C++

Werner Benger

werner@ast1.uibk.ac.at
http://math1.uibk.ac.at/~werner/

10. Juni 1997

Inhaltsverzeichnis

1 Einleitung **4**
1.1 Die Ahnen von C++ . . . 4
1.2 Die Philosophie von C++ . . . 4
1.3 Die Stufen zur objektorientierten Programmierung . . . 5
1.3.1 Prozedurale Programmierung . . . 5
1.3.2 Modulare Programmierung . . . 5
1.3.3 Datenabstraktion . . . 5
1.3.4 Objektorientierte Programmierung . . . 7
1.4 C++ als Programmiersprache . . . 8
1.5 Von C nach C++ . . . 9
1.6 Erweiterungen zu C . . . 9
1.6.1 Typkonversionen als Funktionsaufruf . . . 9
1.6.2 Variablen-/Funktionsnamen dürfen beliebig lang sein . . . 9
1.6.3 Zeilenweise Kommentare . . . 9
1.6.4 Funktionen zur Initialisierung statischer Variablen . . . 9
1.6.5 Strukturdefinitionen sind automatisch ein `typedef` . . . 10
1.6.6 Dynamischer Speicher . . . 10
1.6.7 Die Deklaration ist eine Anweisung . . . 10
1.6.8 Der Bereichsoperator . . . 11
1.6.9 Der Defaultparameter . . . 11
1.6.10 Polymorphie . . . 11
1.6.11 `inline`-Funktionen . . . 12
1.6.12 Referenzen . . . 12
1.6.13 bool - Typ . . . 13

2 Elementfunktionen - Begriff des Objektes **14**
2.1 Konstruktor/Destruktor . . . 18
2.1.1 Der Defaultkonstruktor . . . 19
2.1.2 Der Copy-Konstruktor . . . 19
2.1.3 Konstruktor-Syntax . . . 20
2.1.4 Der Destruktor . . . 20
2.1.5 Dynamische Speicherobjekte . . . 21

3 Schutzmechanismen - Begriff der Klasse **22**
3.1 `friend`-Funktionen . . . 23
3.2 Statische Klassenelemente . . . 24
3.3 Programmieraufgabe: Doppelt verkettete Listen . . . 24
3.4 Zusammenfassung und Begriffsbestimmungen: . . . 28

4 Datenabstraktion - Überladen von Operatoren **29**
4.1 Die Klasse `vector3` . . . 38
4.1.1 Rechenregeln mit Vektoren im $\mathbb{R}^3$ . . . 38
4.2 Das Raytracing - Verfahren . . . 39
4.2.1 Bestimmung der Bildschirmvektoren . . . 41
4.2.2 Berechnung des Lichteinfalls . . . 41

5 Vererbung, virtuelle Funktionen und abstrakte Klassen **42**
5.1 Ableiten von Klassen - Vererbung . . . 45
5.2 Virtuelle Funktionen . . . 47
5.3 Abstrakte Klassen . . . 50
5.4 Virtuelle Destruktoren . . . 51
5.5 Anhang . . . 52

6 Templates **53**
6.1 Templatefunktionen 53
6.2 Templateklassen 55
6.3 Übungsaufgabe 56

7 Weiteres ... **57**
7.1 Überladen der Operatoren new und delete 57
7.2 Stream-Manipulatoren 57
7.3 Virtuelle Klassen 58
7.4 Zeiger auf Klassenelemente 59
7.5 Fehlerbehandlung (exceptions) 60
7.6 mutable - Datenelemente 61
7.7 Runtime-Type-Checking 62
7.8 Namespaces und New Style Headers 62
7.9 Ausblick 63

1 Einleitung

TEILWEISE AUS: *Bjarne Stroustrup, „The* C++ *Programming Language", Second Edition, AT&T Bell Telephone Laboratories, Verlag Addison-Wesley 1991*

1.1 Die Ahnen von C++

C++ baut im wesentlichen auf C auf. C ist eine Untermenge von C++ und verleiht C++ damit die gleichen Fähigkeiten an system- und hardwarenaher Programmierung. Die andere Hauptquelle war Simula67, das Klassenkonzept stammt von dort. Weitere Inspiration kam von Algol68 (Überladen von Operatoren und Deklarationen an nahezu beliebiger Stelle). Die frühesten Sprachversionen waren als „C mit Klassen" seit ca. 1980 im Gebrauch, C++ tauchte erstmals im Juli 1983 auf.

Ursprünglich wurde C++ von Bjarne Stroustrup von AT&T für einige spezielle Anwendungen entwickelt, es gab niemals ein „C++ Projekt" oder ein „C++ Design Komitee". Mit den zu lösenden Problemen hat sich die Sprache entwickelt, auch Mitarbeiter von zu AT&T konkurrierenden Firmen halfen dabei mit - wobei es ein leichtes gewesen wäre, die Entwicklung einfach durch Stillschweigen zum Stehen oder gar zum Scheitern zu bringen.

Der Name „C++ " stammt aus dem Sommer 1983 und ist bezeichnend für die aufbauende Struktur von C++ , denn „++" ist der Inkrementoperator aus C. Eigentlich sollte die Sprache ++C heißen, denn die Sprache wurde ja zuerst erhöht und dann angewandt und nicht umgekehrt. Diese Version hat sich jedoch nicht durchgesetzt. Die Sprache wurde nicht D genannt, da sie eine **Erweiterung** von C darstellt und nicht versucht, manche Probleme dadurch zu vermeiden oder zu lösen, daß einige Features aus C abgeschafft werden.

Viele Probleme resultieren aus der Erbschaft „C", aber dies wurde bewußt in Kauf genommen, da so der Großteil des bereits existierenden C-Codes übernommen werden und dieser somit ebenfalls von den Vorteilen von C++ profitieren konnte. Zudem sind die Schwächen von C bekannt, eine von Grund auf neu entwickelte Sprache hätte auch anfangs unbekannte Schwächen. Es hat sicher zum Erfolg von C++ beigetragen (und trägt immer noch bei), daß ein C-Programmierer diese neue Sprache dadurch erlernen kann, daß nur er die neuen Features erlernen muß und nicht eine völlig neue Sprachstruktur.

Auch die Sprache C hat sich entwickelt, und das Konzept der Funktionsprototypen in ANSI-C beispielsweise stammt aus „C mit Klassen". Das Ideal für C++ wäre es, so nahe an ANSI-C zu sein, wie möglich - aber nicht näher. Hundertprozentige Kompatibilität ist nicht und war auch nie das Ziel von C++ , da dies mehr als einen Kompromiß bezüglich der Programmiersicherheit bedeuten würde; Programmiersicherheit ist aber eines der Hauptziele von C++ .

1.2 Die Philosophie von C++

C wurde entwickelt als maschinenunabhängige Maschinensprache. Dieses Grundkonzept mitsamt seiner Leistungsfähigkeit sollte für C++ beibehalten werden, der wesentliche Unterschied zwischen C und C++ liegt in der Ausdrucksstärke. In C kann sehr kompakt und ausdrucksstark geschrieben werden, in C++ noch viel stärker. Als Konsequenz daraus muß der Programmierer auch viel stärker auf die Typen der Objekte achten, mit denen er hantiert, allerdings wird er dabei auch vom Compiler mehr unterstützt als in C - ohne daß dies auf die Geschwindigkeit des fertigen Codes irgendeinen Einfluß hätte.

C++ hat einen Quantensprung in derjenigen Programmgröße verursacht, die noch überschaubar ist. Ein einzelner Programmierer kann sich in einem kleinen 1000-Zeilen Programm auch dann noch zurecht finden, wenn es entgegen allen Regeln guten Stils geschrieben wurde. Bei einem 10,000-Zeilen Projekt mit schlechter Struktur hingegen werden neue Fehler mindestens so schnell eingeführt, wie alte entfernt werden. C++ wurde so gestaltet, daß auch ein einzelner Programmierer noch Programme über 30,000 Zeilen überschauen und verwalten kann. Es hat sich gezeigt, daß C++ dieses Ziel schon bei weitem überschritten hat.

