1.Einführung

“Dem Einsatz von Mikrocomputern am Arbeitsplatz und auch zu Hause gehört die Zukunft. Immer mehr Menschen werden vor die Frage gestellt, wie sie sich dieses neuen Werkzeugs bedienen wollen: Als im Kern hilflose 'Verbraucher' fremder Programme, die zeitlebens ausschließlich auf vorgefertigtes Software-Werkzeug angewiesen bleiben, oder als mündige Rechner-Nutzer.

Der mündige Rechner-Nutzer ist soweit Herr über den Computer, als dass er der Maschine seinen Willen mitzuteilen und ihr gegenüber diesen Willen durchzusetzen versteht: Das heißt, er kann bis zu einem gewissen Grad selbständig programmieren. Wer seinen Rechner nicht zu programmieren vermag, bleibt der bloße 'Bediener' fremder Programme und in Unmündigkeit gegenüber einem ihm unbegreiflichen System aus Computer und Software gefangen.

Vor diesem Hintergrund beobachtet man mit Bedauern Tendenzen, die geeignet sind, Menschen auf einen falschen Weg zu locken, dessen Begehen ihnen langfristig sehr schaden wird. Falsche Propheten weissagen einer verunsicherten Zuhörerschaft, die Zukunft gehöre dem integrierten Universal-Anwenderprogrammpaket.”

Dieses etwas längere Zitat stammt nicht von mir, sondern aus einer Zeit als MS-DOS gerade mal ein Jahr alt war und an ein UNIX für jedermann überhaupt noch nicht zu denken war. Es stammt von Hans-Georg Joepgen der 1985 ein nettes Büchlein über die erste Version von Turbo-Pascal geschrieben hat. Erstaunlich dabei ist der Weitblick, der in seinen Zeilen steckt, zudem diese seine Aussagen aktueller denn je sind. Leider betrifft es heute nicht nur irgendwelche Anwendungen sondern auch Entwicklungswerkzeuge bzw. Entwicklungssysteme. Auch hier – leider immer mehr selbst aus Entwicklerkreisen – ist eine Tendenz zu verzeichnen, dass man sich an Entwicklungssysteme großer Hersteller bindet deren “vorgekaut” generierter Code kaum oder nur unter erheblichen Schwierigkeiten portabel zu anderen Entwicklungswerkzeugen ist. Viele Entwickler lehnen es heutzutage ab, sich mit Fragen der Speicherverwaltung zu beschäftigen und überlassen dieses lieber Mechanismen ihres Entwicklungssystems. Ob das alles zu besseren und effizienteren Programmen führt möchte ich bezweifeln.

Was Programmiersprachen betrifft leben wir, ebenso wie in der realen Welt der menschlichen Sprachen, inmitten eines babylonischen Durcheinanders, in dem es für den Nutzer immer schwerer wird sich zu entscheiden welche Sprache ein spezielles Problem am besten löst.
Die Ansprüche sind dabei in der Regel hoch, Flexibilität und hoher Abstraktionsgrad sind dabei genauso wichtig wie die Wiederverwendbarkeit von Code oder die Portabilität. Vor allem die fortschreitende Vernetzung heterogener Systeme und Architekturen schreit gerade zu nach portablen Lösungen. Um dieser Entwicklung Rechnung zu tragen sind in den vergangenen 10-12 Jahren von vielen Seiten Anstrengungen unternommen worden. Auf der einen Seite ist es z.B. die Standardisierung der Programmiersprache C++ die für alle Compiler-Hersteller verbindliche Regeln festsetzt was den Sprachumfang betrifft sowie auf der anderen die Schaffung neuer Sprachmittel wie “Java” oder “Dot Net”. Auf Java und Co. möchte ich an dieser Stelle nicht weiter eingehen, weil es in diesen Vortrag nicht um einen Vergleich zwischen verschiedenen Sprachphilosophien geht sondern um die in C++ geschriebene Entwicklungsbibliothek Qt. Eine kleine Bemerkung sei mir trotzdem gestattet, Java benötigt auf der Zielplattform eine “virtuelle Maschine” die den Javabytecode in Maschinenbefehle umsetzt. Bei Qt und ähnlichen Toolkits muss der Sourcecode, also die geschriebenen Quellen, für die jeweilige Architektur bzw. das jeweilige Betriebsystem mit einen Compiler – hier natürlich ein C++-Compiler – übersetzt werden. Das heißt, es gibt hier keine Zwischenschicht wie bei Java, sondern die Programme werden nativ für das betreffende System

übersetzt, gebunden und somit erstellt. Qt-Programme kann man nicht für ein System z.B. Linux erstellen um sie dann unter Windows laufen zu lassen. Plattformunabhängig ist daher nicht unbedingt der richtige Begriff, portabel ist hier wohl angebrachter.
Aber dazu später, wenden wir uns zunächst den Grundlagen zu, die benötigt werden um überhaupt die Philosophie und grundlegenden Konstrukte des Qt-Toolkits zu verstehen. Leider finden die allgemeinen Grundlagen in vielen Veröffentlichungen und Tutorials zu wenig Beachtung oder werden einfach vorausgesetzt. Einige Begriffe der objektorientierten Programmierung, wie Objekt oder Instanz, sind oftmals mehrdeutig und haben u.U. bei unterschiedlichen Sachverhalten unterschiedliche Bedeutung, was gerade für Programmieranfänger schnell zu Frustration führen kann. Man braucht nur in Entwicklungsforen oder Mailinglisten zu schauen, um zu sehen, wie die Leute oftmals aneinander vorbei reden.
Der Grund liegt hier einfach in der Historie der objektorientierten Programmierung, denn viele strukturierte Sprachen sind nach und nach mit objektorientierten Erweiterungen versehen worden und jede hat ihre eigene Interpretation der Begriffe eingeführt.
Wie schon erwähnt ist Qt eine in C++ geschriebene Klassenbibliothek, daher empfiehlt es sich ein wenig mit dieser Sprache zu beschäftigen. Denn gerade im deutschsprachigen Qt-Forum erlebe ich oft, dass dort Antworten auf Fragen gegeben werden die nicht nur falsch sind, sondern auch die Unkenntnis einiger Nutzer in den Grundlagen von C++ aufzeigen.

2. Was ist eigentlich C++

Bjarne Stroustrup, wenn man so will der Erfinder dieser Programmiersprache, würde C++ kurz und bündig als universelle Programmiersprache die vorzugsweise zur Systemprogrammierung dient, beschreiben. Oft werden die Programmiersprachen C und C++ in einen Atemzug genannt ohne jedoch auf die wichtigen Unterschiede in der Sprachstruktur sowie Philosophie hinzuweisen. Unübersehbar ist das die Sprache C als Basis für die Definition von C++ gewählt wurde und dort weiterhin als Untermenge zur Verfügung steht, allerdings in einen etwas anderen Kontext. Auf die wesentlichsten Unterschiede beider Sprachen werde ich einige Zeilen später noch genauer kommen. Im übrigen hat C++ aber noch weitere Paten, so z.B. Simula67 woraus das Klassenkonzept stammt. C++ ist, wie viele andere auch, eine sogenannte Hybridsprache mit der man sowohl strukturiert sowie objektorientiert programmieren kann. Da Qt eine objektorientierte Klassenbibliothek ist, wird dieser Aspekt einen wichtigen Platz in unserer Betrachtung einnehmen.
Eine sogenannte “low-level” - sprich hardwarenahe - Programmierung, ein Erbe der Sprache C, ist genauso möglich wie eine Programmierung auf einer höheren Abstraktionsebene. Ein weiter wichtiger Aspekt ist, dass C++ die Möglichkeit bietet in C geschriebene Bibliotheksfunktionen zu nutzen, unter der Voraussetzung, dass diese binderkompatibel und ANSI-Konform sind.
C++ besitzt nur wenige sogenannte Schlüsselworte und keine in die Sprache eingebauten High-level-Datentypen. Werden solche Typen benötigt, können sie mit C++ definiert werden. Die Standardbibliothek, die zu jedem C++-Compiler gehört, liefert viele Beispiele neu definierter Typen, wie Zeichenketten oder Container.
So wie in C korrespondieren die grundlegenden Sprachmittel, wie Typen, Operatoren und Anweisungen, möglichst eng mit maschinennahen Dingen, wie Zahlen, Zeichen und Adressen. C++ bedarf daher, im Gegensatz zu anderen Sprachen, nur minimaler Laufzeitunterstützung.
C++ unterstützt eine Vielzahl von Programmierstilen. Alle basieren auf einer strengen statischen Typenprüfung. Die meisten haben außerdem zum Ziel, einen hohen Grad an Abstraktion zu erreichen und die Ideen des Programmierers möglichst direkt zu repräsentieren. Um C++ zu erlernen sind keine Vorkenntnisse in der Sprache C nötig. Es ist ohnehin eher darauf zu orientieren, dass der Programmierer, ob er nun das Programmieren in einer anderen Sprache gewöhnt ist oder nicht, Zeit für die Verinnerlichung der C++-Programmiertechniken investieren muss.

Programmiertechniken die aus anderen Programmiersprachen gedankenlos übernommen werden führen in der Regel zu schlecht laufenden und schwer zu wartenden Programmen. Es ist schon beim Schreiben ärgerlich, wenn man bei den Compiler-Fehlermeldungen ständig daran erinnert wird, dass man nicht in seiner gewohnten Sprache programmiert. Das heißt nicht, dass andere Sprachen keine Ideen für die Programmierung mit C++ liefern können, diese müssen jedoch so angepasst werden, dass sie zur generellen Struktur und dem Typsystem von C++ passen.

2.1Abgrenzung zwischen C und C++

Im vorigen Abschnitt habe ich es schon angedeutet, dass es einige Unterschiede zwischen C und C++ gibt. Glücklicherweise sind es nur wenige Unterschiede die wirklich zu Inkompatibilitäten führen und letztlich zu Problemen. Solange man sich an die neuesten Spezifikationen der Sprache C hält und auf veraltete Sprachkonstrukte verzichtet, dürften kaum ernsthafte Probleme auftreten. Viele Unterschiede beziehen sich eher darauf, dass es in C++ typensichere Möglichkeiten gibt um z.B. dynamisch Freispeicher anzufordern oder explizite Typenumwandlungen vorzunehmen.

