar, weil die notwendige Initialisierung fehlt.

3.4 Generische Programmierung

3.4.1 Template-Funktionen

Oft hat man es in C mit dem Mißstand zu tun, daß man ein und denselben Algorithmus immer wieder implementieren muß, nur weil man ihn einen anderen Variablentyp verarbeiten lassen will. Ein Beispiel dafür ist eine Sortierfunktion, die ein int-Feld sortieren kann. Will man mit dieser Funktion ein float-Feld sortieren, muß sie umgeschrieben werden, was nicht sehr elegant ist und zusätzlichen Aufwand bedeutet. Am Algorithmus selbst hat sich nichts geändert. Diesen Mißstand kann man mit einer C++-Erweiterung, die *template*[57] genannt wird, umgehen. Ein einfaches Beispiel zeigt, wie soetwas aussehen kann:

```
template < class T > void swap (T& a, T& b)
{
        T temp = a;
        a = b;
        b = temp;
}

template < class T > T maximum (T min, T max)
{
        return (min>max) ? min : max;
}

int main()
{
        char a = 'Z';
        char z = 'A';
        char max_char;

        int u = 102;
        int v = 42;
```

[57] *template* heißt soviel wie „Vorlage“

```
    int max_int;

    int max_sonstwas;
    char max_kaputt;

    swap (a, z);      // 'A' in a speichern und 'Z' in z speichern
    swap (u, v);      // 102 in v speichern und 42 in u speichern

    max_char = maximum (a, z);        // ok
    max_int = maximum (u, v);         // auch ok
    max_sonstwas = maximum (a, z);    // auch noch ok
    max_kaputt = maximum (u, v);      // funktioniert, aber ein
                                      // int wird in einem char
                                      // gespeichert
    //max_int = maximum (a, v);       // hier streikt der Compiler
    max_int = maximum ((int)a, v);    // der char a wird nach int
                                      // gecastet -> funktioniert
}
```

Die Funktion swap hat den Rückgabewert void. Mit **template** ⟨ **class** T ⟩ wird T als „Platzhaltertyp“ reserviert. Ab da ist die Funktionsdefinition wie gehabt - mit der Ausnahme, daß anstelle eines konkreten Typs der „Platzhaltertyp“ T verwendet wird. Die Funktion maximum hat zusätzlich T als Rückgabewert. Das Hauptprogramm zeigt, daß der Compiler über die übergebenen Variablentypen entscheidet, welche Versionen der Funktionen benötigt werden. Es ist klar, daß der Compiler einen Fehler meldet, wenn man nicht auf eine gewisse „Typensauberkeit“ achtet.

Von der Programmiererseite sieht es so aus, als würde man tatsächlich die selbe Funktion für verschiedene Typen benutzen. Das stimmt allerdings nicht: Der Compiler erzeugt für alle verwendeten Typen separate Funktionen. Die Konsquenz ist, daß Template-Funktionen immer komplett in Headerdateien stehen müssen: Template-Funktionen lassen sich nicht als solche übersetzen, sondern nur mit konkreten Typen. Im übersetzten Programm macht es keinen Unterschied, ob man template-Funktionen benutzt hat, oder sich für jeden Typ eine separate Funktion geschrieben hat. Templates ermöglichen eine elegantere Programmierung und verringern den Aufwand.