Eine Programmiersprache hat zweierlei Anforderungen zu erfüllen: Sie soll dem Programmierer die Möglichkeit schaffen, seine Anweisungen optimal auszuführen, und sie soll eine ausreichende

Menge an Konzepten zur Verfügung stellen, damit er auch das tun kann, was er eigentlich möchte. Der erste Punkt erfordert eine „maschinennahe“ Sprache, mit der die Fähigkeiten der Hardware leicht ausgenützt werden können. Genau das ist das Grundkonzept der Sprache C. Der zweite Aspekt setzt eine „problemorientierte“ Sprache voraus, sodaß das Problem an sich möglichst direkt und ohne Umschweife umgesetzt werden kann. Diese Vision hatte (und hat) man bei den Erweiterungen von C nach C++ immer vor Augen.

1.3 Die Stufen zur objektorientierten Programmierung

1.3.1 Prozedurale Programmierung

Am Anfang stand die prozedurale Programmierung. Sie stand unter dem Dogma:

Entscheide, welche Funktionen nötig sind.
Verwende die bestmöglichen Algorithmen.

Kern der Programmierweise ist der verwendete Algorithmus. Eine Programmiersprache unterstützt dieses Konzept mit Funktionen, denen Parameter übergeben werden können und einen Wert liefern. Die erste prozedurale Sprache war Fortran; Algol, Pascal und C sind spätere Erfindungen im gleichen Sinn.

Ein Paradebeispiel für „guten“ prozeduralen Programmierstil ist die Wurzelfunktion `sqrt()`. Die Funktion erhält ein Argument und liefert mithilfe eines ausgefeilten mathematischen Algorithmus ein Resultat.

1.3.2 Modulare Programmierung

Im Laufe der Jahre haben die Programme eine Komplexität erreicht, die neben dem Entwurf von Funktionen auch eine gut durchdachte Organisation der Daten verlangt. Eine Ansammlung von aufeinander abgestimmten Funktionen mit den dazugehörigen Daten wird *Modul* genannt. Das Programmierprinzip lautet nun:

Entscheide, welche Module nötig sind.
Teile das Programm so auf, daß Daten nur
innerhalb eines Modules verwendet werden.

Dieser Grundsatz wird oft auch als das „Geheimnisprinzip“ bezeichnet. Natürlich kann es sinnvoll sein, auch Funktionen und Konstanten vor direktem, unkontrolliertem Zugriff von auáen zu verbergen. Alle Regeln für prozedurale Programmierung sind nun **innerhalb** eines Modules anzuwenden. Sollte es nicht möglich sein, eine Gruppe zusammengehöriger Daten und Funktionen zu finden, ist die gewöhnliche prozedurale Programmierung weiterhin sinnvoll.

In C kann dieses Prinzip über das „User-Interface“ der Header-Dateien und statische Variablen realisiert werden.

1.3.3 Datenabstraktion

Das Programmieren mit Modulen führt zu einer Zentralisierung aller Daten eines bestimmten Verwendungszweckes unter der Kontrolle eines Modules. Um die Fähigkeiten eines Modules mehr als einmal ausnützen zu können, müssen alle Daten gleichen Verwendungszweckes zu einer Datenstruktur zusammengefaát werden, damit sie parallel existieren können.

Für jeden neuen Datentyp müssen nun auch die notwendigen Funktionen zur Verfügung gestellt werden, um die Datenstruktur korrekt auf- und abzubauen. Dadurch unterscheiden sich diese Datentypen von den Standardtypen, wodurch leicht Fehler in der Verwendung entstehen können. Das Modulkonzept ermöglicht diese Programmierweise, aber unterstützt sie nicht.

In C++ und einigen anderen Sprachen können jedoch Typen definiert werden, die sich (fast) genauso verhalten wie die Standardtypen. Ein solcher Typ wird als *abstrakter Datentyp* bezeichnet. Das Grundkonzept lautet hier:

Entscheide, welche Typen nötig sind,
und stelle eine vollständige Anzahl an
Operationen für jeden Typ zur Verfügung.

Wo es keinen Sinn macht, mehr als eine Datengruppe gleichen Typs zu verwenden, ist modulares Programmieren weiterhin von Vorteil. Allerdings lassen sich die meisten, wenn auch nicht alle, Module besser über Datenabstraktion formulieren.

BEISPIEL: Umsetzung des mathematischen Ausdruckes $\vec{a} = \vec{b} + \vec{c}$
In C:
```
struct  vector  a,b,c;
        add(a,b,c);
```

In C++ :
```
vector  a,b,c;
        a = b + c;
```

Ein abstrakter Datentyp funktioniert wie eine „Black Box“. Einmal festgelegt, kann er nicht mehr geändert werden und auch nicht mehr für neue Anwendungen erweitert werden, außer, die Definition selbst wird geändert. Das ist natürlich extrem unflexibel. Eine Möglichkeit besteht darin, im Datenfeld selbst eine Typeninformation unterzubringen, anhand derer unterschieden werden kann, was im Speziellen zu geschehen hat.
BEISPIEL:
```
struct  geometric_Object
{
        vector  origin;
        double  size;
        enum    {       NOOBJECT, CIRCLE, TRIANGLE, QUADRAT } type;
};

void    draw(const geometric_Object*goe)
{
        switch(go->type)
        {
        case NOOBJECT: break;
        case CIRCLE:     ...
                        break;
        case TRIANGLE:   ...
                        break;
         ...
        }
        return;
}
```

Dieser Code könnte etwa einem Programm zur Behandlung von geometrischen Objekten entstammen, bei dem eine bestimmte Anzahl von Objekten graphisch dargestellt werden soll.

Eine Funktion wie `draw(const geometric_Object*)` muß über alle möglichen Arten von geometrischen Objekten Bescheid wissen. Soll irgendwann ein neuer Typ eingeführt werden, so muß diese Funktion wie auch alle anderen Operationen erweitert werden. Das ist eine Gefahrenquelle, da dadurch auch bereits ausgetester und fehlerfreier (soweit dies überhaupt möglich ist) Code verändert werden muß.

1.3.4 Objektorientierte Programmierung

Das Problem im vorigen Beispiel besteht darin, daß es keine Unterscheidung zwischen dem geometric_Object als solchem und den einzelnen, spezifischen Unterarten gibt . Diesen Unterschied auszudrücken und daraus Vorteile zu erzielen definiert das *objektorientierte* Programmieren. Der Vererbungsmechanismus aus C++ (ursprünglich von Simula übernommen) bietet eine Möglichkeit hierzu.

OBJEKTORIENTIERTE VERSION:

```
struct  geometric_Object
{
        vector          origin;
        double          size;

        virtual void    draw();
};
```

Diejenigen Funktionen, deren Aufruf festgelegt werden kann, aber nicht deren Funktionsweise, werden als „`virtual`“ bezeichnet. Sie kann dann in einer *abgeleiteten Klasse* definiert werden. Mithilfe dieses Mechanismus können Funktionen mit dem allgemeinen Objekt `geometric_Object` hantieren:

```
void    draw_element(const geometric_Object*go)
{
        go->draw();
        return;
}
```

Die eigentliche Funktion wird dann in einer abgeleiteten Klasse festgelegt:

```
struct  Circle :  geometric_Object
{
        void    draw()
        {
                 ...
        }
};
```

Die Sprechweise in C++ ist dann:

Die Klasse `Circle` *wurde abgeleitet von der Klasse* `geometric_Object`,

bzw.

Die Klasse `geometric_Object` *ist eine* Basisklasse *der Klasse* `Circle`.

> *Entscheide, welche Klassen nötig sind;*
> *stelle eine vollständige Menge von Operationen für jede Klasse zur Verfügung*
> *und verdeutliche Gemeinsamkeiten mithilfe des Vererbungsmechanismus.*

Wo solche Gemeinsamkeiten nicht existieren, ist Datenabstraktion ausreichend. In manchen Bereichen, wie interaktiver Graphik, besteht eine enormes Potential für objektorientiertes Programmieren, in anderen, wie arithmetischen Berechnungen, scheint hingegen auf den ersten Blick kaum Platz zu sein für die Möglichkeiten, die objektorientierte Programmierung bietet.

Gemeinsamkeiten zwischen verschiedenen Typen zu finden ist kein trivialer Prozeß. Sie müssen explizit gesucht werden und stellen sich oft erst beim Betrachten fertiger Klassen heraus.