Beispiel 1:

Funktionen in C können mit einer Syntax definiert werden, die optional alle Argumententypen nach der Liste der Argumente spezifiziert

```
/* veraltet */
void f(a,b,c)
int a;
char* b;
double c;
{
	/* Funktionsrumpf */
}

Solche Definitionen müssen neu geschrieben werden

/* neu */
void f( int a, char* b, double c)
{
	/* Funktionsrumpf */
}
```

Das die Verwendung der in C++ neu eingeführten Schlüsselworte wie “this”, “new” oder “delete” usw. nicht als Funktions- bzw. Variablennamen verwendet werden können dürfte klar sein, ganz abgesehen davon würde der C++-Compiler in so einen Falle eine Fehlermeldung ausgeben.
Im Gegensatz zu C müssen in C++ lokale Variablen die z.B. in einer Funktion verwendet werden nicht am Anfang des Funktionsrumpfes vereinbart werden sondern erst dort wo sie auch gebraucht werden.

Beispiel 2:

```
/* C */
void f(){
```

```
  int i; /* Zählvariable in der gesamten Funktion gültig */
  char* str = “Hallo World”;
  for( i= 0; i < strlen(str) ; i++){
      printf( "str[%d]: %c \n",i,str[i]);
  }
}

/* C++ */
void f(){
  char* str= “Hallo World;
  for( int i= 0; i < strlen(str) ; i++){ /* i nur in der for-Schleife gültig */
      printf( "str[%d]: %c \n",i,str[i]);
  }
}
```

Für die dynamische Speicherverwaltung werden in C++ die Operatoren “new” und “delete” eingeführt. Mit “new” wird Speicher für ein Datentyp angefordert und mit “delete” wieder freigegeben. Damit werden die in C bekannten Speicherplatz-Funktionen “malloc()”, “calloc()” sowie für die Reservierung von Zusatzspeicher verwendete Funktion “realloc()” und die für die Freigabe des angeforderten Speicher zuständige Funktion “free()” ersetzt.
Die neuen Operatoren “new” und “delete” bieten mehrere Vorteile :

- Sie gehören zum Sprachumfang von C++ und nicht zu einer Standardbibliothek.
- Der Typ für den Speicher angefordert wird muss nicht mehr als Parameter geklammert und mit “sizeof” versehen werden, sondern kann als Operand direkt angegeben werden.
- Es kann der Operator “new” für eigene Datentypen selbst implementiert werden um damit die Speicherverwaltung zu optimieren.

Oftmals werden in der Praxis C-Bibliotheken verwendet die logischerweise intern mit den Speicherplatz-Funktionen arbeiten. Das ist auch nicht weiter tragisch, da die Speicherbehandlung in sich abgeschlossen ist. Aber was man tunlichst vermeiden sollte, ist eine Mischung von beiden in den zu erstellenden Programmen.
Früher oder später wird man bei der Beschäftigung mit C oder C++ auf Begriffe wie “call by reference” oder “Referenz” stoßen. Aber was ist eigentlich damit gemeint? Nun, in C gibt es prinzipiell zwei Möglichkeiten auf Daten im Speicher zuzugreifen. Die erste Möglichkeit ist über Variablennamen und die zweite über sogenannte Zeiger (engl. pointer). Zeiger sind im Prinzip nichts anderes als Variablen die die Adresse einer Speicherstelle eines bestimmten Datentyps speichern. Ein Zeiger verweist auf eine Speicherstelle und gibt mit den Typ gleichzeitig an, wie diese Speicherstelle zu lesen und zu beschreiben ist. In C, wie auch in C++, können Zeiger jeden beliebigen Typ haben. Deklariert werden solche Zeigervariablen indem der Zeigeroperator “*” hinter dem Datentyp und vor dem Variablennamen geschrieben wird. Um die Adresse einer Variablen zu ermitteln muss der Adressoperator “&” verwendet werden.

Beispiel 3:

```
int a; // Integervariable
int b= 7; // Integervariable schon mal initialisiert
a=b; // Zuweisung
int* zeiger; // Zeiger auf einen Integer
```

```
zeiger = &a; // Zeiger erhält die Adresse von a
*zeiger = b; // Wertzuweisung an die Speicherstelle auf die “zeiger” zeigt
```

In C++ hingegen gibt es noch eine weitere Möglichkeit auf Daten zuzugreifen, und zwar über Referenzen. Eine Referenz kann man sich als alternativen Namen oder Alias einer Variablen vorstellen. Im Gegensatz zu einen Zeiger wird hier also nicht die Adresse einer Speicherstelle in eine andere Variable (Zeigervariable) gespeichert, sondern nur ein zusätzlicher Name für eine bereits existierende Speicherstelle vergeben. Daher muss, um sicherzustellen, dass eine Referenz ein Name für etwas ist, diese initialisiert werden. D.h. die Variable, die referenziert werden soll, kann der Referenz ausschließlich bei der Deklaration zugewiesen werden. Jede im Programmablauf folgende Zuweisung an eine Referenz bedeutet eine Zuweisung an die referenzierte Variable. Ist daher eine Referenz, also ein zweiter Name, auf eine Variable angelegt, kann diese genau wie die Variable selbst weiterverwendet werden. Was gelegentlich für Verwirrung sorgt, ist der Gebrauch des “&” als Referenzoperator den wir ja schon als Adressoperator kennen gelernt haben. Die jeweilige Bedeutung ergibt sich daraus auf welcher Seite der “&”-Operator bei einer Zuweisung steht. Steht er vor einer Variablen auf der rechten Seite einer Zuweisung, den sogenannten (R-Wert), dann ist er ein Adressoperator der die Adresse einer Variablen oder Funktion ermittelt. Wird er dagegen vor einer Variablen hinter dem entsprechenden Datentyp auf der linken Seite (L-Wert) bei einer Deklaration gebraucht mutiert “&” zum Referenzoperator.

Beispiel 4:

```
int a = 4; // Zuweisung an eine Variable vom Typ Integer
int b; // Variable vom Typ Integer
int& ref = b; // Referenz auf den Integer b ( ref und b repräsentieren dieselbe Speicherstelle )
ref = 6; // Wertzuweisung an ref und damit zugleich an b

int *zeiger = &a; // Verwendung des Adressoperators
int & referenz = a; // Verwendung des Referenzoperators
```

Aber warum haben der oder die Schöpfer von C++ diese auf dem ersten Blick recht verwirrende Eigenschaft der Sprache hinzugefügt? Nun, dafür gibt es genau zwei Gründe die sich zum einen aus der Praxis sowie zum anderen aus der Sprachphilosophie von C++ ergeben. Nehmen wir uns zunächst den rein praktischen Aspekt vor. Oftmals steht der Programmierer vor dem Problem, dass er mehr als einen Rückgabewert einer Funktion benötigt. Aus guten Grund, den ich an dieser Stelle nicht weiter beleuchten möchte weil es sonst einfach zu weit führen würde, kann nur ein Wert von einer Funktion zurückgegeben werden. Dieser Fakt trifft bei allen Programmiersprachen die ich kenne zu. Daher wäre es doch recht Praktisch, wenn man die Variablen, die als Parameter einer Funktion übergeben und in ihr weiterverarbeitet also geändert werden weiterverwenden könnte. Mit ganz normalen Variablen (Werten) ist das nicht realisierbar, da die Parameter nur Kopien der übergebenen Argumente darstellen und nachdem die Funktion während des Programmlaufs abgearbeitet wurde nicht mehr zur Verfügung stehen. Eine Möglichkeit dieses zu realisieren wäre die Verwendung von Zeigern als Funktionsparameter denen man beim Aufruf die Adressen von außerhalb der Funktion gültigen Variablen gleichen Typs übergibt. Hier wird innerhalb der Funktion mit Variablen gearbeitet die die Adresse von Speicherstellen besitzen die außerhalb der Funktion vereinbart wurden. Da man also die Adressen der Variablen kennt die z.B. verändert werden sollen, kann man auch auf ihre Inhalte zugreifen und sie verändern. Nach dem verlassen der Funktion während des Programmlaufs sind zwar die Zeigervariablen die wir als Parameter

vereinbart haben nicht mehr gültig, aber die Variablen deren Adressen wir übergeben haben existieren weiterhin. Dieses Vorgehen ist unter C üblich und wird dort als sogenanntes “call by reference” bezeichnet obwohl gar keine Referenzen im engeren Sinne verwendet werden. Im Gegensatz dazu besteht in C++ die Möglichkeit wirklich Referenzen einzusetzen um derartige Ergebnisse zu erzielen, also ein echtes “call by reference” durchzuführen ohne den Umweg über Zeiger zu nutzen. Wenn man Parameter als Referenzen deklariert, werden diese bei einem Funktionsaufruf mit den übergebenen Argumenten initialisiert. Es handelt sich dann aber nicht um eine Kopie, sondern um einen zweiten Namen für das angegebene Argument. Jede Änderung die in der Funktion am Parameter durchgeführt wird, wird somit auch am Argument welches übergeben wurde, durchgeführt.

Beispiel 5:

```
/* Call By Reference – Mechanismen
*Die Funktionen sollen zwei gegebene Werte vertauschen!
*/

#include <stdio.h>

void swap_falsch( int a, int b ); // call by value
void swap_zeiger(int* a, int* b); // C-like Unechtes call by reference
void swap_referenz( int& a, int&b ); // C++ Echtes call by reference

int main(){
	int x = 1, y = 9;

	swap_falsch( x, y );
	printf(“\nErgebnis Call By Value : x= %d  y= %d Falsch!\n”, x, y );
	swap_zeiger( &x, &y );
	printf(“Ergebnis tauschen über Zeiger : x= %d y= %d\n”, x, y );
	swap_referenz( x, y );
	printf(“Ergebnis Call By Reference : x= %d  y= %d\n”, x, y );

	return 0;

}

void swap_falsch( int a, int b ){// kein Vertauschen möglich
	int tmp = a;		// hier werden nur die lokalen Variablen vertauscht
	a = b;
	b=tmp;
}

void swap_zeiger( int* a, int* b ){
	int tmp = *a;		// den Inhalt der übergebenen Speicherstelle vertauschen
	*a = *b;
	*b=tmp;
}
```

```
void swap_referenz( int &a, int &b ){
	int tmp = a;		// tauschen über Alias-Namen der übergebenen Variablen
	a = b;
	b=tmp;
}
```

Das Ergebnis würde so aussehen:

```
rechner /home/cppman>
Ergebnis Call By Value : x= 1  y= 9 Falsch!
Ergebnis tauschen über Zeiger : x= 9 y= 1
Ergebnis Call By Reference : x= 9  y= 1
```