3.4.2 Template-Klassen

Das Konzept der Templates läßt sich auch auf ganze Klassen erweitern. Solche Klassen werden dann als Template-Klassen bezeichnet. Folgendes Beispiel illustriert die Syntax einer einfachen Template-Klasse:

```
/* --- Headerdatei Feld.h --- */

#include "FeldException.h"

template < class T > class Feld
{
private:
```

```
        T *feld_intern;
        int feld_intern_groesse;
public:
        T& operator[] (int i)
        {
                if (i>=feld_intern_groesse)
                        throw FeldException("Index ungueltig!");
                return feld_intern[i];
        }
public:
        Feld(int groesse)
        {
                feld_intern_groesse = groesse;
                feld_intern = new T[feld_intern_groesse];
        }
        ~Feld()
        {
                delete [] feld_intern;
        }
};

/* --- Hauptprogramm main.c --- */

#include "Feld.h"
#include "FeldException.h"
#include "Person.h"

int main()
{
        Feld<int>    int_feld(10);
        Feld<float>  float_feld(20);
        Feld<Person> personen_feld(30);

        int i;
        float f;
        Person p;

        // ...

        try
        {
                i = int_feld[3];
                f = float_feld[i];
                p = personen_feld[(int)f];
        }
        catch (FeldException e)
        {
                // Fehlerbehandlung ...
```

```
        }

        // ...
}
```

Auch für Template-Klassen gilt, daß sie nur in Headerdateien stehen dürfen (gleiche Gründe wie bei den Template-Funktionen). (nicht fertig)

3.5 Die Standardbibliothek

(nicht fertig)

A Zahlensysteme

A.1 Dezimalsystem

Im Dezialsystem benutzen wir einen Ziffernvorrat von zehn Ziffern $z_i \epsilon$ Z = {0,1,...,9}. Der Wert einer aus mehreren Ziffern zusammengesetzten natürlichen[58] Zahl
X = $z_{n-1}z_{n-2}...z_2z_1z_0$ errechnet sich mit der Formel

$$X = z_{n-1} * 10^{n-1} + z_{n-2} * 10^{n-2} + ... + z_2 * 10^2 + z_1 * 10^1 + z_0 * 10^0.$$

A.1.1 Umrechnung in andere Zahlensysteme

b soll die Basis des Zahlensystems sein, in das Umgerechnet werden soll. z_{i+1} ist eine Ziffer des Ziel-Zahlensystems. Die Zahl X ist die Dezimalzahl, die in das Ziel-Zahlensystem umgewandelt werden soll. Man benutzt folgenden iterativen Algorithmus[59]:

$$\text{Initialisierung: } i = -1,\ X_i = X$$

$$z_{i+1} = X_i \text{ modulo } b$$
$$X_{i+1} = \lfloor \frac{X_i}{b} \rfloor$$

Im folgenden Beispiel wird die Dezimalzahl $X = 2000$ in ihre Oktaldarstellung umgerechnet:

$$\text{Initialisierung:}$$
$$i = -1;\ X_i = X_{-1} = 2000;\ b = 8$$

$$z_0 = X_{-1} \text{ modulo } b = 2000 \text{ modulo } 8 = 0;\ X_0 = \lfloor \frac{X_{-1}}{b} \rfloor = \lfloor \frac{2000}{8} \rfloor = 250$$
$$z_1 = X_0 \text{ modulo } b = 250 \text{ modulo } 8 = 2;\ X_1 = \lfloor \frac{X_0}{b} \rfloor = \lfloor \frac{250}{8} \rfloor = 31$$
$$z_2 = X_1 \text{ modulo } b = 31 \text{ modulo } 8 = 7;\ X_2 = \lfloor \frac{X_1}{b} \rfloor = \lfloor \frac{31}{8} \rfloor = 3$$
$$z_3 = X_2 \text{ modulo } b = 3 \text{ modulo } 8 = 3$$

Die fertige Okalzahl setzt sich aus den errechneten Oktalziffern $z_3z_2z_1z_0$ zusammen. Die Dezimalzahl 2000_{10} entspricht also der Oktalzahl 3720_8.

A.2 Hexadezimalsystem

Das Hexadezimalsystem ist ein Zahlsystem zu Basis 16, d.h. es wird ein Ziffernvorrat von 16 Ziffern verwendet: $h_i \epsilon$ H = {0,1,...,9,A,B,C,D,E,F}[60]. Der Wert einer aus mehreren hexadezimal-Ziffern zusammengesetzten natürlichen Zahl X = $h_{n-1}h_{n-2}...h_2h_1h_0$ errechnet sich analog zum Dezimalsystem mit der Formel

$$X = h_{n-1} * 16^{n-1} + h_{n-2} * 16^{n-2} + ... + h_2 * 16^2 + h_1 * 16^1 + h_0 * 16^0.$$

Zahlen in hexadezimaler Darstellung können in C/C++ direkt eingegeben werden, wenn man den hexadezimalen Ziffern ein „0x“ voranstellt.