1.4 C++ als Programmiersprache

Als Mensch denkt man zuallererst intuitiv - in Konzepten. Diese Konzepte muß man dann zum Programmieren in handfeste Gedanken, schlußendlich Programmzeilen, umwandeln; das Programmieren von Konzepten selbst war bislang nicht möglich. C++ ermöglicht dies jedoch mithilfe von *Klassen*, die die Gemeinsamkeiten ähnlicher Problemstellungen zusammenfassen, und mehr noch mithilfe von *templates*, die nichts anderes sind als programmierte Konzepte, bei deren Erstellung noch gar nicht bekannt sein muß, wofür sie später einmal eingesetzt werden können.

Kern der Programmierweise in C++ ist *objektorientiertes Denken*, das Denken in abgeschlossenen Systemen, einzelnen Elementen, die für sich selbst verantwortlich sind und nach außen als ein Ganzes wirken. Die Aufgabe des Programmierers besteht neben dem Erstellen solcher Objekte nur noch darin, die Kommunikation dieser Objekte untereinander zu verwalten. **Wie** diese abläuft und was dabei genau geschieht, ist unwichtig und trägt nicht zur Lösung eines Problemes bei. Die Objekte selbst können unabhängig von der Umgebung erstellt und getestet werden, wodurch maximale Sicherheit erreicht wird. Da diese Objekte nicht speziell auf ein konkretes Problem angepaßt entworfen werden, ist es ein leichtes, sie auch für andere, ähnliche Problemstellungen einzusetzen.

Dennoch ist eine Denkweise nicht wirklich etwas, das erlernt werden kann. Wem eine objektorientierte Denkweise von vorneherein fremd ist, wer nicht sofort den Nutzen gegenüber einer zentral gelenkten, alles steuernden molochartigen Verwaltung sieht und keine Hemmungen davor hat, sich auf einen guten Editor zu verlassen, mit dem Programmzeilen kreuz und quer durch den Quellcode gejagt, d.h. geschoben und kopiert, werden können, wem es nicht danach verlangt, identische Zeilen im Programmtext zu vermeiden, der sollte nicht versuchen, C++ zu erlernen, er hätte keinen Nutzen davon. C++ ist nicht mehr und nicht weniger als die Umsetzung des bereits Gedachten in maschinenlesbare Form.

C++ ist mehr als eine Programmiersprache, C++ ist eine Philosophie!

1.5 Von C nach C++

Viele Warnungen aus C wurden in C++ zu einem Fehler erhoben, auch manches, das in C nicht einmal eine Warnung zur Folge hat. Gerade dies kann für einen eingefleischten C-Programmierer ausgesprochen lästig sein, da er so gezwungen wird, „sauberer“ zu programmieren. Vor allem ältere Programme, die noch nicht nach dem ANSI-C Standard geschrieben wurden, sind kaum auf C++ umzuschreiben, da sie meist zuviele prähistorische (soll heißen: prä C++) Ungereimtheiten aufweisen — und dies teilweise im vollen Bewußtsein des Autors, häufiger aber aus reiner Bequemlichkeit.

1.6 Erweiterungen zu C

Einige Erweiterungen von C nach C++ sind äußerst praktisch und auch im herkömmlichen, prozeduralen Programmierstil gut verwendbar; sie sind letztlich eine Konsequenz des objektorientierten Konzeptes, lassen sich aber auch ohne Bezug zum Klassenmechanismus einsetzen.

1.6.1 Typkonversionen als Funktionsaufruf

Anstatt `(double)i` kann auch `double(i)` geschrieben werden um beispielsweise eine `int`-Variable in einen `double` zu konvertieren. Diese Schreibweise entspricht eher der intuitiven Vorstellung. Soll jedoch in einen Typ konvertiert werden, der aus mehr als einem Wort besteht, muß weiter die alte C-Schreibweise verwendet werden (wie in `(long double)i` oder `(int*)vptr`).

Dies ist eine Konsequenz daraus, daß die Typenkonversion einen überladbaren Operator darstellt.

In neueren Versionen von C++ werden zudem verschiedene Arten von Typkonversionen unterschieden:
`static_cast`<*Typ*>(*Variable*) zur Konversion unterschiedlicher Datentypen,
`const_cast`<*Typ*>(*Variable*) zur Modifizierung der `const`-Spezifikation
`reinterpret_cast`<*Typ*>(*Variable*) zur Interpretation eines Zeigers als Zahlenwert (`int`) und vice versa.
Daneben gibt die `dynamic_cast`<*Typ*>(*Variable*) - Typenkonversion, die jedoch nur in der objektorientierung eine Rolle spielt und für intrinsische Datentypen unwichtig ist.

1.6.2 Variablen-/Funktionsnamen dürfen beliebig lang sein

In C dürfen Variablen-/Funktionsbezeichnungen eine maximale Länge nicht überschreiten, zumeist 32 Zeichen. Überzählige Zeichen werden ignoriert. Dieses Limit aus den Tagen des frühen Compilerbaues wurde in C++ abgeschafft.

1.6.3 Zeilenweise Kommentare

Zusätzlich zum blockweisen Kommentar mit `/*` und `*/` kann in C++ ein zeilenweiser Kommentar verwendet werden, der mit `//` beginnt und bis zum Zeilenende reicht. Da Kommentare üblicherweise vom Präprozessor und nicht vom C-Compiler behandelt werden, lassen manche C-Compiler, die auch für C++ ausgelegt sind, derartige Kommentare auch in C zu. Dies ist eigentlich ein Fehler, da `//`-Kommentare nur in C++ erlaubt sind, ein echter C-Compiler würde einen Fehler melden.

Wer seine Programme als C++ übersetzen möchte, um irgendwelche Features dieser Sprache auszunützen, sollte auch in C++ programmieren.

1.6.4 Funktionen zur Initialisierung statischer Variablen

In C können statische Variablen nur durch konstante Ausdrücke initialisiert werden. In C++ sind Funktionsaufrufe estattet, auch bei globalen Variablen. Die Initialisierungsfunktion wird dabei zur Laufzeit, **vor** dem Aufruf von `main()`, ausgewertet.

Der notwendige Mechanismus, der all dies ermöglicht, ist ein Abfallprodukt aus der Tatsache, daß Konstruktoren und Destruktoren globaler Objekte ebenfalls vor bzw. nach `main()` ausgeführt werden müssen.

1.6.5 Strukturdefinitionen sind automatisch ein `typedef`

Wird ein `struct punkt { ... };` definiert, kann der erstellte Typ sofort mit `punkt`, nicht wie in C als `struct punkt`, angesprochen werden. Diese Vereinfachung enspricht eher der intuitiven Vorstellung und macht das Schlüsselwort `typedef` weitestgehend überflüssig.

Hier wurde einfach historischer Ballast über Bord geworfen.

1.6.6 Dynamischer Speicher

Während `malloc()` und `free()` nur Libraryfunktionen sind und eigentlich von vorneherein nichts mit der Sprache C zu tun haben, sind die Funktionen zur Verwaltung des dynamischen Speicher in C++ in der Sprache selbst implementiert. Mit dem **Operator** `new` wird Speicher in der Größe des angegebenen Objekte geholt und ein Zeiger vom Typ „Zeiger auf dieses Objekt" zurückgeliefert (bzw. `NULL`, wenn kein Speicher vorhanden ist):

```
struct punkt { int x,y; };

punkt *punkt_ptr;

        punkt_ptr = new punkt;
```

Die fehlerträchtige und unschöne Zeigerkonversion beim Aufruf von `malloc()` fällt somit ebenfalls weg. Parallel zu `new` gibt es den Operator `delete` um den angeforderten Speicher wieder freizugeben.

`new` und `delete` sind in C++ überladbare Operatoren, die sich für jede Klasse anders verhalten können, um beispielsweise eine effizientere Speicherverwaltung für eine bestimmte Gruppe von Objekten zu implementieren.