Das Arbeiten mit Referenzen als Parameter einer Funktion kann, so nützlich es auch sein mag, schnell zum Alptraum werden. Es müssen zwar keine Zeiger mehr übergeben werden um Argumente zu verändern, aber es gibt keine Sicherheit mehr, dass bei einen Funktionsaufruf die übergebenen Argumente nicht verändert werden können. Am Funktionsaufruf ist nämlich nicht zu erkennen ob nun mit Referenzen gearbeitet wird. Allein die Deklaration der Parameter gibt darüber Aufschluss. Im obigen Beispiel ist leicht zu sehen, dass x und y wie einfache Integer übergeben werden. Einfache Referenzen sollten demnach nur dann verwendet werden wenn wirklich eine Änderung der übergebenen Argumente erfolgen soll. Damit kommen wir zum zweiten wichtigen Punkt der letztlich für die Verwendung von Referenzen spricht.
Schauen wir uns zunächst noch einmal das sogenannte “call by value” an. Die Funktion “swap_falsch” legt Kopien der übergebenen Argumente x und y an. In der C++-Welt spricht man vom aufrufen der Copy-Konstruktoren. Aber was ist darunter zu verstehen? Ohne an dieser Stelle auf die objektorientierten Aspekte von C++ genauer einzugehen, hierzu ein paar Vorbemerkungen. In vielen Programmiersprachen ist es möglich, eigene Datentypen zu deklarieren die aus unterschiedlichen fundamentalen Datentypen bzw. aus schon deklarierten eigenen Datentypen bestehen. Für diese “Datenverbünde” gibt es, in den verschiedenen Sprachen, unterschiedliche Bezeichnungen wie z.B. Records oder Strukturen. Im folgenden möchte ich genau wie in C oder C++ üblich für diesen Sachverhalt die Bezeichnung “Struktur” verwenden. Praktische Anwendungen für solche Strukturen gibt es genug. So eine Struktur kann z.B. den Datensatz einer Person oder eine “Inode” im Ext3Fs repräsentieren. Strukturen bieten damit die Möglichkeit eine Abstrahierung, etwa die Zusammenfassung verschiedener Einzelteile(Komponenten), in Programmen auszudrücken.
Ohne hier an dieser Stelle genauer auf den in C++ verwendeten Begriff “Klasse” einzugehen, kann man eine Klasse als C-Struktur auffassen die mit zusätzlichen Eigenschaften ausgestattet ist, wie z.B. Zugriffskontrolle oder Funktionen als Komponenten. Unter anderen kann eine solche “Klasse” besondere Funktionen als Komponenten beinhalten die dafür zuständig sind die Art und Weise festzulegen wie die einzelnen Komponenten dieser Klasse und damit sie selbst zu initialisieren sind. Diese besonderen Funktionen nennt man Konstruktoren. Sind keine Konstruktor-Funktionen explizit angegeben, wie in einfachen Datentypen oder Strukturen, generiert der Compiler in der Regel einen “Default-Konstruktor”. Ein “Default-Konstruktor” ist ein Konstruktor, der ohne Angabe eines Parameters aufgerufen werden kann. “Copy-Konstruktoren” sind Konstruktoren die anhand einer existierenden Variable eine direkte Kopie davon anlegen.
Wird eine Strukturvariable in einem Programm erzeugt, erfolgt die Initialisierung jeweils für jede einzelne Komponente, also den in der Struktur verwendeten Daten, nach und nach. Sie wird sozusagen konstruiert. Jeder Kopiervorgang erfordert daher die Neukonstruktion einer Struktur, die mit den Werten der zu kopierenden Struktur ebenso “komponentenweise” initialisiert wird. Im Falle

einer Struktur wird, da hier kein Copy-Konstruktor angegeben ist, der “Default-Copy-Konstruktor” aufgerufen, der die “komponentenweise” Initialisierung vornimmt. Es ist natürlich leicht einzusehen, dass die Anzahl der Komponenten der Struktur sowie deren Beschaffenheit, Strukturen können wiederum Strukturen als Komponenten besitzen, die Zeit beeinflusst in der die Kopie durchgeführt wird. D.h. je komplexer diese Strukturen aufgebaut sind, kann dies zu einen erheblichen Laufzeitnachteil führen. Aber wie kann man Laufzeitnachteile vermindern oder verhindern? Nun, nach dem bisher gelesenen(besser: Nach dem Bisherigen könnte...) könnte man auf die Idee kommen die unter C übliche Alternative zu verwenden indem Zeiger als Parameter übergeben werden. Die Parameter enthalten dann die Adressen der zu ändernden Datentypen. Allerdings wäre es nicht mehr möglich mit Strukturen so zu rechnen wie mit fundamentalen Datentypen.
Der Ausweg aus diesen Dilemma heißt hier Referenzen als Parameter. Allerdings steht hier noch das Problem der Veränderlichkeit der übergebenen Argumente. Aber auch da gibt es eine einfache Lösung. Jede Variable kann in C++ als Konstante definiert werden! D.h. bei der Deklaration muss einfach das Schlüsselwort “const” angegeben werden.

Beispiel 6 :

```
/* Deklaration von Konstanten */

const int anzahl = 3;
const double pi = 3.1415926;
```

Die Compiler testen in diesem Falle übrigens ab ob der Wert verändert wird oder nicht.
Das können wir auch bei der Deklaration der Parameter als Referenzen machen, sie also als konstante Referenzen deklarieren. Somit haben wir gleich mehrere Fliegen mit einer Klappe erschlagen. Wir können so verhindern, dass die Werte der Argumente verändert werden oder Copy-Konstruktoren aufgerufen werden, sowie schaffen die Möglichkeit einer Übergabe von Konstanten an die Funktion.

Beispiel 7 :

```
/* Referenzen als Konstanten */

void f( const int &a ){
        a = 5; // geht nicht Compiler-Fehler!
        ....
}
```

Fassen wir also zusammen, sollen der als Referenz übergebene Parameter nicht geändert werden, ist es besser ihn als Konstante zu deklarieren.
Das ich mich so ausführlich mit diesen Thema beschäftigt habe liegt daran, dass in QT diese Art der Parameterdeklaration sehr oft zu finden ist, der Grund dafür sollte jetzt jedem klar sein.
Eine weitere wichtige Besonderheit in C++ ist die Vorbelegung von Funktionsparametern mit einen Wert. In der QT-Bibliothek wird davon häufig Gebrauch gemacht um z.B. wenn Aufzähltypen dafür genutzt werden sollen, die die Aufgabe der Funktion genauer spezifizieren.

Beispiel 8 :

```
/* Hier ist der Parameter cs mit Qt::CaseSensitive vorbelegt */
bool QString::contains( const Qstring & str, Qt::CaseSensitivity cs = Qt::CaseSensitive ) const

/* Der Aufruf könnte dann so aussehen */
QString str = “C++ ist eine tolle Sprache”;
str.contains(“Sprache”) // liefert “true”
```

Diese Funktion der Klasse QString liefert, je nach dem ob die in der Funktion angegebene Zeichenkette in dem QString “str” vorhanden ist oder nicht, dementsprechend “true” oder “false” zurück. Voreingestellt für die Funktion ist “ Qt::CaseSensitive” also abhängig von der Groß- bzw. Kleinschreibung. Der Wert “ Qt::CaseSensitive” ist schon bei der Deklaration angegeben worden. Dieser angegebene “Default-Wert” wird vom Compiler automatisch eingesetzt, wenn nur eine Zeichenkette als Argument übergeben wird. Man hat dadurch etwas weniger Schreibarbeit wenn man denn den voreingestellten Wert nutzen will. Falls man eigene Funktionen schreibt und diesen Sachverhalt nutzen möchte sollte man folgendes beachten. Die Argumente einer Funktion werden beim Aufruf der Reihenfolge nach aufgerufen. Also das erste Argument ist der erste Parameter und so weiter. D.h. das die Default-Werte immer nur nach denen stehen können, die mit übergebenen Argumenten belegt sind.
Es gibt natürlich noch mehr Vereinfachungen gegenüber C, deren Bedeutung ich allerdings nicht so hoch einschätze um sie hier genauer zu beleuchten.
Nun, das wären die grundlegenden Unterschiede bzw. Besonderheiten der Sprachen C und C++. Wenden wir uns nun etwas den Sprachmitteln von C++ zu, die wichtig für das Verständnis der QT-Bibliothek sind.

2.2 Grundlegende Sprachmittel in C++

An dieser Stelle möchte ich nicht anfangen zu erklären wie irgendwelche Schleifen oder Kontrollstrukturen in C++ dargestellt werden. Diese Informationen können aus der Vielzahl von schriftlichen Veröffentlichungen oder im Internet bezogen werde. Statt dessen möchte ich an dieser Stelle einige wichtige Begriffe, Möglichkeiten oder Verfahrensweisen erklären, die in vielen Publikationen - wenn überhaupt - nur Ansatzweise beschrieben sind. Einige wichtige Aspekte in C++, wie Referenzen, sind schon im vorigen Abschnitt behandelt worden, so dass sich eine nochmalige Klärung erübrigt. Auf Begriffe aus der objektorientierten Programmierung werde ich in einen eigenen Abschnitt gesondert eingehen.
Also Ärmel hoch und los gehts!
Den Begriff “Zeiger” als solches haben wir schon erklärt. Offengeblieben dabei sind einige Fragen zur Initialisierung, den Zusammenhang zwischen Zeiger und Felder sowie zur sogenannten Zeigerarithmetik. Letzteres wird denen die z.B. C nicht kennen sehr befremdlich vorkommen, aber es ist wahr, man kann mit Zeigern gewissermaßen rechnen. Bleiben wir zunächst bei der Initialisierung. Wird eine Zeigervariable vereinbart ohne sie zu initialisieren liegt ein undefinierter Zustand vor, denn die in ihr gespeicherte Adresse ist irgendeine zufällige. D.h. der Zeiger zeigt irgendwo hin, was bei einen Zugriff auf solch ein uninitialisierten Zeiger fatale Folgen haben kann. Daher sollte eine Zeigervariable immer initialisiert, also in einen definierten Zustand gebracht werden. Einen Zeiger kann initialisiert werden indem man ihm eine Adresse einer Variablen oder einen anderen Zeiger zuweist. Aber was ist wenn wir die Adresse vor dem Programmlauf noch nicht kennen? Dann kommt die Konstante “NULL” ins Spiel. Was ist aber darunter zu verstehen?
Nun, “NULL” ist eine Konstante die für einen Zeiger steht der nirgendwo hinzeigt, also keine wirkliche Adresse hat. Wird nun ein Zeiger mit “NULL” initialisiert erhalten wir einen definierten

Zustand. Zumindest können wir jetzt im Programm abprüfen ob den Zeiger eine sinnvolle Speicheradresse zugeordnet wurde oder nicht. Intern gesehen ist “NULL” nichts anderes als ein Integer mit dem Wert 0, der an Zeiger zugewiesen werden darf. Daher sieht man oft dass Zeiger direkt mit den Wert 0 initialisiert werden.
Wer sich schon einmal mit der Programmierung beschäftigt hat kennt Felder (Arrays) die aus mehreren Elementen des gleichen Typs bestehen, die sequenziell angeordnet sind. Der Zugriff auf die einzelnen Feldelemente erfolgt in C oder C++ üblicherweise über den Indexoperator [].
Allerdings bestehen in C und C++ eine enge Verwandschaft zwischen Feldern und Zeigern. In C++ und in C bezeichnet nämlich der Name des Feldes zugleich die Anfangsadresse des Feldes. Genauer gesagt bezeichnet der Name einer Feldvariablen – also der Feldbezeichner – den Zeiger auf das erste Feldelement. Dadurch kann ein Zugriff auch über Zeiger auf ein Feld erfolgen.