[58] eine natürliche Zahl ist eine ganze Zahl, die einen Wert größer gleich Null hat

[59] die Schreibweise $\lfloor x \rfloor$ bedeutet, daß die Zahl x zur nächsten ganzen Zahl abgerundet wird (ist x bereits eine ganze Zahl, wird nicht weiter abgerundet).

[60] A hat den Zahlenwert 10, ..., F hat den Zahlenwert 15

A.3 Oktalsystem

Das Oktalsystem ist ein Zahlensystem zur Basis 8. Der Ziffernvorrat besteht aus 8 Ziffern: $o_i \epsilon$ O = {0,1,...,7}. Die Berechnung des Zahlenwertes von X = $o_{n-1}o_{n-2}...o_2o_1o_0$ erfolgt mit der Formel

$$X = o_{n-1} * 8^{n-1} + o_{n-2} * 8^{n-2} + ... + o_2 * 8^2 + o_1 * 8^1 + o_0 * 8^0.$$

Zahlen in oktaler Darstellung können in C/C++ direkt eingegeben werden, wenn man den oktalen Ziffern eine „0“ voranstellt. Im Gegensatz zur Hexdezimaldarstellung wird die Okaldarstellung äußerst selten benutzt. Daß sie überhaupt von C/C++ unterstützt wird, hat wohl historische Gründe.

A.4 Dualsystem

Das Dualsystem ist ein Zahlensystem zur Basis 2. Der Ziffernvorrat besteht folglich nur aus 2 Ziffern: $d_i \epsilon$ D = {0,1}. Die Berechnung des Zahlenwertes von X = $d_{n-1}d_{n-2}...d_2d_1d_0$ erfolgt mit der Formel

$$X = d_{n-1} * 2^{n-1} + d_{n-2} * 2^{n-2} + ... + d_2 * 2^2 + d_1 * 2^1 + d_0 * 2^0.$$

Dualzahlen können in C/C++ leider **nicht** direkt eingegeben werden. Die Umrechnung in[61] Hexadezimal- oder Oktalzahlen ist aber sehr einfach.

A.4.1 Umrechnung ins Oktalsystem

Da eine Dualzahl aus drei Bits maximal einen Wert von 7 erreichen kann[62], lassen sich je drei Bits einer Dualzahl zu einer Oktal-Ziffer zusammenfassen.

A.4.2 Umrechung ins Hexadezimalsystem

Eine Dualzahl aus vier Bits kann maximal einen Wert von 15 erreichen[63]. Es lassen sich also je vier Bits zu einer Hexadezial-Ziffer zusammenfassen.

B Bedienung des Compilers

Der von uns verwendete DJGPP Compiler ist eine Portierung des unter Unix verwendeten GNU C/C++ Compilers. Nachdem mit der in C:\DJGPP liegenden Batch-Datei init.bat die notwendigen Pfade und Umgebungsvariablen initialisiert wurden, entspricht seine Bedienung seinem Unix-Pendant. Die folgende Kurzanleitung funktioniert also auch unter Unix.

[61] oder von

[62] Es gilt: $111_2 = 1 * 2^2 + 1 * 2^1 + 1 * 2^0 = 4 + 2 + 1 = 7 = 7_8$

[63] Es gilt: $1111_2 = 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 = 8 + 4 + 2 + 1 = 15 = F_{16}$

B.1 Übersetzung eines Quelltextmoduls

Für die Übersetzung eines C-Quelltextes sollte der **G**NU **C**-**C**ompiler *gcc* verwendet werden. C Quelltexte haben typischerweise die Dateiendung **.c**. Der GNU C++ Compiler heißt unter Unix *g++*. Die DJGPP-Version des C++ Compilers heißt *gxx*. Die Dateiendung eines C++ Quelltextes ist **.cc**, **.C** oder **.cpp**. Wir werden die Endung **.cc** für die Quelltexte und **.h** für die Headerdateien benutzen. Ein Quelltext quelltext.c/.cc läßt sich mit folgendem Kommando in ein Object-Modul quelltext.o übersetzen:

- für C: *gcc -c quelltext.c*
- für C++: *g++ -c quelltext.cc* (Unix) bzw. *gxx -c quelltext.cc* (DJGPP)

Mehrere Object-Moduln können zu einem Programm programm verlinkt werden, wobei genau ein Modul eine main-Funktion enthalten muß:

- für C: *gcc -o programm modul1.o modul2.o modul3.o*
- für C++: *g++ -o programm modul1.o modul2.o modul3.o*

Benötigt das Programm beim Übersetzen eine Headerdatei header.h, die im Unterverzeichnis headerdateien liegt und über #include ⟨header.h⟩ ins Programm eingebunden wird, muß der Compiler-Suchpfad für Headerdateien erweitert werden:

- *gcc -I./headerdateien -c quelltext.c* bzw. *g++ -I./headerdateien -c quelltext.c*

Werden Funktionen aus einer Bibliothek - z.B. der Mathematikbibliothek libm.a (DJGPP/Unix) / libm.so (Linux) - benutzt, so muß diese Bibliothek, wenn ein lauffähiges Programm erzeugt werden soll, mit *-lm*[64] hinzugelinkt werden:

- für C: *gcc -o programm modul1.o modul2.o -lm*
- für C++: *g++ -o programm modul1.o modul2.o -lm*

Benutzt man eine eigene Bibliothek, die nicht im Suchpfad des Compilers liegt, muß man mit *-L⟨Pfad⟩* den Suchpfad des Compilers für Bibliotheken erweitern. In diesem Beispiel ist der Pfad ein Unterverzeichnis und heißt bibs. Die Bibliothek heißt libmeinebib.a bzw. libmeinebib.so:

- für C: *gcc -o programm modul1.o modul2.o -L./bibs -lmeinebib*
- für C++: *g++ -o programm modul1.o modul2.o -L./bibs -lmeinebib*

Macht ein Programm Probleme, kann man sein Verhalten im *Debugger* gdb untersuchen. Dazu muß allerdings mit dem Parameter -g Debug-Information beim Übersetzen eingebunden werden. Beim Linken kann diese Debug-Information (und andere nicht benötigte Informationen) mit dem Parameter -s herausgefiltert werden. Den gleichen Effekt hat man, wenn man das Programm nachträglich mit dem Programm strip behandelt.

[64] Name der Bibliothek ohne das Prefix *lib* und ohne das Suffix *.a* bzw. *.so*

Um sich einen sauberen Programmierstil anzueignen, sollte man stets alle Warnungen des Compilers einschalten. Dies funktioniert beim Übersetzen mit dem Parameter -Wall.

Sollten hier Fragen offen bleiben, verweise ich auf die ausführliche Dokumentation des Compilers.