1.6.7 Die Deklaration ist eine Anweisung

In C können Variablen nur zu Beginn eines Programmblockes deklariert (bzw. definiert) werden, in C++ an jeder beliebigen Stelle, wo auch eine Anweisung erlaubt ist. Besonders beliebt ist dies bei `for`-Schleifen:

```
         ...
        for(int i=0;i<100;i++)
                ...
         ...
```

Die Notwendigkeit bei der Definition einer Variablen zu kommentieren, wozu sie drei Seiten weiter gebraucht wird, fällt damit weg, da man sie jetzt wirklich erst dort definieren kann, wo sie auch gebraucht wird. Die Variable ist ab dem Punkt ihrer Definition bis zum Ende des momentanen Programmblockes gültig. Für den Compiler ist es ebenfalls eine Optimierungshilfe, wenn eine Variable nur in einem kurzen Bereich benötigt wird.

Diese Erweiterung wurde eingeführt, da die Initialisierung eines Objektes recht aufwendig sein kann. Oft können Objekte nicht sofort zu Beginn eines Blockes initialisiert werden, da noch irgendwelche Berechnungen ausgeführt werden müssen. Durch diese Erweiterung kann eine unnötige „Leerinitialisierung" mit anschließendem Kopieren beim eigentlichen Initialisieren vermieden werden (beides kann sehr aufwendig sein).

1.6.8 Der Bereichsoperator

Wird in einer C-Funktion eine Variable gleichen Namens wie eine globale Variable definiert, gibt es keine Möglichkeit, diese globale Variable anzusprechen, da die lokale Variable Vorrang hat. In C++ kann der neu eingeführte Bereichsoperator „::" für die globale Variable verwendet werden:

```
int     value;

void    f()
{
int     value;

        value = 12;      // lokale Variable belegen
        ::value = 23;    // globale Variable belegen
}
```

Bei verschachtelten Blöcken bringt allerdings auch der Bereichsoperator keine Hilfe mehr.

Selten wird der Bereichsoperator in C++ wirklich in dieser Form angewandt, weit öfter wird er gebraucht um Klassenelemente eindeutig ansprechen zu können.

1.6.9 Der Defaultparameter

Bei der ersten Funktionsdeklaration können Defaultwerte für die Parameter vergeben werden:

```
void    putpixel(int x, int y, int color = 15);
```

Die Funktion `putpixel()` kann mit zwei oder drei Parametern aufgerufen werden, für `color` wird ggf. der Zahlenwert 15 eingesetzt. Ab einem bestimmten Parameter können die weiteren (zumeist unbedeutenderen) mit mehr oder weniger sinnvollen Werten vordefiniert werden. Nach einem Defaultparameter müssen auch alle folgenden vorbelegt werden.

Werden alle Parameter vorbelegt, kann eine Funktion wie eine `void`-Funktion aufgerufen werden, obwohl sie dennoch Parameter auswertet.

Anmerkung: Dieser Mechanismus ist nicht identisch mit der aus C bekannten Ellipse, den Funktionen mit beliebig vielen Parametern. Eine Funktion mit Defaultparametern wird immer mit der gleichen Anzahl an Parametern aufgerufen, nur müssen nicht alle angegeben werden.

1.6.10 Polymorphie

In C wird eine Funktion eindeutig anhand ihres Namens erkannt, in C++ werden auch die Funktionsparameter zur Identifikation herangezogen. Das heißt, mehrere Funktionen können den gleichen Namen haben, sofern sie andere Parameter besitzen:

```
void    putpixel(int x,int y,int color);
void    putpixel(double x,double y,int color);
```

Diese beiden Deklarationen beschreiben zwei völlig unterschiedliche Funktionen, die `double`-Version könnte beispielsweise eine zusätzliche Skalierung durchführen. Beim Aufruf entscheidet der Compiler anhand der Parameter, welche Variante verwendet wird, in Zweifelsfällen wird eine Meldung ausgegeben. Ein Funktion wie die obige wird als „*polymorph*" bezeichnet, da sie in mehreren (*polys* - griech. „viel") Varianten (*morpheo* - griech. „verwandeln") auftreten kann.

Polymorphie ist der Schlüssel zur Datenabstraktion, denn so kann für benutzerdefinierte Typen eine zusätzliche Funktionsversion geschrieben werden, sodaß sich die Verwendung dieses Typs in nichts von der Verwendung eines Standardtyps unterscheidet:

```
void    putpixel(punkt *p,int color);
```

1.6.11 `inline`-Funktionen

Macros werden in C aus Effizienzgründen häufig angewendet, sind aber sehr fehlerträchtig, da sie ja nicht den Gesetzen der Sprache C unterliegen. In C++ kann einer Funktion das Schlüsselwort `inline` vorangestellt werden - damit verhält sie sich effizienzmäßig wie ein Macro, unterliegt aber der vollen Syntaxkontrolle durch den Compiler. `inline`-Funktionen werden nicht aufgerufen, sondern an der Stelle ihres Aufrufes eingefügt. Der Compiler kann dadurch auch innerhalb der Funktion noch etwas optimieren, beispielsweise Ausdrücke auswerten, wenn die Funktion mit konstanten Parametern aufgerufen wird. Dies erwartet man ja auch von einem Macro.

`inline`-Funktionsdefinitionen gehören ebenfalls in eine Headerdatei (im Gegensatz zu gewöhnlichen Funktionsdefinitionen), da dem Compiler beim „Aufruf" die gesamte Funktion bereits bekannt sein muß.

Diese Erweiterung ist notwendig geworden, da das Schutzkonzept von Klassenelementen in C++ die Verwendung vieler kleiner Funktionen nötig macht. Ein echter Funktionsaufruf an jeder Stelle würde das gesamte Konzept ineffizient und damit unattraktiv werden lassen.

1.6.12 Referenzen

Eine Referenz ist ein derefenzierter Zeiger. Ihr Nutzen ist vor allem bei Funktionsaufrufen einsichtig:

```
void    putpixel(punkt&p,int color)
{
        // Dieser Funktionsaufruf ist nicht rekursiv, sondern polymorph !
        putpixel(p.x,p.y,color);
}
```

Beim Aufruf muß nicht explizite die Adresse der Strukturvariablen übergeben werden, dennoch bleibt die Effizienz dieselbe. Das Konzept der Referenz in C++ ist viel allgemeiner als etwa in Pascal (dort entsprechen die `var`-Parameter einer Referenz als Funktionsparameter):

```
int     i;
int     &j = i;
```

Hier wird `j` als Referenz auf `i` definiert. `j` ist quasi ein Aliasname für `i`. Wird der Referenz `j` etwas zugewiesen, wird `i` verändert. Die Adresse von `j` ist die Adresse von `i` (also `&j==&i`). Eine Referenz kann nur initialisiert werden, eine nachträgliche Änderung ist nicht möglich, da durch die Zuweisung ja immer das referenzierte Objekt angesprochen wird.

Praktisch ist die Referenz neben der Verwendung als Funktionsparameter bei komplizierten Ausdrücken:

```
punkt   Parray[100];
        for(int i=0;i<100;i++)
        {
        punkt&p = Parray[i];
                p.x = i;
                p.y = 100-i;
        }
```

Im obigen Beispiel wird der Arrayzugriff nur einmal ausgeführt (`Parray[i]`). Neben der übersichtlicheren Schreibweise ist dies auch effizienter.

1.6.13 bool - Typ

Heftige Diskussionen hat der Vorschlag ausgelöst, einen eigenen `bool`-Typ einzuführen. Im Zusammenhang damit werden auch `true` und `false` Schlüsselwörter. Eine Variable vom Typ `bool` ist **kein** Bit, sondern *mindestens* ein `char`, denn er muß adressierbar sein, von einem Bit kann aber keine Adresse gebildet werden.

Zu Recht wird der eingefleischte C-Programmierer fragen, wozu ein `bool` nötig ist, gibt es doch den `int`, der entweder 0 (falsch) oder nicht 0 (wahr) ist.

Eines der Argumente ist, daß viele Anwendungen aus rein ästethischen Gründen einen Boolean-Typ mit `true`- und `false`-Werten definieren, entweder als `enum` , oder `int`, manchmal kleingeschrieben, dann wieder mit großem Anfangsbuchstaben oder einfach alles in Großbuchstaben. Mit einem intrinsischen `bool`-Typ können alle diese Schreibweisen vereinheitlicht werden.

Das viel stärkere Argument aber ist, daß Funktionen für diesen neuen Typ überladen werden können. Der Ausdruck `1<2` wird also bei der Ausgabe mithilfe von polymorphen Funktionen nicht mehr `1` liefern, sondern den Text „true“ (oder „wahr“, wenn die geplante länderspezifische Streamanpassung realisiert wird), da die Ausgabefunktion für den `bool`-Typ zusätzlich überladen werden kann.