Beispiel 9 :

/* Felder und Zeiger */

```
char feld[] = “Zeichenkette”; // eine einfache Zeichenkette
char* zeiger = NULL;  // mit NULL initialisierter Zeiger vom Typ “char”
zeiger = feld; // Zuweisung der Anfangsadresse des Feldes an einen Zeiger
zeiger = &feld[0]; // analog wie oben
```

Damit ist “feld” ein Zeiger vom Typ char auf die Speicherstelle des ersten Feldelements, also auf die Speicherstelle wo das “Z” abgelegt ist. In einen Ausdruck sind darum “feld” und “&feld[0]” äquivalent. Der Zugriff auf einzelne Feldelemente ist damit in C++ genau wie in C sehr eng mit der Zeigerarithmetik verbunden. Hier dazu noch ein Beispiel:

Beispiel 10 :

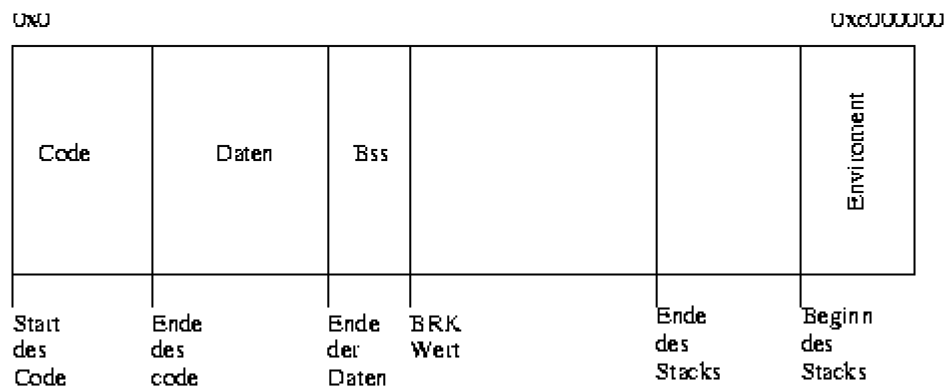
```
int feld[] = { 1, 2, 3, 4 }; // Array mit vier Inter gern
int* zeiger = feld; // Zuweisung an einen Zeiger
*(zeiger + 2 ) = 6; // Zuweisung an das dritte Feldelement
zeiger[3] = 5;  // Zeiger kann genau wie das Feld angesprochen werden

/* der Inhalt von feld wäre jetzt 1,2,6,5 */
```

Es mag auf den ersten Blick ein wenig seltsam anmuten, dass auch bei Zeigern die bei Feldern übliche Schreibweise genutzt werden kann. Intern macht der Compiler ohnehin nichts anderes daraus als: “Gehe von der Anfangsadresse aus, die angegebenen Male die Größe des Datentyps in Byte weiter!”. Das gilt für das Feld selbst wie auch für den Zeiger der darauf zeigt.
Es ist also möglich, Zeiger und ganzzahlige Werte zu addieren bzw. zu substrahieren, das was man als Zeigerarithmetik bezeichnet. Dabei wird die Größe des Datentyps, auf das ein Zeiger zeigt , automatisch berücksichtigt. Allerdings wird nicht geprüft ob eventuell die Feldgrenzen überschritten sind. Das eben gesagte trifft auch auf den Index-Operator [] zu. Da ist der Programmierer selbst in der Pflicht! Die Containerklassen vieler Bibliotheken, u.a. die Standard Template Bibliothek, bieten dafür eine Funktion “at()” an. Falls man sich also die Arbeit sparen will selbst zu prüfen ob man innerhalb der Feldgrenzen operiert, sollte diese Funktion benutzt werden.
Da wir uns ohnehin schon mit den Zeigern aufhalten, können wir uns auch gleich einige Grundregeln für die dynamische Speicherverwaltung vornehmen. Um aber genauer darauf eingehen zu können, sollten wir erst einmal darüber sprechen wie im Grundsatz die Speicherverwaltung von

Programmen funktioniert.
Ein C++-Programm wird vom C++-Compiler übersetzt. D.h. er wandelt die C++-Anweisungen in die Maschinensprache des Prozessors um und legt diese in einer Binärdatei ab. Das macht er natürlich nicht willkürlich, sondern er legt den Binärcode formatiert ab. Jedes Betriebssystem hat dabei sein eigenes Binärformat das Aufbauprinzip ist jedoch immer sehr ähnlich. Unter LINUX wird dabei das sogenannte ELF-Format benutzt. Auf das ELF-Format möchte ich an dieser Stelle nicht weiter eingehen, weil eine Beschreibung dessen für das Verständnis nicht weiter notwendig ist. Die Formatierung der Binärdatei erfolgt immer so, dass Code (also die Anweisungen die abgearbeitet werden sollen) und Daten getrennt voneinander abgelegt werden.
Wird das Programm gestartet, muss es in den Hauptspeicher geladen werden. Dabei erhält das Programm einen zusätzlichen Verarbeitungsstapel, den sogenannten “Stack”, der zusätzliche für den Ablauf des Programmes notwendige Daten aufnimmt. Bei den drei Teilen eines Programms (Code, Daten, Stack) spricht man auch von Segmenten. Unter LINUX werden alle drei Segmente in separaten Speicherseiten verwaltet. Nach dem Laden des Programmes in den Hauptspeicher werden die verschiedenen Segmente durch Einträge in die Struktur “struct mm_struct”, die Bestandteil der Task-Struktur ist, verwaltet.

Bild: Segmente einer Programmdatei



Aber was genau befindet sich nun in den einzelnen Segmenten? Nun, im Code-Segment sind natürlich die Verarbeitungsbefehle für den Prozessor untergebracht. Hier steht wie und was in welche Register geladen wird und wie etwas zu bearbeiten ist. Bei den beiden anderen Hauptsegmenten ist der Fall nicht so klar, es bedarf also etwas mehr Erklärung.
In C++ gibt es drei fundamentale Möglichkeiten Speicherplatz für Daten zu verwenden.

- Statischer Speicherplatz, wo globale Variablen, Variablen von Namensbereichen und statische Variablen im statischen Speicherbereich angelegt werden. Das sind allesamt Variablen die einmal erzeugt und bis zum Ende des Programmes erhalten bleiben. Sie haben während der Laufzeit des Programms immer die gleiche Adresse. Statische Variablen werden bei der Deklaration mit dem vorangestellten Schlüsselwort “static” erzeugt. Diese Variablen befinden sich im Datensegment.
- Automatischer Speicherplatz, in den Funktionsargumente und lokale Variablen angelegt werden. Jeder Einstieg in eine Funktion oder einen Block bekommt seine eigene Kopie dieser Daten. Dieser Speicherplatz wird automatisch erzeugt und auch wieder zerstört. Man bezeichnet automatischen Speicherplatz auch als im “Stack” sein. D.h. alle Variablen die bisher in diesen Script in den Beispielen deklariert und definiert wurden, befinden sich im “Stack-Segment”. Eigentlich kann man als Faustregel gelten lassen wenn man sagt, dass alle die Variablen die auf lokaler Ebene, ob nun in Klassen, Funktionen oder Blöcken, ohne zusätzliches Schlüsselwort

vereinbart sind im Stack angelegt werden.
- Freispeicher, der aus dem Speicherplatz für Variablen explizit mit “new” angefordert werden kann und mit “delete”, wenn der Speichrplatz nicht mehr benötigt wird, an das System zurückgegeben werden kann. Überall wo im Vorfeld noch nicht genau bekannt ist wieviel Speicher wirklich benötigt wird greift man auf diese Möglichkeit zurück. Dieser vom Betriebssystem angeforderte Speicher wird auch als “Heap”, was soviel wie Halde bedeutet, bezeichnet. An dieser Stelle eine kleine Anmerkung: Es ist völlig egal ob es sich um einen fundamentalen Datentyp oder ein Objekt handelt. Es kann immer dafür Speicher auf dem Heap angefordert werden. Die einzige Beschränkung die es dabei gibt ist die Größe des Freispeichers selbst und nichts anderes. Ich sage das an dieser Stelle weil es in einigen Foren, z.B. dem deutschsprachigen QT-Forum, recht merkwürdige, wenn nicht sogar gefährliche Aussagen dazu gibt!

Nun haben wir so einiges über die Speicherverwaltung kennengelernt, so dass wir zumindest mit den Begriffen “Stack” oder “Heap” etwas anfangen können. Wobei ich zu letzteren doch noch einwenig sagen möchte.
Wie schon mehrfach gesagt kann mittels dem Operator “new” explizit Speicherplatz für eine Variable eines bestimmten Typs auf dem sogenannten “Heap” angefordert werden. Als Operand hinter new wird einfach der Typ der Variablen angegeben, für die explizit Speicher angelegt werden soll. Zurückgeliefert wird ein Zeiger auf die Variable. Sofern es sich bei dem Typ um eine Klasse handelt und ein entsprechender Konstruktor definiert ist, wird dieser aufgerufen. Allerdings kann man auf ähnliche Art und Weise auch fundamentale Datentypen initialisieren. Auch wenn für diese Datentypen keine Klassen existieren haben diese zumindest Default-Konstruktoren
Beispiel 11:

```
/* Speicherplatz im Freispeicher anfordern */

double* dp = new double; // einfacher Datentyp
char* cp = new char('a'); // einfacher Datentyp mit Initialisierung
TestKlasse* tkp = new TestKlasse; // default Konstruktor
TestKlasse* tkp = new TestKlasse(“TestName”); // ruft Konstruktor TestKlasse(const char* name)
```

Falls new keinen Speicher anfordern (allokieren) kann, wird von new die Konstante “NULL”, die wir ja schon kennen, zurückgegeben. Deshalb sollte ein Aufruf von new immer auf “NULL” getestet werden. Es gibt allerdings eine Ausnahme. Falls ein sogenannter “New-Händler”, eine spezielle Funktion welche die Fehler beim Aufruf von new behandelt, installiert ist, kann diese Abfrage entfallen. Es ist aber nicht verboten es dennoch zu tun!