C ASCII-Tabelle

oct.	dez.	hex.	char	ESC	oct.	dez.	hex.	char	ESC
0000	0	0x00	NUL	'\0'	0100	64	0x40	@	-
0001	1	0x01	SOH	-	0101	65	0x41	A	-
0002	2	0x02	STX	-	0102	66	0x42	B	-
0003	3	0x03	ETX	-	0103	67	0x43	C	-
0004	4	0x04	EOT	-	0104	68	0x44	D	-
0005	5	0x05	ENQ	-	0105	69	0x45	E	-
0006	6	0x06	ACK	-	0106	70	0x46	F	-
0007	7	0x07	BEL	'\a'	0107	71	0x47	G	-
0010	8	0x08	BS	'\b'	0110	72	0x48	H	-
0011	9	0x09	HT	'\t'	0111	73	0x49	I	-
0012	10	0x0A	LF	'\n'	0112	74	0x4A	J	-
0013	11	0x0B	VT	'\v'	0113	75	0x4B	K	-
0014	12	0x0C	FF	'\f'	0114	76	0x4C	L	-
0015	13	0x0D	CR	'\r'	0115	77	0x4D	M	-
0016	14	0x0E	SO	-	0116	78	0x4E	N	-
0017	15	0x0F	SI	-	0117	79	0x4F	O	-
0020	16	0x10	DLE	-	0120	80	0x50	P	-
0021	17	0x11	DC1	-	0121	81	0x51	Q	-
0022	18	0x12	DC2	-	0122	82	0x52	R	-
0023	19	0x13	DC3	-	0123	83	0x53	S	-
0024	20	0x14	DC4	-	0124	84	0x54	T	-
0025	21	0x15	NAK	-	0125	85	0x55	U	-
0026	22	0x16	SYN	-	0126	86	0x56	V	-
0027	23	0x17	ETB	-	0127	87	0x57	W	-
0030	24	0x18	CAN	-	0130	88	0x58	X	-
0031	25	0x19	EM	-	0131	89	0x59	Y	-
0032	26	0x1A	SUB	-	0132	90	0x5A	Z	-
0033	27	0x1B	ESC	-	0133	91	0x5B	[	-
0034	28	0x1C	FS	-	0134	92	0x5C	\	'\\'
0035	29	0x1D	GS	-	0135	93	0x5D	]	-
0036	30	0x1E	RS	-	0136	94	0x5E	∧	-
0037	31	0x1F	US	-	0137	95	0x5F	_	-
0040	32	0x20	SPACE	-	0140	96	0x60	‘	-
0041	33	0x21	!	-	0141	97	0x61	a	-
0042	34	0x22	“	-	0142	98	0x62	b	-
0043	35	0x23	#	-	0143	99	0x63	c	-
0044	36	0x24	$	-	0144	100	0x64	d	-

Tabelle 6: Die 128 Zeichen des ASCII-Codes (Seite 1)

oct.	dez.	hex.	char	ESC	oct.	dez.	hex.	char	ESC
0045	37	0x25	%	-	0145	101	0x65	e	-
0046	38	0x26	&	-	0146	102	0x66	f	-
0047	39	0x27	’	-	0147	103	0x67	g	-
0050	40	0x28	(	-	0150	104	0x68	h	-
0051	41	0x29	)	-	0151	105	0x69	i	-
0052	42	0x2A	*	-	0152	106	0x6A	j	-
0053	43	0x2B	+	-	0153	107	0x6B	k	-
0054	44	0x2C	,	-	0154	108	0x6C	l	-
0055	45	0x2D	-	-	0155	109	0x6D	m	-
0056	46	0x2E	.	-	0156	110	0x6E	n	-
0057	47	0x2F	/	-	0157	111	0x6F	o	-
0060	48	0x30	0	-	0160	112	0x70	p	-
0061	49	0x31	1	-	0161	113	0x71	q	-
0062	50	0x32	2	-	0162	114	0x72	r	-
0063	51	0x33	3	-	0163	115	0x73	s	-
0064	52	0x34	4	-	0164	116	0x74	t	-
0065	53	0x35	5	-	0165	117	0x75	u	-
0066	54	0x36	6	-	0166	118	0x76	v	-
0067	55	0x37	7	-	0167	119	0x77	w	-
0070	56	0x38	8	-	0170	120	0x78	x	-
0071	57	0x39	9	-	0171	121	0x79	y	-
0072	58	0x3A	:	-	0172	122	0x7A	z	-
0073	59	0x3B	;	-	0173	123	0x7B	{	-
0074	60	0x3C	⟨	-	0174	124	0x7C	|	-
0075	61	0x3D	=	-	0175	125	0x7D	}	-
0076	62	0x3E	⟩	-	0176	126	0x7E	∼	-
0077	63	0x3F	?	-	0177	127	0x7F	DEL	-

Tabelle 7: Die 128 Zeichen des ASCII-Codes (Seite 2)