Mit altem Programmcode sollte es keine Schwierigkeiten geben, da jeder `int`-Ausdruck nach den herkömmlichen Regeln in einen `bool` konvertiert werden kann.

2 Elementfunktionen - Begriff des Objektes

AUFGABENSTELLUNG:
Es soll eine *Klasse* zum einfachen Umgang mit Graphikdateien erstellt werden.

SCHLÜSSELBEGRIFFE:
Elementfunktionen, Gültigkeitsbereich, Bereichsoperator, Konstruktor, Destruktor, `this`

PROGRAMMBESCHREIBUNG:
Die Basis einer derartigen Klasse ist eine Grundstruktur, die alle notwendigen Daten enthält. Neben den eigentlichen Daten des Dateiheaders (siehe C-Skriptum) und einem für die Dateioperationen notwendigen `FILE*` sind dies die Pixeldaten, also jeweils drei `char`'s für Rot, Grün und Blau bei einer RGB-Datei. Die entsprechende Datenstruktur einer Targa-Datei hätte also die folgende Form:

```
struct  Targa
{
        FILE    *file;          // Dateistruktur aus <stdio.h>

        short   MaxX, MaxY;     // TARGA – Dateiheader
         ...                    // weitere Datenelemente

        char    *pixel;         // Pixeldaten
};
```

Mit dieser Datenstruktur unmittelbar verbunden sind die Funktionen zum

- Öffnen der Datei
- Schließen der Datei
- Setzen eines Pixels auf eine bestimmte Farbe
- Diverse andere Funktionen wie Lesen eines Pixels, Linie zeichnen, u.v.a.m.

Beim Öffnen der Datei wird die Datenstruktur initialisiert, der notwendige Speicher für die Pixeldaten angefordert und die Daten einer existierenden Datei ggf. eingelesen (sofern keine neue Datei erzeugt werden soll). Umgekehrt schreibt die Schließfunktion die Daten vom Speicher in die Datei und gibt den Speicher wieder frei. Diese Funktionen könnten schematisch so aussehen:

```
// Neue Datei öffnen
void    targa_open(Targa*targastruct, const char*filename, int X, int Y)
{
        targastruct->file = fopen(filename, "wb");
        targastruct->MaxX = MaxX;
         ...
        targastruct->pixel = new char[3*MaxX*MaxY];
}

// Datei schließen
void    targa_close(Targa*targastruct)
{
        // Dateiheader schreiben
        fputc(targastruct->MaxX, targastruct->file);
         ...
```

```
        fwrite(targastruct->pixel,
        targastruct->MaxX*targastruct->MaxY*3, 1, targastruct->file);

        delete targastruct->pixel;
}

void    targa_putpixel(Targa*targastruct, int x, int y, char r, char g, char b)
{
char    *p = targastruct->pixel + 3*(x+y*targastruct->MaxX);
        *p++ = b;
        *p++ = g;
        *p++ = r;
}
```

Ein Hauptprogramm, das eine diagonale, farbige Linie in eine Targadatei zeichnen soll, würde dann in etwa folgendermaßen aussehen:

```
main()
{
Targa   tga;
        targa_open( &tga, "test.tga", 100, 100);

        for(int i=0;i<tga.MaxX;i++)
                targa_putpixel(&tga, i,i, 255-2*i, 2*i, 0);

        targa_close( &tga );
}
```

MECHANISMEN IN C++

So wie oben beschrieben, kann in C eine Graphikdateiverwaltung aufgebaut werden. Dennoch wird sie immer etwas unbefriedigend bleiben, da die Schreibweise recht umständlich ist. C++ bietet mit den *Elementfunktionen* die Möglichkeit einer sehr eleganten Formulierung. Eine Elementfunktion (engl.: *memberfunction*) wird wie ein Datenelement *innerhalb* einer Struktur definiert. Sie wird damit quasi zu einem Teil dieser Struktur und kann wie ein solcher aufgerufen werden. Zudem sind alle Datenelemente für eine Elementfunktion direkt verfügbar:

```
struct  Targa
{
        FILE    *file;          // Dateistruktur aus <stdio.h>

        short   MaxX, MaxY;     // TARGA – Dateiheader
         ...                    // weitere Datenelemente

        char    *pixel;         // Pixeldaten

        // Elementfunktion open()
        // Neue Datei öffnen
        void    open(const char*filename, int X, int Y)
        {
                file = fopen(filename, "wb");
```

```
                MaxX = X;
         ...
        pixel = new char[3*X*Y];
        }

        // Elementfunktion close()
        // Datei schließen
        void    close()
        {
                // Dateiheader schreiben
                fputc(MaxX, file);
                 ...

                fwrite(pixel, MaxX*MaxY*3, 1, file);

                delete pixel;
        }

        void    putpixel(int x, int y, char r, char g, char b)
        {
        char    *p = pixel + x+y*MaxX;
                *p++ = b;
                *p++ = g;
                *p++ = r;
        }

};
```

Alle Datenelemente innerhalb einer Elementfunktion sind *relativ* zum aktuellen Datenobjekt, in diesem Fall einer Variablen vom Typ `struct Targa`. D.h., eine Elementfunktion kann nicht für sich alleine aufgerufen werden, sondern ausschließlich[1] in Verbindung mit einem Datenobjekt. Der Aufruf erfolgt wie der Zugriff auf ein Datenelement:

```
main()
{
Targa   tga;
        tga.open("test.tga", 100, 100);

        for(int i=0;i<tga.MaxX;i++)
                tga.putpixel(i,i, 255-2*i, 2*i, 0);

        tga.close();
}
```

Diese Formulierung ist um einiges eleganter als das C-Pendant. Man sagt quasi dem Objekt `tga`, es soll „sich selbst“ öffnen, einen Pixel setzen, sich schließen, anstatt eine Funktion zu verwenden, der man sagt, mit welchem Objekt sie was tun soll. Das Datenobjekt, dessen Adresse in der C-Version explizit mit `&tga` angegeben werden muß, wird in C++ implizit beim Aufruf der Elementfunktion übergeben, die so immer „weiß“, zu welchem Objekt sie eigentlich gehört.

In Fällen, in denen eine Elementfunktion die Adresse des Objektes kennen muß, zu dem sie

[1] Soferne sie nicht als `static` deklariert wurde, genaueres später

aufgerufen wurde, steht die Variable `this` zur Verfügung. `this` ist nur in Elementfunktionen definiert und vom Typ eines Zeigers auf die Struktur, zu der die Elementfunktion gehört.

```
struct  Targa
{
         ...

        // Elementfunktion open()
        // Neue Datei öffnen
        void    open(const char* filename, int X, int Y)
        {
                printf("Öffne Targa-Objekt %p als Datei %s.\n",this, filename
                 ...
        }

         ...
};


main()
{
Targa   tga1, tga2;
        printf("tga1 hat die Adresse %p, tga2 die Adresse %p.\n",&tga1, &tga2);
        tga1.open("1.tga",100,100);
        tga2.open("2.tga",100,100);
}
```

Im obigen Fall ist `this` in der Elementfunktion `open()` vom Typ `Targa*`.

Der eigentliche Name einer Elementfunktion ist mehr als nur der Funktionsname selbst, da dies sonst zu häufig zu Namenskonflikten würde. Die vollständige Bezeichnung einer Elementfunktion setzt sich aus Strukturnamen und Funktionsnamen zusammen, getrennt durch den *Bereichsoperator* `::`. Die Elementfunktion `open()` der Struktur `Targa` hat daher den vollständigen Namen

```
        Targa::open()
```

Eine Elementfunktion kann auch in einer Struktur nur deklariert und außerhalb an anderer Stelle definiert werden. Während man sich bei der *Deklaration* noch die explizite Angabe des *Gültigkeitsbereiches* (die Struktur `Targa`) sparen kann, ist dies bei der *Definition* unbedingt nötig:

```
struct  Targa
{
         ...

        // Deklaration der Elementfunktion open()
        // Neue Datei öffnen
        void    open(const char* filename, int X, int Y);

         ...
};

void    Targa::open(const char* filename, int X, int Y)
```

```
{
        ...
}
```

In der Elementfunktion `Targa::open()` selbst sind die Datenelemente (`MaxX` etc.) genauso direkt verfügbar wie bei der Funktionsdefinition innerhalb der Struktur `Targa`.