Beispiel 12:

```
/* Speicher anfordern mit Test auf NULL */

TestKlasse* tkp = new TestKlasse(“TestName”);
if( tkp == NULL ){
	// Fehlerbehandlung : new konnte kein Speicher anfordern
}
```

Durch Verwendung der eckigen Klammern “[]” können auch Felder (Arrays) auf dem Heap angelegt werden.

Werden Felder mit new angelegt, wird ein Zeiger auf das erste Element zurückgegeben. Sollte dabei nicht genug Speicherplatz zur Verfügung stehen wird, wie schon bei einer einzelnen Variable, NULL zurückgegeben. Auch hier ist, wenn kein New-Händler installiert, der Zeiger abzuprüfen.
Beispiel 13:

/* Anlegen von Feldern */

```
char* s = new char [len + 1]; // Zeichenkette der Länge len+1
TestKlasse* tkp = new TestKlasse [20]; // Array mit 20 TestKlassen
float** value = new float* [10] // Array mit 10 Zeigern auf float
```

Um den mit new angeforderten Speicher wieder freizugeben muss der Operator “delete” verwendet werden. Dabei muss der von new zurückgelieferte Zeiger als Operand übergeben werden. Es ist besonders wichtig darauf hinzuweisen, dass ein Aufruf von delete auf ein nicht mit new angeforderten Speicherplatz fatale Folgen haben kann. Denn so ein Aufruf ist in der Regel undefiniert. Insbesondere die Freigabe von Speicher, welcher mit “malloc()” oder “realloc()” angefordert wurde, ist falsch. Der Operator “delete” kennt dabei nicht unbedingt die korrekte Größe des freizugebenden Speichers. Zwar kann man Glück haben, dass intern die Implementierung mittels “malloc()” bzw. “free()” durchgeführt wurde, doch ist so ein Programm nicht mehr portabel. Ein “delete” kann auf einen NULL-Zeiger angewendet werden. Das hat zwar keinen Effekt, ist aber dennoch erlaubt. Für die Freigabe von mit “new” angelegten Feldern gibt es für “delete”, ähnlich wie bei “new”, eine eigene Syntax. Diese verlangt ebenfalls die Verwendung von eckigen Klammern”[]”. Ein Programmierer ist hier wiederum selbst dafür verantwortlich, ob ein Zeiger auf ein einzelnes oder Feld von irgendetwas zeigt. Eine fehlerhafter Aufruf von “delete” führt zu undefinierten Zuständen im Programm. Der Grund dafür sollte jeden klar sein der den Abschnitt über Zeiger und Felder gelesen hat. Der Compiler selbst jedenfalls kann nicht entscheiden ob nun ein Feld oder ein einzelner Zeiger zerstört werden soll.

Beispiel 14:

```
/* Freigabe von Speicher */

void killMem( TestKlasse* tkp )
{
        // was ist nun gemeint???
        delete tkp; // bei tkp = new TestKlasse
        delete [] tkp; // bei tkp = new TestKlasse [10]
}
```

Nachdem wir die beiden Operatoren für die dynamische Speicherverwaltung kennengelernt haben, sollte zumindest Klarheit darüber bestehen, dass es immer eine sichere Strategie bedarf den angeforderten Speicher, wenn er während der Laufzeit des Programmes nicht mehr benötigt wird, wieder freizugeben. Sichergestellt sollte ebenfalls sein, dass auf bereits zerstörte also nicht mehr gültige Speicherbereiche zugegriffen wird. Um den Umgang mit dem Freispeicher für den Programmierer zu erleichtern gibt es einige, zum Teil vielversprechende, Ansätze. In einigen Bibliotheken findet man einen Ansatz einer Speicherverwaltung die Variablen findet auf die nicht mehr verwiesen wird und deren Speicherplatz freigibt und erneut diesen anderen Variablen zur Verfügung stellt. Dieses wird üblicher Weise als Garbage-Collection bezeichnet. Auf eines möchte ich an dieser Stelle hinweisen. In der, ab der Version 4.0 besser die, Qt-Bibliothek(en) gibt es solche

Garbage-Collection nicht! Auch wenn es einige Schlaumeier behaupten. An dieser Stelle mal ein kleiner Vorgriff auf eine Eigenschaft der Qt-Bibliothek. Das sogenannte “Qt Object Model”, das eine ganze Reihe von Erweiterungen des Standard C++ Objektmodels enthält. Es sind solche Dinge wie das Signal/Slot-System, welches ein Kommunikationssystem zwischen Objekten darstellt und vieles mehr. Interessant für uns an dieser Stelle ist die Verwaltung von Objekten in hierarchischen Strukturen, den sogenannten “Object tree's”. D.h. alle von der “QObject-Klasse” abgeleiteten Kassen, und damit alle konkreten Objekte davon, verwalten sich selbst nebst ihren Nachfahren jeweils in solchen Bäumen. Da in der Qt-Bibliothek alle Oberflächenelemente “QObject” als Basisklasse (Begriff klären wir noch!) haben, werden sie bei ihrer Verwendung so verwaltet. Nehmen wir an wir erzeugen mit Qt ein einfaches Fenster und darin weitere Elemente wie Schaltflächen, Labels oder Eingabefelder, dann gehören sie alle zu der Hierarchie des Fenster-Objektes. Wird dieses einfache Fenster gelöscht, werden auch alle anderen darin befindlichen Kinder-Objekte gelöscht. Das ist sicherlich sehr praktisch, doch mit einen generellen Speichermanagement wie “Garbage Collection” hat das nicht viel zu tun.
Auch wenn es zu vielen Dingen noch etwas zu sagen gäbe.
Die wichtigsten allgemeinen Grundlagen von C++ sollten jetzt aber geklärt sein, so dass wir uns einen neuen Thema zuwenden können, nämlich der objektorientierten Programmierung, kurz OOP.

2.3 Objektorientierte Programmierung mit C++

Über den Begriff “objektorientierte Programmierung” ist in der Vergangenheit schon viel geschrieben bzw. geredet worden. Irgendwie hat man manchmal den Eindruck diesem Begriff haftet etwas mystisches an. Was auch kein Wunder ist, denn die Vielfalt der Begriffswelt die sich in die OOP eingebürgert hat und sich die einzelnen Begriffe fast wie ein Chamäleon von mal zu mal in ihrer Bedeutung ändern, machen es nicht leicht diese Art der Programmierung zu verstehen. Letztlich sorgt der akademische Streit darüber was nun objektorientiert ist oder nicht, unter denjenigen die sich damit beschäftigen wollen oder müssen, für einige Verwirrung. Egal ob sich die Experten und Propheten einigen oder nicht, am Ende ist die Praxis das Kriterium jedweder Theorie. Wählen wir also einen pragmatischen praktischen Ansatz für die Erklärung.
Die rasanten Veränderungen in der Computerwelt und der damit erhöhte Bedarf an bedienerfreundlicher Software führte dazu, dass neue Techniken in der Softwareentwicklung gebraucht wurden. Die Richtung für diese neuen Techniken war klar, wiederverwendbarer Code sowie möglichst einfaches Anpassen des Codes an die gestellten Probleme. Mehr und mehr setzte sich der Ansatz der OOP durch, der heute kaum noch aus der Programmentwicklung wegzudenken ist.
Objektorientiertes Programmieren ist einfach eine Programmiertechnik, also eine Verfahrensweise zum Schreiben von Programmen, für ein bestimmtes Problem. Der Begriff objektorientierte Sprache bedeutet daher, dass die Sprache Mechanismen zur Verfügung stellt, mit dessen Hilfe diese Programmiertechnik angemessen einfach, sicher und effizient angewendet werden kann.
Aber was bedeutet das im Einzelnen? Nun, neu ist die Stellung des Programmflusses auf der einen und die zu bearbeitenden Daten auf der anderen Seite. Früher lag das Hauptaugenmerk mehr auf den Algorithmen des Programmes, also der Ausführung der benötigten Berechnung. Die Algorithmen bildeten dabei aber noch keine Einheit mit den durch sie zu bearbeitenden Daten. Ein großer Schritt in der Geschichte der Programmiersprachen war die Schaffung der Möglichkeit, mehrere Daten zu einen Datenverbund zusammenzufassen. Damit können in Strukturen oder Records, welche wir schon unter Punkt 2.1 kurz besprochen haben, verschiedene zusammengehörige Eigenschaften in einen Datentyp gebündelt werden. Solche Strukturen erlauben es also, dass eine Variable alle Daten enthält, die zu dem von der Variablen repräsentierten

Sachverhalt gehören.
Dadurch wird eine Art der Abstrahierung des zu repräsentierenden Sachverhaltes in seinen benötigten Einzelheiten möglich. In der Praxis sieht es z.B. folgendermaßen aus: Eine Struktur welche eine Person beschreiben soll, also für den Datentyp "person", kann aus den Einzelelementen (Komponenten) Vorname, Nachname, Alter, Geschlecht und Beruf bestehen.

Beispiel 15:

```
/* Deklaration der Struktur "person" */

struct person{
        char* name; // Harts
        char* vorname; // Hans
        int alter; // 66
        char geschlecht; // M
        char* beruf; // Computerfuzzi
};
```

Wie wir sehen, ermöglicht eine Struktur die Beschreibung eines Sachverhaltes oder auch eines Vorganges in Programmen zu verwalten. Es sind damit Abbildungen wesentlicher Eigenschaften von realen "Objekten" aus unserer Umwelt. Ein Objekt ist "etwas", das betrachtet, verwendet oder eine Rolle spielt. Das kann so ziemlich "alles" sein, z.B. ein Fahrzeug, ein Tier oder der Vorgang der Gärung bei der Bierherstellung. Je nach Aufgabenstellung sind verschiedene Aspekte eines Objektes von Interesse. In Strukturen werden die Eigenschaften der zu betrachtenden "Objekte" in den Komponenten festgehalten. Aber wenn wir "Objekte" betrachten interessiert nicht nur wie sie aufgebaut sind oder was sie repräsentieren, sondern was man mit ihnen anfangen kann. Von Interesse sind also auch die "Operationen" die mit diesen Objekt durchgeführt werden können, z.B. eine Beschreibung welche Zustände das Objekt überhaupt annehmen kann. Bei einen Fahrzeug z.B. ist nicht nur von Interesse ob es einen Antrieb oder einen Führerstand hat, sondern auch ob und wie man damit fahren, fliegen oder schwimmen kann und wie der aktuelle Füllstand des Treibstoffes ist. In der rein "strukturierten Programmierung" wird diesem Sachverhalt nur ungenügend Rechnung getragen. Datentypen die diesem Sachverhalt genügen, bezeichnet man als "abstrakte Datentypen". Die Möglichkeit mit solchen "abstrakten Datentypen" in einer Programmiersprache arbeiten zu können, ist eine der Grundbedingungen für die objektorientierte Programmierung.
In C++ ist diese Möglichkeit gegeben, indem es dem Benutzer erlaubt ist solche Datentypen zu definieren. In der OOP hat man für die Realisierung abstrakter Datentypen den Begriff "Klasse" geprägt. Eine Klasse ist damit die Implementierung, also Darstellung in der Programmiersprache, eines abstrakten Datentyps. Diese beschreibt, im Gegensatz zu einer einfachen Struktur, neben den Einzelheiten des Objektes auch dessen Verhalten.

Beispiel 16:

```
/* Deklaration einer Klasse in C++  ( QT)*/

class FunctionDlg: public QDialog
{
   Q_OBJECT
public: // öffentliche Schnittstelle
   FunctionDlg( QWidget* parent = 0 );
```

```
    ~FunctionDlg();

    double getLowX();
    double getHighX();
    double getXStep();

    double getLowY();
    double getHighY();
    double getYStep();

    QString function();

private slots: // interne Funktion
    void acceptClicked();



private: // Datenelemente dieser Klasse – kein Zugriff von außen!
    QLabel *l;
    QString func;
    double lowX;
    double lowY;
    double highX;
    double highY;
    double step;
    QLineEdit    *funcEingabe,
                 *fromX, *fromY, *toX, *toY, *widthX;
    QPushButton  *ok_pb;

};
```

Bevor wir fortfahren noch ein paar Sätze zur Begriffswelt in der OOP. Wie schon am Anfang gesagt herrscht ein ziemliches Durcheinander in den Begriffsdefinitionen. Viele Begriffe sind mehrdeutig, für ein und denselben Sachverhalt existieren gleich mehrere Begriffe oder ein Begriff wird für die Beschreibung verschiedener Sachverhalte genutzt. Deshalb ist es an dieser Stelle notwendig sich auf bestimmte Begriffe für die einzelnen Sachverhalte festzulegen.

Klassen

Klassen sind die Beschreibung wesentlicher Einzelheiten, Eigenschaften und Verhalten von Objekten. Sie sind somit die programmiertechnische Umsetzung des Begriffes "abstrakter Datentyp". Eine Klasse in C++ ist eine Typenbeschreibung selbstdefinierter Typen. Man kann sie gewissermaßen als selbstdefinierte Baupläne für Variablen begreifen die, wenn in einen Programm erzeugt und verwendet, den Sachverhalt repräsentieren für den sie bestimmt sind. In C++ sind Klassen Struktur-Datentypen, deren Komponenten neben Daten auch Funktionen sein können und Zugriffsbeschränkungen unterliegen. Neben den "konkreten Klassen" von denen man (Objekt)-variablen erzeugen kann gibt es auch "abstrakte Klassen" die sozusagen einen Oberbegriff darstellen. Von "abstrakten Klassen" kann man keine (Objekt)variablen erzeugen, es würde auch keinen Sinn machen da mit ihnen nur ganz allgemeine Beschreibungen implementiert werden.

Objekt

Der zentrale Begriff der objektorientierten Programmierung. Ein Objekt ist ein Informationsträger, der verschiedene Daten repräsentiert, die einen bestimmten Zustand (Wertbelegung) besitzen.
Ein Objekt muss nicht zwingend etwas Greifbares sein, es kann beliebig abstrakt sein und auch Vorgänge beschreiben. Wenn man objektorientiert programmiert, versucht man die Objekte, die im Problemfeld des Programms eine Rolle spielen, zu ermitteln und zu implementieren.

Instanzen (Objektvariablen)

Während eine Klasse eine Beschreibung eines Sachverhaltes ist und damit ein Datentyp, nennt man Variablen die von diesen Datentyp erzeugt werden "Instanzen". Leider ist dieser Begriff ein wenig zur Hure geworden. Für das Verständnis ist es besser nicht diesen Begriff zu verwenden, sondern lieber "Objektvariable", "Klassenvariable" oder ganz einfach "Objekt". In C++ haben ohnehin die Begriffe "Instanz" und "Objekt", im Gegensatz zu anderen Sprachen, dieselbe Bedeutung. Übrigens in der Sprachspezifikation wird dieser Begriff gar nicht verwendet.

Methoden (Elementfunktionen)

In der objektorientierten Sprachwelt nennt man die in einer Klasse definierten Operationen auch "Methoden". Der Hintergrund für diesen Begriff ist, dass jede Operation, die für ein Objekt aufgerufen wird, als Nachricht an das Objekt interpretiert wird, welches zu deren Verarbeitung eine bestimmte Methode verwendet. Ich will um Gottes willen diese Sichtweise nicht abwerten.
Da aber C++-Klassen Struktur-Datentypen sind, ist es sinnvoller Klassenelemente mit den Oberbegriff Komponenten zu versehen die in Datenelement und Elementfunktionen unterschieden werden.

Einen Hinweis muss ich an dieser Stelle noch zu dem von mir verwendeten Begriff Komponente machen. Leute die sich schon mit visuellen Entwicklungssytemen wie z.B. den C++-Builder bzw. Delphi der Firma Borland beschäftigt haben kennen diesen Begriff schon in einen ganz anderen Zusammenhang. Dort steht der Begriff nicht nur für ein Element einer Struktur oder Klasse, sondern für eine "Klassenkategorie", "Modul" oder "Paket". Solche Komponenten werden im Idealfall durch eine Reihe von Schnittstellen beschrieben, die zu deren Implementierung verwendet werden, sowie eine Reihe von Schnittstellen, die deren Anwendern zur Verfügung gestellt wird. So eine Komponente oder Paket bietet sozusagen eine Sammlung von Klassen, Typen usw. die in sich geschlossen sind für einen definierten Zweck. Das kann z.B. ein "Komponente/Paket" für den Zugriff auf eine Datenbank sein.

Wenden wir uns aber weiter den wesentlichen Kennzeichen der objektorientierten Programmierung zu. Einige Kennzeichen der OOP, wie Klassen, haben wir schon besprochen. Es gibt aber noch weitere drei Kennzeichen die ich nicht vorenthalten kann.
Ein Problem der C-Strukturen ist, dass sie öffentlich zugänglich sind und man jederzeit auf alle Komponenten zugreifen kann. D.h. in jeden Programmteil in den eine Struktur verwendet wird, können die Komponenten einer Struktur auch geändert werden. Geschieht das an Stellen des

Programms an denen eigentlich keine Änderung vorgesehen ist, führt dies leicht zu Fehlern und Inkonsistenzen. So könnte z.B. versehentlich bei einer Liste von Personen deren Anzahl verringert werden bzw. ein einzelner Satz von Personendaten direkt manipuliert werden. Um dies zu verhindern entstand die Idee der “Datenkapselung” (engl: encapsulation).

Datenkapselung

Unter Datenkapselung versteht man, dass ein Zugriff auf ein Objekt nur über eine definierte Schnittstelle erfolgt. Es wird damit sichergestellt, dass keiner mit den Objekt machen kann was er will. Jeder Anwender, der mit einen solchen Objekt umgeht, soll nur die Operationen durchführen können, die der Designer des entsprechenden Datentyps, also der die Klasse implementiert hat, für einen öffentlichen Zugriff vorgesehen hat. Die Datenelemente sowie die rein für die Implementation benötigten Elementfunktionen einer Klasse werden vor dem Zugriff von außen geschützt. In der englischsprachigen Literatur wird dafür oftmals der Begriff “information hiding” verwendet, was soviel bedeutet wie das “verbergen von Informationen”. Auf C++ bezogen ist dies aber nicht korrekt, da die internen Daten nur vor den Zugriff von außen gesperrt, aber auf keine Weise verborgen werden. In C++, wie auch in vielen anderen Sprachen, kann der Zugriff auf die Komponenten einer Klasse oder Struktur über spezielle Zugriffsschlüsselwörter definiert werden. Ohne die Verwendung dieser Zugriffsschlüsselwörter, also die “default”-Einstellung, ist folgendes zu beachten:

- die Komponenten einer einfachen Struktur sind alle öffentlich zugänglich
- die Komponenten einer Klasse hingegen sind alle “private”, also nicht öffentlich zugänglich

Es gibt im Standard drei Schlüsselwörter um den Zugriff, auf Elementfunktionen und Datenelemente, von außen zu steuern.

public: Die nach diesen Schlüsselwort deklarierten Elementfunktionen oder Datenelemente sind außerhalb der Klasse öffentlich zugänglich. Typischer Weise sind unter diesen Schlüsselwort die Elementfunktionen deklariert die die öffentliche Schnittstelle bilden. Da der Zugriff auf die Datenelemente tunlichst vermieden werden soll, sind hier in der Regel nur Elementfunktionen deklariert die den Zugriff von außen ermöglichen.

private: Alle nach diesen Schlüsselwort deklarierten Komponenten sind außerhalb der Klassen-Implementierung nicht zugänglich. D.h. diese Komponenten können nicht von einen “Anwender”, in diesen Fall der Programmierer der diese Klasse für sein Programm nutzt, aufgerufen werden. Neben den Datenelementen befinden sich hier in der Regel Elementfunktionen, die u.a. Hilfsfunktionen für die Klassen-Implementation sind.

protected: Dieses Schlüsselwort hat, erst mal für diese Betrachtung, die selbe Schutzfunktion wie “private”. Der eigentliche Unterschied ist erst bei der Vererbung von Klassen relevant.

Diese Schlüsselworte für die einzelnen Schutzklassen können überall in der Klassendeklaration stehen und auch u.U. dort mehrfach genutzt werden. Auch wenn das Schlüsselwort “private” in Klassen nicht explizit angegeben werden muss, sollte man es wegen der Übersichtlichkeit dennoch tun.
Neben dem Schutz der Datenelemente vor versehentlicher Veränderung gibt es noch einen weiteren Aspekt der für das Konzept der Datenkapselung spricht. Es steht weniger im Vordergrund wie

etwas implementiert ist, sondern was man mit Objekten des jeweiligen Typs anfangen kann. Implementationsdetails sind dabei zunächst irrelevant. Es ist nur wichtig was man mit den Objekten anfangen kann. Das beste Beispiel hierzu ist die Standard-Bibliothek von C++. Es ist nirgends vorgeschrieben wie etwas zu implementieren ist, sondern nur was bestimmte Objekte leisten und anbieten müssen. Allerdings ist es in der Praxis oftmals dennoch von Nöten sich bestimmte Details der Implementierung genauer anzusehen. Es kann durchaus sein, dass der Einsatz von Objekten eines bestimmten Typs von seiner Implementierung abhängt.