Elementfunktionen, die innerhalb einer Struktur definiert werden, sind `inline`. Soll eine Elementfunktion nicht `inline` ausgeführt werden, **muß** sie außerhalb ihrer Struktur definiert werden. Dies ist immer dann ratsam, wenn eine Funktion zu komplex ist, als daß sie als `inline`-Funktion von Nutzen wäre. Im allgemeinen liegt die Grenze, ab der eine Funktion nicht mehr `inline` sein sollte, bei etwa drei Zeilen. Bei größeren Funktionen ist der Zeitaufwand für den Funktionsaufruf im Vergleich mit der Ausführungszeit des Funktionsblocks vernachlässigbar.

Viele `inline`-Funktionen bewirken zudem eine bisweilen massive Vergrößerung des kompilierten Programmes und setzen die Compiliergeschwindigkeit merklich herab. Die Übersichtlichkeit des Quellcodes steigt ebenfalls, wenn die Definition von Elementfunktionen in ein separates Modul ausgelagert wird. Vor der Definition einer Elementfunktion muß die zugehörige Struktur bereits definiert sein und die entsprechende Deklaration der Elementfunktion aufweisen. *Eine Elementfunktion kann* **nur** *in einer Struktur deklariert werden, eine nachträgliche Deklaration ist nicht möglich.*

2.1 Konstruktor/Destruktor

Unter allen Elementfunktionen ist diejenige besonders ausgezeichnet, deren Name mit dem der Struktur identisch sind:

```
struct  Targa
{
        ...

        Targa(const char* filename, int X, int Y);

        ...
};
```

Diese Funktion, `Targa::Targa()` im obigen Fall, heißt der *Konstruktor* der Struktur `Targa`. Er wird **ausschließlich** bei der Definition („Konstruktion“) eines Objekte aufgerufen, ein nachträglicher Aufruf ist nicht möglich:

```
main()
{
Targa   tga("test.tga", 100, 100);
        ...
}
```

Ein Konstruktor wird üblicherweise verwendet, um ein Objekt mit sinnvollen Werten vorzubelegen. So kann gewährleistet werden, daß niemals eine Datenstruktur mit undefinierten Werten existiert, wie dies sonst nach Definition eines Objektes und vor dem Aufruf der `Targa::open()`-Funktion der Fall wäre. Eine derartige Gewährleistung ist in C (und manchen anderen „objektorientieren“ Programmiersprachen) nicht möglich.

Die Konstruktorfunktion liefert keinen Funktionswert, ein Funktionstyp darf bei der Deklaration/Definition deshalb nicht angegeben werden, nicht einmal der eigentlich korrekte Typ `void`, da es keine Alternativen dazu gibt und eine explizite Angabe sinnlos wäre.

Wie alle[2] Funktionen in C++ , können auch Konstruktorfunktionen polymorph sein, d.h. es können mehrere Funktionen gleichen Namens mit unterschiedlichen Parametern existieren:

```
struct  Targa
{
        // existierende Datei öffnen
        Targa(const char*filename);

        // neue Datei erzeugen
        Targa(const char*filename, int X, int Y);
};
```

2.1.1 Der Defaultkonstruktor

Einen Spezialfall stellt derjenige Konstruktor dar, der ohne Parameter aufgerufen wird, der sogenannte *Defaultkonstruktor*:

```
struct  Targa
{
        Targa();            // Defaultkonstruktor
};
```

Er wird immer dann aufgerufen, wenn ein Datenobjekt nicht mit Parametern initialisiert wird, d.h. so, wie es in C der Fall wäre:

```
main()
{
Targa   tga;                // Aufruf des Defaultkonstruktors
}
```

Aus Kompatibilitätsgründen ist der Defaultkonstruktor auch definiert, wenn er nicht explizit definiert wurde (dann tut er „nichts“, d.h. er läßt das zugehörige Datenobjekt uninitialisiert), allerdings nur dann, wenn keine anderen Konstruktorfunktionen definiert wurden. Falls eine Struktur Konstruktorfunktionen enthält, aber keinen Defaultkonstruktor, ist ein parameterloser Aufruf wie im obigen Beispiel nicht mehr möglich. Auf diese Weise kann die unbedingte Angabe von Parametern bei der Konstruktion von Objekten vom Programmierer erzwungen werden.

2.1.2 Der Copy-Konstruktor

Ein weiterer Spezialfall eines Konstruktors, der implizit definiert ist (aber auch explizit definiert werden kann), ist der *Copy-Konstruktor*, der immer dann verwendet wird, wenn ein Objekt mit einem Objekt gleichen Typs initialisiert werden soll:

```
struct  Targa
{
        Targa(Targa&);
};
```

[2] Mit Ausnahme der `main()`-Funktion und den Destruktorfunktionen (siehe später)

```
main()
{
Targa   tga1;           // Aufruf des Defaultkonstruktors
targa   tga2(tga1);     // Aufruf des Copykonstruktors
}
```

Der Copykonstruktors entspricht einem Konstruktor mit einem Parameter vom Typ einer Referenz auf ein Objekt der gleichen Struktur, also in diesem Fall `Targa::Targa(Targa&)`. Dabei kann der Parameter auch `const` sein. Falls der Copykonstruktors nicht explizit definiert wird, ist er als elementweises Kopieren definiert. Wenn eine Struktur wiederum aus weiteren Strukturen aufgebaut ist, wird dabei jeweils deren Copykonstruktor aufgerufen.

2.1.3 Konstruktor-Syntax

Im Konstruktor können Datenelemente durch die aus C bekannte *Zuweisung* initialisiert werden:

```
Targa::Targa(const char*filename, int X, int Y)
{
        file = fopen(filename, "rb+");
        MaxX = X;
        MaxY = Y;
         ...
}
```

Alternativ ist in C++ die *Konstruktorsyntax*, die an den Aufruf eines Konstruktors für jedes Datenobjekt angelehnt ist, für Initialisierungen möglich:

```
Targa::Targa(const char*filename, int X, int Y)
:  file( fopen(filename, "rb+") ), MaxX(X), MaxY(Y) ...
{
}
```

Diese Syntax ist sogar notwendig, wenn eine Struktur Objekte enthält, für die kein Defaultkonstruktor existiert (vgl. 2.1.1). Dabei ist zu beachten, daß die Initialisierung der Objekte in der Reihenfolge ihrer *Definition in der Struktur* erfolgt, *nicht* in der Reihenfolge, wie sie im Konstruktor selbst angegeben werden!

2.1.4 Der Destruktor

In Analogie zum Konstruktor gibt es eine spezielle Elementfunktion, die mit der Tilde ~ und dem Strukturnamen bezeichnet wird (in Anlehnung an das bitweise NOT in C), den *Destruktor*. Er wird immer dann aufgerufen, wenn ein Objekt „vernichtet“ wird, beispielsweise beim Verlassen einer Funktion mit `return`:

```
struct  Targa
{
        ~Targa();
};

Targa::~Targa()
```

```
{
        printf("Objekt %p wird zerstört!\n", this);
}

main()
{
Targa   tga;
         ...
        return 0;       // impliziter Aufruf von Targa::~Targa()
}
```

Der Destruktor wird implizit am Ende der Lebensdauer eines Objektes aufgerufen, ohne daß sich der Programmierer darum kümmern muß. Er wird üblicherweise dazu verwendet, Speicher freizugeben, Dateien zu schließen und ähnliches.

Da ein Destruktor nicht explizit aufgerufen wird, können an ihn auch keine Parameter übergeben werden, die Destruktorfunktion ist daher immer parameterlos (und kann deshalb auch nicht polymorph sein).

Mit den entsprechenden Konstruktor/Destruktorfunktionen in der Struktur `Targa` läßt sich das Hauptprogramm von früher in der folgenden, kompakten, aber dennoch übersichtlichen Form schreiben:

```
main()
{
Targa   tga("test.tga", 100, 100);

        for(int i=0;i<tga.MaxX;i++)
                tga.putpixel(i,i, 255-2*i, 2*i, 0);
}
```

2.1.5 Dynamische Speicherobjekte

Bei Objekten, die dynamisch mittels `new` erzeugt bzw. mit `delete` vernichtet werden, wird wie in den oben erwähnten Fällen die Konstruktor- bzw. Destruktorfunktion aufgerufen. Bei `new` können dabei die Konstruktorparameter angegeben werden:

```
main()
{
Targa   *tgaptr = new Targa("test.tga", 100, 100);

        for(int i=0;i<tga.MaxX;i++)
                tgaptr->putpixel(i,i, 255-2*i, 2*i, 0);

        delete tgaptr;
}
```

Der Vorteil dynamischer Objekte liegt auf der Hand: Nur so kann etwa eine Funktion ein Objekt erzeugen, daß dann an die aufrufende Funktion weitergegeben wird, ohne daß das Objekt selbst dabei kopiert werden muß.

3 Schutzmechanismen - Begriff der Klasse

AUFGABENSTELLUNG:
Es soll eine *Klasse* zum einfachen Umgang mit doppelt verketteten Listen erstellt werden.

SCHLÜSSELBEGRIFFE:
`private`, `public`, `class`, `struct` als abgeleiteter Typ, `friend`-Funktionen, `friend`-Klassen.

Im ersten Abschnitt wurde gezeigt, wie man Daten und Funktionen zu einer Einheit zusammenfügen kann, und wie gewährleistet werden kann, daß eine Datenstruktur immer gültige Datenelemente enthält. Der Programmierer braucht sich nicht mehr um die Details kümmern, er muß nicht wissen, ob eine Struktur beispielsweise dynamische Daten enthält, da diese ggf. von selbst angefordert und freigegeben werden.

Dennoch handelt es sich bei dieser Programmierweise um reine Konvention. Niemand hindert einen Programmierer daran, in der Struktur `Targa` aus dem vorigen Beispiel den `pixel`-Zeiger nach dem Aufruf von `Targa::open()` auf `NULL` zu setzen oder auf unsinnige Werte zu verbiegen. Natürlich legt sich der Programmierer damit selbst hinein, aber das muß ja nicht unbedingt absichtlich geschehen. Vor allem dann, wenn fremder Code übernommen werden soll, können Unklarheiten entstehen. Es kann sehr wohl sinnvoll sein, ein manche Datenelemente zu verändern, beispielsweise könnte man ein Datenelement für die momentante Zeichenfarbe vorsehen, sodaß auch Elementfunktionen ohne explizite Angabe einer Farbe möglich sind, andere Datenelemente, wie den Zeiger auf die Pixeldaten, sollten nur durch die dafür vorgesehenen Elementfunktionen umgesetzt werden.

Um klarzustellen, welche Elemente von außen verändert werden dürfen und um damit den Programmierer „vor sich selbst“ zu schützen, bietet C++ das **Schutzkonzept**. Sollen bestimmte Strukturelemente (der Begriff „Element“ umfaßt hierbei sowohl Datenelemente als auch Elementfunktionen) von außen nicht zugänglich sein, werden sie als `private` oder `public` deklariert:

```
struct  Targa
{
        char    r, g, b;            // letzte Zeichenfarbe

private:  FILE*file;                // private Datenelemente
        int     MaxX, MaxY;
         ...

public: int     Xsize()         {       return MaxX;            }
        int     Ysize()         {       return MaxY;            }
};
```

Alle auf `private` folgenden Datenelemente und Elementfunktionen sind vor dem Zugriff von außen geschützt („privat“), jeder Versuch wird mit einer Fehlermeldung des Compilers quittiert. Das Gegenstück zu `private` ist `public`: Alle auf `public` folgenden Strukturelemente sind von außen zugänglich („öffentlich“). Im obigen Fall sind die Datenelemente `MaxX` und `MaxY` vor dem Zugriff von außen geschützt (was fatal wäre, da diese Zahlenwerte mit der angeforderten Größe des Pixelspeichers übereinstimmen müssen) und können nur innerhalb von Elementfunktionen verändert werden. Um die Zahlenwerte für die aktuelle Bildgröße dennoch auch außerhalb der Elementfunktionen verwenden zu können, stehen die Elementfunktionen `Targa::Xsize()` und `Targa::Ysize()` zur Verfügung. Deren Aufruf ist ungefährlich, da sie die Datenstruktur ja nicht verändern.

Es ist recht gebräuchlich, öffentliche (`public-`) Elementfunktionen zu schreiben, deren einziger Nutzen darin besteht, den Wert eines privaten Datenelementes zu liefern. So kann dieses Datenelement von außen zwar ausgelesen, aber nicht beschrieben werden. Ein guter Compiler optimiert

den Aufruf einer solchen Elementfunktion, wenn sie `inline` ist, so, daß er einem direkten Zugriff auf das Datenelement entspricht. Der zusätzliche Schutz kostet daher keinerlei Laufzeit, für den Programmierer verringert sich jedoch das Fehlerpotential.

Da ein Objekt als Ganzes behandelt werden soll, sind normalerweise die meisten Strukturelemente `private` und nur wenige `public`. Um dies zu unterstützen, kann statt `struct` das C++ –Schlüsselwort `class` verwendet werden. *Eine* **Klasse** *(*`class`*) ist eine Struktur (*`struct`*), in der alle Elemente von vorneherein* `private` *sind.* Um eine Klasse verwenden zu können, müssen also auf jeden Fall einige Elemente `public` deklariert werden:

```
class   Targa
{
        FILE*file;          // private Datenelemente
         ...

public: Targa(const char*filename);
        Targa(const char*filename, int x, int y);
        ~Targa();

        void    putpixel(int x, int y, char r, char g, char b);
};
```

3.1 `friend`-Funktionen

Die Zugriffserlaubnis auf die Elemente einer Klasse kann zusätzlich für einzelne Funktionen oder ganze Klassen vergeben werden. Dazu müssen diese als `friend` deklariert werden:

```
class   A
{
        int     i,j,k;                   // private Elemente

        friend  int sum(const A&a);

        friend  class   B;
};

class   B
{
        int     Asum;

public: B(const A&a)
        :Asum( a.i + a.j + a.k)
        {}
};

int     sum(const A&a)
{
        return a.i+a.j+a.k;
}
```

Im obigen Beispiel wird der Funktion `sum()` erlaubt, auf die privaten Elemente der Klasse A zuzugreifen. Der Klasse B wird generell der Zugriff auf alle Elemente der Klasse A erlaubt, d.h.

jede Elementfunktion der Klasse B darf auf die privaten Elemente der Klasse A zugreifen. Nur deshalb darf der Konstruktor der Klasse B im obigen Beispiel die privaten Elemente der Klasse A ansprechen. Umgekehrt hat jedoch die Klasse A keine Rechte an der Klasse B, diese müßten in der Klasse B erst explizite vergeben werden.

Eine `friend`-Deklaration ist nur in der Definition der Klasse möglich, eine nachträgliche Deklaration außerhalb der Klassendefinition ist nicht erlaubt.

3.2 Statische Klassenelemente

Es kann Fälle geben, in denen Klassenelemente zwar Teil des Gesamtkonzeptes einer Klasse sind, aber dennoch keinem einzelnen Datenobjekt zuzuordnen sind, d.h. für alle Objekte gleich verfügbar oder global innerhalb einer Klasse sein sollen. Im Prinzip könnte man dazu einfach eine globale Variable oder Funktion erfüllen, die Anforderungen wären damit schon erfüllt. Allerdings ist eine solche globale Variable bzw. Funktion nicht vor Zugriffen außerhalb der Klasse geschützt. Um das zu verhindern, kann ein Klassenelement als `static` deklariert werden: Dann existiert es „global", d.h. ist nicht einem bestimmten Datenobjekt zugeordnet, gehört aber dennoch dem Gültigkeitsbereich der Klasse an, mit allen damit verbundenen Schutzmechanismen und Rechten:

```
class   A
{
        int     local_i;
static  int     global_i;

public: void    increment(int i)
        {
        local_i += i;
                global_i += i;
        }

        friend  void    print_global();
};

int     A::global_i = 0;

void    print_global()
{
        printf("Gesamter Wert:  %d\n", A::global_i);
}
```

Da eine `static`-Deklaration eines Klassenelementes noch keine Definition darstellt, muß diese später in irgendeinem Modul nachgeholt werden (während die Klassendefinition üblicherweise in einer Headerdatei steht). Im obigen Beispiel ist die Variable `A::global_i` vor Zugriffen von außen geschützt, nur Element- und `friend`-Funktionen können auf sie zugreifen. Die eigentliche Datenstruktur `class A` enthält jedoch nur einen `int` (`sizeof(A) == sizeof(int)`). Gleiches gilt auch für statische Elementfunktionen. Alternativ könnte man `print_global()` als statische Elementfunktion `A::print_global()` definieren, damit hat sie ebenfalls die vollen Zugriffsrechte auf alle Klassenelemente (kurze ÜBUNGSAUFGABE).