Mit den bisherigen Wissen sind wir in der Lage, die wesentlichen Eigenschaften realer Objekte in Klassen abzubilden. Auch können wir mit den Zugriffsbeschränkungen dafür Sorge tragen, dass nur über definierte öffentliche Elementfunktionen auf die Daten dieser Klasse zugegriffen werden kann. Nehmen wir einmal an, wir haben eine Klasse geschrieben die wir schon erfolgreich in einigen von unseren Programmen einsetzen und die auch gut funktioniert. Aber was ist, wenn die Eigenschaften die wir in diese Klasse implementiert haben erweitert werden müssen?
Nehmen wir mal ein alltägliches Beispiel. Fotografieren tun wir alle mehr oder weniger oft und gerne. Betrachten wir einmal das Objekt „Kamera“ genauer. Eine Kamera besitzt zumindest ein Objektiv, ein Auslöser und ggf. ein eingebautes Blitzlicht. Das Objektiv sorgt dafür, dass unser Motiv scharf aufgenommen wird. Nach betätigen des Auslösers wird ein Bild aufgenommen.
Für eine allgemeine Betrachtung des Fotografierens reichen diese Eigenschaften aus. Aber wenn man den Blickpunkt auf die Speicherung der aufgenommenen Bilder setzt, müssen weitere Eigenschaften in die Betrachtung einbezogen werden. Dabei ist es nicht unerheblich, ob die aufgenommenen Bilder analog auf einen Film oder digital auf einen Chip abgelegt werden. D.h. eine bestehende Klasse „Kamera“ müsste spezifiziert werden. Man könnte natürlich versuchen alle möglichen Eigenschaften in eine Klasse „Kamera“ einzubauen oder für jede einzelne Kamera eine eigene Klasse implementieren. Ein eleganterer Weg wäre die Möglichkeit Sprachmittel zur Verfügung zu haben, die es erlauben Eigenschaften aus bereits existierenden Klassen weiter zu nutzen. In der objektorientierten Programmierung gibt es dafür das Konzept der Vererbung, das auch in C++ zur Verfügung steht. Dieses konzept bietet die Möglichkeit hierarchische Beziehungen zwischen Klassen auszudrücken.

Vererbung

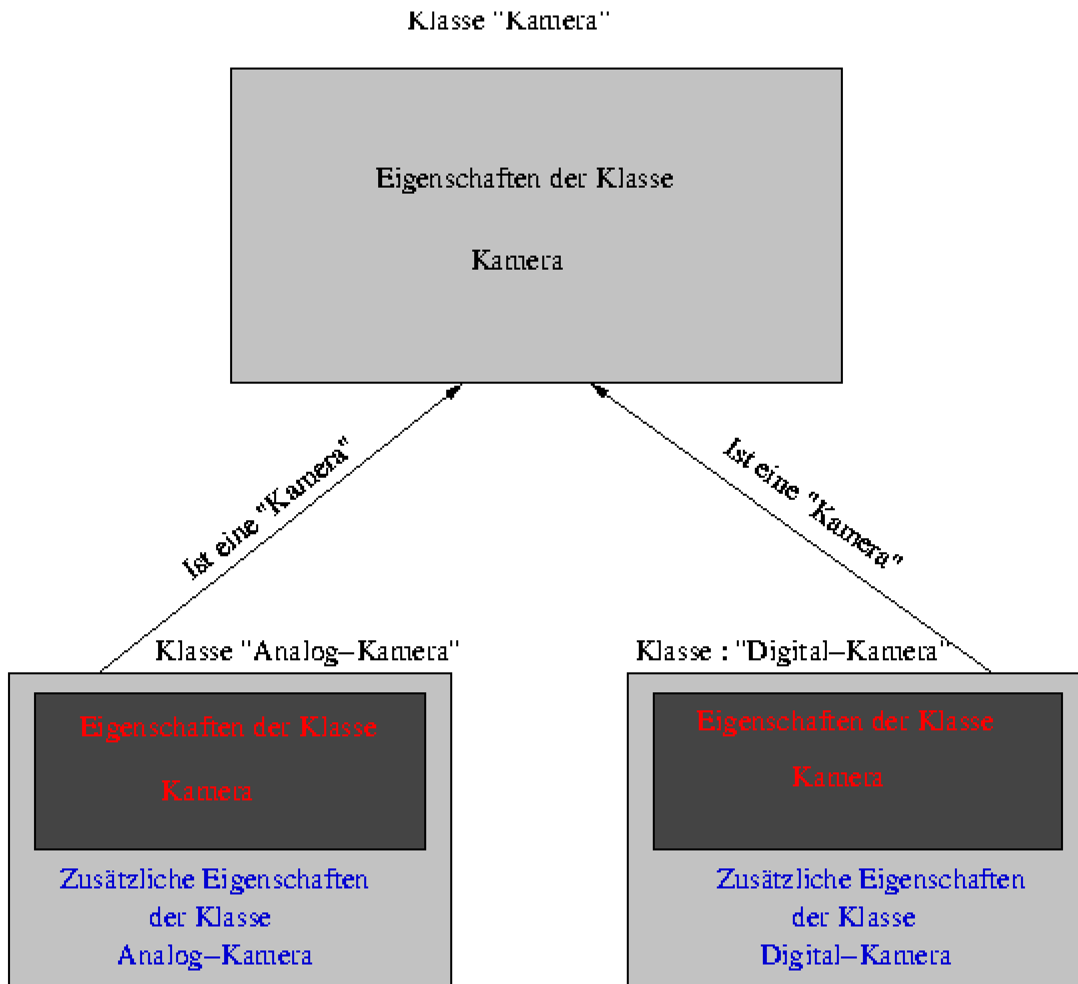
Wenn man das Verhalten von Objekten beschreibt gibt es oft Eigenschaften die verschiedenartige Objekte gemeinsam haben. Im natürlichen Sprachgebrauch werden dafür Oberbegriffe verwendet. Aus einer “Canon 350D” mit Zoom-Objektiv und einer 1 GB-Speicherkarte wird einfach zu einer “Kamera”, solange die Details nicht von irgendein Interesse sind. So eine Verallgemeinerung findet sich z.B. bei einer Fototasche. Für diese ist es unerheblich was für eine Kamera darin transportiert wird. Bestenfalls ihre Maße sind von Interesse. Dennoch können in einer Fototasche die unterschiedlichsten Kameras transportiert werden. Nach der Entnahme aus der Tasche können jedoch die speziellen Eigenschaften wieder von Interesse sein. D.h. je nach Bedarf kann der Oberbegriff oder dessen Spezialisierung verwendet werden.
Für dieses Abstraktionsmittel von “Generalisierung (Oberbegriff)” und “Spezialisierung” gibt es in den herkömmlichen Sprachen kein äquivalentes Datenmodell.

- **Generalisierung:** Eine Oberklasse ist die Generalisierung der Unterklasse.
- **Spezialisierung:** Eine Unterklasse ist die Spezialisierung der Oberklasse.

Damit können verschiedene Klassen in einer hierarchischen Beziehung zueinander stehen. Eine “abgeleitete” Klasse ist immer eine Spezialisierung ihrer “Basisklasse”. Umgekehrt ist die

Basisklasse die Generalisierung der abgeleiteten Klasse. Das bedeutet, dass alle Eigenschaften (Datenelemente und Elementfunktionen) der Basisklasse übernommen (geerbt) und in der Regel durch genauere Eigenschaften ergänzt werden. Für den Begriff "Basisklasse" existieren auch noch alternative Bezeichnungen, z.B. "Elternklasse". Auch für abgeleitete Klassen gibt es alternative Benennungen, z.B Kindklasse.

Bild : Konzept der Vererbung (Ist – Beziehung)



Der praktische Nutzen ist dabei, dass bei Beibehaltung der Eigenschaften nur noch Neuerungen implementiert werden müssen. Dies führt neben einer Einsparung von Code zu einen Konsistenzvorteil, da gemeinsame Eigenschaften nicht an mehreren Stellen stehen und somit nur einmal geprüft bzw. geändert werden müssen.
In C++ gibt es neben der einfachen Vererbung, d.h. eine abgeleitete Klasse kann nur eine Basisklasse besitzen, die Mehrfachvererbung. Bei der Mehrfachvererbung sind mehrere Basisklassen möglich. Aber wie wird eine Vererbung mit den Sprachmitteln von C++ realisiert? Für die einfache Vererbung gilt folgende Syntax :

```
class Klassenname : Schutztyp Basisklassenname
{
        // Datenelemente und Elementfunktionen
}
```

Für unser Beispiel mit der Kamera würde das wie folgt aussehen:

Beispiel 17:

```
/* Beispiel einer Vererbung */

#include<objektiv.h>
#include<libtypes.h>

/* Deklaration einer Basisklasse */
class Kamera {  // Basisklasse

public:
	Kamera( Objektiv* obj = 0 ); // Konstruktor

	// Funktionalität die jede Kamera hat!
	void scharfstellen(bool* ok);
	bool auslösen();
	void blitz_an_aus( const bool &flag );
private:
	Objektiv* o;
};
/* abgeleitete Klasse */
class Analog_Kamera : public Kamera { // abgeleitete Klasse

public:
	Analog_Kamera( Objektiv* obj , LibTypes::FilmTyp ft ); //Konstruktor

	// zusätzliche Funktionalität für analog
	void film_einlegen();
	bool istFilmVoll();
	int anzahlVerknipsterBilder();
private:
	int filmlänge;

};
/* abgeleitete Klasse */
class Digital_Kamera : public Kamera { // abgeleitete Klasse

public:
	Digital_Kamera( Objektiv* obj, LibTypes::MemType mt ); // Konstruktor

	// zusätzliche Funktionalität für digital
	void auflösung_einstellen( const int &x, const int &y );
	int restSpeicherPlatz();
private:
	int speicher;

};
```

Der bei der Vererbung angegebene “Schutztyp” gibt an, wie auf die von der Basisklasse geerbten Datenelemente und den Elementfunktionen zugegriffen werden darf. Hierbei existieren folgende drei Regeln:

1. Auf privat-Komponenten kann grundsätzlich nicht zugegriffen werden, unabhängig davon, welcher Schutztyp für die Basisklasse in der abgeleiteten Klasse angegeben wird.
2. Ist private für die Basisklasse in der abgeleiteten Klasse angegeben, können die Elementfunktionen der abgeleiteten Klassen auf die geerbten public- und protected-Komponenten der Basisklasse zugreifen. Das gilt jedoch nicht für weitere Unterklassen, die durch weitere Vererbung der abgeleiteten Klasse entstehen. Die public-und protected-Komponenten der Basisklasse sind somit private-Komponenten in der direkt abgeleiteten Kindklasse.
3. Wird public für die Basisklasse in der abgeleiteten Klasse angegeben, so können die Elementfunktionen der abgeleiteten Klasse auf alle public- und protected-Komponenten der Basisklasse zugreifen. Dieses Zugriffsrecht wird auch an eventuelle weitere Unterklassen der abgeleiteten Klasse vererbt.

Nun wird auch deutlich, warum das Schlüsselwort “protected” eingeführt wurde. Ohne dieses Schlüsselwort müsste man alle Komponenten einer Basisklasse, die weitervererbt werden sollen, als “public” einstufen. Die Konsequenz daraus würde bedeuten, dass auch Elementfunktionen oder Datenelemente, die eigentlich nicht zur öffentlichen Schnittstelle zählen, den Zugriff durch alle anderen Funktionen des Programms zu ermöglichen. Damit wäre das Konzept der Datenkapselung aufgehoben. Um einen solchen Zustand zu vermeiden wurde eine Zwischenstufe zwischen den beiden extremen Schutztypen “private” und “public” eingeführt, nämlich der Schutztyp “protected”. Bleibt aber noch die Frage was eigentlich mit den Konstruktoren bzw. Destruktoren bei der Vererbung passiert. Die Konstruktoren werden, wie jede andere Elementfunktion, ebenfalls vererbt. Allerdings werden sie innerhalb der Unterklasse nicht automatisch verwendet. Das heißt, ohne eine Neudefinition von Konstruktoren existiert innerhalb einer Unterklasse vorerst nur der Standardkonstruktor (Default-Konstruktor) dieser Klasse. Wir erinnern uns, der Standard-konstruktor, auch Default-Konstruktor genannt, ist der automatisch mit jeder Klassendefinition bereitgestellte Konstruktor. Wird dann dieser Standardkonstruktor aufgerufen, werden zunächst die Standardkonstruktoren sämtlicher Basisklassen aufgerufen. Klassenobjekte werden von unten nach oben konstruiert. Zuerst die Basisklasse dann die Komponenten und schließlich die abgeleitete Klasse selbst. Komponenten und Basisklassen werden in der Reihenfolge ihrer Deklaration in der Klasse konstruiert und in der umgekehrten Reihenfolge zerstört. Wird in der Unterklasse ein Konstruktor explizit definiert, kann dort auch ein bestimmter Konstruktor der Basisklasse aufgerufen werden. In dieser Hinsicht stellt sich eine Basisklasse genau wie eine Komponente der abgeleiteten Klasse dar. Der Konstruktor einer abgeleiteten Klasse kann nur Initialisierungen für seine eigenen Komponenten und seine unmittelbaren Basisklassen angeben. Er kann nicht direkt Komponenten einer Basisklasse initialisieren.

Beispiel 18:

```
Digital_Kamera::Digital_Kamera(  Objektiv* obj, LibTypes::MemType mt )
        : Kamera( obj ) // Aufruf des Konstruktors der Basisklasse
{
        // ........
}
```

Das Beispiel zeigt wie ein expliziter Aufruf eines Konstruktors einer Basisklasse erfolgen kann. Dazu dient nämlich die in C++ eingeführte “Initialisierungsliste”. Eine Initialisierungsliste kann nur bei der Definition eines Konstruktors verwendet werden. Sie dient dazu, Basisklassen und Datenelemente der Klasse Parameter zur deren Initialisierung zu übergeben. Sie wird bei der Definition des Konstruktors hinter dessen Parameterliste durch einen Doppelpunkt getrennt angegeben und besteht aus durch Kommata getrennten Ausdrücken der Form :

Komponente(Argumente)

Es ist durchaus anzuraten, gerade bei Datenelementen einer Klasse, Initialisierungslisten zu verwenden. Sie sind meist effektiver als die Alternative der Zuweisung, da sie meist weniger Zeit zur Konstruktion eines Objektes brauchen.

Für unser Beispiel mit der Kamera mag die bisherige Betrachtung völlig ausreichend sein. Aber oftmals finden sich Beispiele für Oberbegriffe in der Praxis, deren verallgemeinerten Eigenschaften in den Spezialisierungen unterschiede aufweisen. Nehmen wir zum Beispiel eine Kugel, einen Würfel und einen Kegel. Es fällt uns nicht schwer diese drei Dinge mit den Oberbegriff “Körper” zu versehen. Alle dieser Körper haben eine bestimmte Eigenschaft, nämlich sie haben alle ein Volumen. D.h. eine Basisklasse “Körper” könnte wie folgt aussehen :

Beispiel 19:

```
/* Deklaration der Klasse “Körper” */

class Koerper{

public:

        Koerper( );
        int volumen() const; // berechnet das Volumendes Körpers
        .... // weitere öffentliche Elementfunktionen
private:
        .... // Datenelemente
};
```

Nun ist es aber so, dass die Berechnung der Volumen der einzelnen Körper unterschiedlich erfolgt. Es wird zwar im Prinzip dasselbe gemacht, aber je nach Körper auf unterschiedliche Art. Um diesen Sachverhalt wiederzuspiegeln, kann man für jede Klasse, die irgendeine Art von Körper beschreibt, eine eigene Version der Volumenberechnung implementieren. Allerdings muß die Elementfunktion der Basisklasse zur Berechnung des Volumens als “virtuell” deklariert werden.
Beim Arbeiten mit virtuellen Elementfunktionen sind folgende Punkte zu beachten:

1. Wenn in einer Klasse eine Elementfunktion als “virtual” deklariert wurde, müssen in allen abgeleiteten Unterklassen die gleichnamigen Elementfunktionen ebenfalls als “virtual” deklariert werden. Die Elementfunktionen müssen dabei in Funktionsname, Anzahl der Parameter, Datentyp der Parameter und Datentyp des Rückgabewertes identisch sein.
2. Wird in einer Unterklasse eine virtuelle Elementfunktion nicht explizit neu definiert, so wird die entsprechende, in der Hierarchie zuletzt definierte Elementfunktion aufgerufen.
3. Deshalb muss eine virtuelle Elementfunktion in der Klasse definiert werden, in welcher sie das erste mal deklariert wurde.

Beispiel 20:

```
/* Deklaration der Klasse “Körper” mit virtuelle Elementfunktion */

class Koerper{

public:

        Koerper( );
        virtual int volumen() const; // berechnet das Volumendes Körpers
        .... // weitere öffentliche Elementfunktionen
private:
        .... // Datenelemente
};
```

Diesen Sachverhalt bezeichnet man auch als “Polymorphie”, was soviel wie “Vielgestaltigkeit” bedeutet.

Polymorphie

Virtuelle Elementfunktionen die in der Basisklasse deklariert sind, können in der abgeleiteten Klasse redefiniert werden. Die Redefinition wird oftmals auch “Überschreiben” genannt. Der Compiler und Linker garanieren die korrekte Zuordnung zwischen Objekten und die auf sie angewandten Elementfunktionen. Bei der Redefinition ist allerdings zu beachten, dass die Anzahl und Typen der Funktionsargumente identisch sein müssen!
Polymorphie beschreibt damit die Fähigkeit, dass eine Operation für verschiedene Objekte aufgerufen werden kann und dabei zu unterschiedlichen Auswirkungen führt. Ein Typ mit virtuellen Elementfunktionen wird “polymorpher Typ” genannt. Polymorphie erlaubt, dass verschiedenartige Objekte zeitweise unter einen gemeinsamen Oberbegriff verwendet und manipuliert werden können. Polymorphie setzt die sogenannte “späte Bindung” voraus.
Da fällt mir ein, dass ich noch zwei Begriffe klären muß:

- **frühe Bindung:** Beim Aufruf einer nicht virtuellen Elementfunktion ist zum Zeitpunkt der Übersetzung die Adresse der Funktion bekannt. Diese Adresse wird direkt in den Maschinencode eingefügt. Wird eine virtuelle Elementfunktion über einen Objektnamen aufgerufen, so ist ebenfalls zur Übersetzungszeit klar, welche Version der Elementfunktion gemeint ist. Auch hier handelt es sich um eine “frühe Bindung”.
- **späte Bindung:** Beim Aufruf einer virtuellen Elementfunktion über einen Zeiger oder eine Referenz steht dagegen zur Zeit der Übersetzung noch nicht fest, welche Elementfunktion zur Laufzeit ausgeführt werden soll. Der Compiler muß daher Maschinencode erzeugen, bei dem erst zur Laufzeit die Zuordnung zu einer bestimmten Elementfunktion stattfindet. Realisiert wird das intern mit Hilfe sogenannter “virtuellen Methodentabellen”, die für jede Klasse mit mindestens einer virtuellen Elementfunktion vom Compiler angelegt wird. In dieser VMT werden die Startadressen der virtuellen Elementfunktionen gespeichert. Jedes Objekt das aus einer polymorphen Klasse erzeugt wird besitzt einen VMT-Zeiger, also einen bei der Übersetzung automatisch generierten Zeiger auf die VMT. Über diesen Zeiger wird dann zur Laufzeit die Startadresse der aufzurufenden Elementfunktion ermittelt. Die späte Bindung ermöglicht es, bereits übersetzten Quellencode nachträglich zu erweitern, ohne dass der Quellecode selbst vorhanden sein muß.

In Klassenhirarchien existieren oftmals Basisklassen von denen es keinen Sinn macht konkrete Objekte zu erzeugen. Nehmen wir unsere Klasse “Körper” als Beispiel, macht es außerdem kaum einen Sinn die Elementfunktion “volumen()”, die das Volumen eines Körpers berechnen soll, zu definieren. Der Begriff “Körper” ist zu allgemein um eine Volumenberechnung durchführen zu können. Für solche Fälle gibt es in C++ die Möglichkeit auf eine Definition, also die Implementation der Berechnung, zu verzichten. Statt einer Implementierung wird der virtuellen Elementfunktion der Wert 0 zugewiesen. D.h. diese Elementfunktion ist zwar deklariert doch ihre Definition ist offengelassen worden und muß von den abgeleiteten Klassen übernommen werden. Solche Elementfunktionen werden “rein (pure) virtuelle Funktionen” genannt. Rein virtuelle Funktionen legen fest, dass für eine Klasse, die als Basisklasse dient, eine bestimmte Funktion aufrufbar ist obwohl sie noch nicht definiert wurde. Einen weiteren Pluspunkt ist die Tatsache, dass solange eine Klasse rein virtuelle Funktionen besitzt keine Objekte von ihr erzeugt werden können. Damit wird diese Klasse automatisch zur “abstrakten Klasse”. Dies gilt auch für davon abgeleiteten Klassen, sofern die rein virtuellen Elementfunktionen nicht überschrieben werden.

Nun, denke ich, sollten die wichtigsten Begriffe und Verfahrensweisen der Programmiersprache C++ geklärt sein.