3.3 Programmieraufgabe: Doppelt verkettete Listen

Es kommt nicht selten vor, daß man mit mehreren Objekten arbeiten muß, ohne daß deren Anzahl von anfang an bekannt wäre. Die Anordnung in einem statischen Feld scheidet daher aus, wenn die einzelnen Objekte zudem unterschiedlich groß sind und/oder ihre Größe jederzeit ändern können,

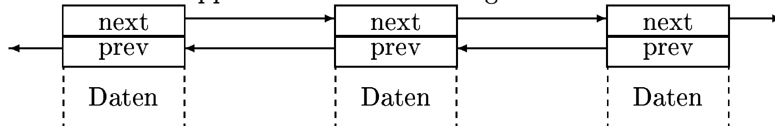
ist auch ein zusammenhängender dynamischer Speicherbereich nicht mehr anwendbar. Typisches Beispiel ist ein Editor, der jede Zeile einer Datei in den Speicher lädt. Jede Zeile kann dabei unterschiedlich lang sein und ihre Länge jederzeit während des Editiervorganges ändern. Auch die Reihenfolge der Zeilen kann sich jederzeit ändern, da ja Zeilen kopiert und verschoben werden können.

An dieser Stelle soll eine Möglichkeit vorgestellt werden, wie diese Aufgabenstellung gehandhabt werden kann. Annahme sei, daß ein Programm mit einer unbekannten Menge gleichartiger Objekte arbeiten muß, deren Reihenfolge sich jederzeit ändern kann.

Hierzu bieten sich die doppelt verketteten Listen an. Jeder Eintrag einer doppelt verketteten Liste enthält neben den eigentlichen Daten, beispielsweise einer Zeile, einen Zeiger auf das nächste und das vorherige Element. Dadurch ist es möglich, von einem Element ausgehend, die anderen Elemente in beiden Richtungen zu erreichen:

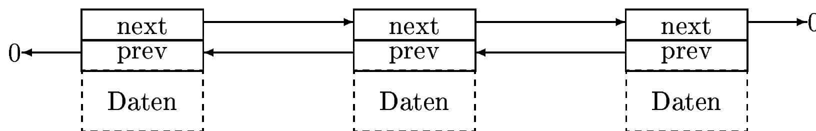
```
class   dlist
{
        dlist   *next, *prev;
         ...
};
```

Generell hat eine doppelt verkettete Liste folgende Struktur:

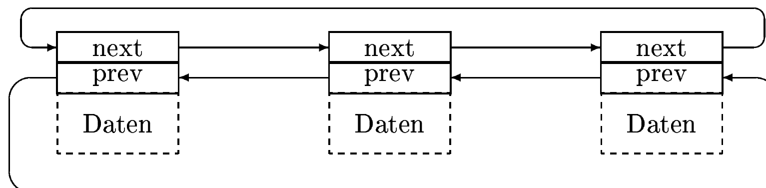


Die einzelnen Objekte hängen somit hintereinander und können über ihre `next`- und `prev`-Zeiger angesprochen werden. Unerklärt ist noch, wie man eine solche Liste abschließt. Dazu gibt es zwei Möglichkeiten:

- **Den Abschluß mittels 0-Zeiger (offene Liste):** Dasjenige Element, das als `next`-Zeiger eine 0 enthält, ist das letzte in der Liste, das mit einer 0 im `prev`-Zeiger das erste. Wenn man alle Elemente der Liste nacheinander ansprechen will, muß man vom ersten Element ausgehend die folgenden Elemente durchgehen, solange bis ein 0-Zeiger das Ende anzeigt.



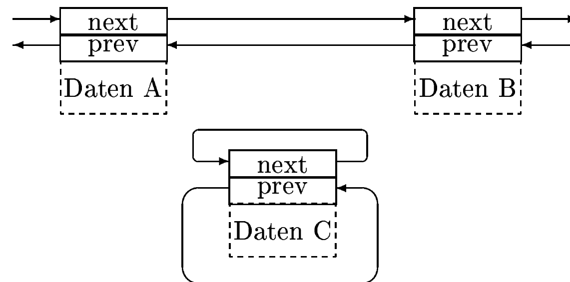
- **Die zirkulare Liste (geschlossene Liste):** Eine zirkulare Liste hat keinen Anfang und kein Ende. Es gibt nur ein Startelement, das durch die Listenstruktur nicht festgelegt wird, und von dem aus man sich durch die Liste hangeln kann, bis der `next`-Zeiger wieder auf das Startelement zeigt.



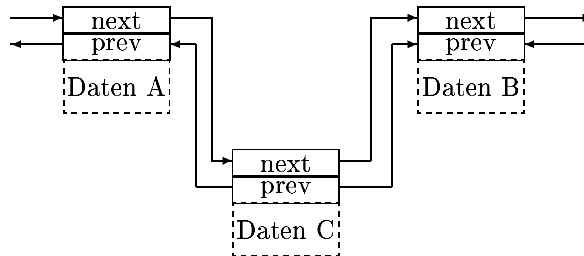
Vergleich beider Methoden: Ein listenexterner Zeiger ist sowohl bei der geschlossenen als auch bei der offenen Liste notwendig, um zumindest ein Element ansprechen zu können. Obwohl es bei der offenen Liste günstig ist, wenn dieser auf das Anfangselement zeigt, ist dies nicht unbedingt erforderlich, da man ja Anfang und Ende jederzeit ermitteln kann. Bei der geschlossenen Liste ist

dies hingegen völlig gleichgültig, da kein Element gegenüber den anderen besonders ausgezeichnet ist.

Wie wird nun ein Element in die Liste eingetragen? Der Zustand vor dem Eintragen sieht folgendermaßen aus:



Das neu einzutragende Element C ist noch keiner Liste zugeordnet bzw. die Liste besteht aus einem einzigen Element; in diesem Falle handelt es sich um eine geschlossene Liste, der `next`- und der `prev`-Zeiger deuten dabei auf „sich selbst“. Bei einer offenen Liste würden beide Zeiger 0 enthalten. Nach dem Eintragen in die Liste zwischen die Elemente A und B soll der `next`-Zeiger von A auf C und derjenige von C auf B zeigen, analog die `prev`-Zeiger. Dieser Zustand sieht dann so aus:



Die programmtechnische Realisierung kann nun wie folgt geschehen:

```
// Elementfunktion zur Initialisierung einer Listenstruktur
void    dlist::init_list()
{
        next = prev = this;      // Die Struktur bildet selbst eine
                                 // vollständige geschlossene Liste.
}
```

Aufgabe von `init_list()` (wird vorzugsweise von den Konstruktoren aufgerufen) wäre es somit, die Listenstruktur zu initialisieren (am Beispiel einer geschlossenen Liste):

```
// Elementfunktion zum Einhängen in eine Liste
void    dlist::inlist(dlist*dl)
{
        next = dl->next;         // Aktuelle Struktur wird nach
        prev = dl;               // der Struktur *dl in die Liste
        next->prev = this;       // eingefügt. Zustand nachher:
        prev->next = this;       // dl->next ≡ this, prev ≡ dl
}

// Elementfunktion zum Austragen aus einer Liste
void    dlist::exlist()
{
```

```
        prev->next = next;
        next->prev = prev;
}
```

Spätestens vom Destruktor wird `exlist()` aufgerufen.

Um nun alle Datenelemente einer Liste durchzugehen, muß von einem Anfangszeiger, der „Wurzel“, ausgehend jedes Listenelement der Reihe nach angesprochen werden, bis wieder das Ausgangselement erreicht wird:

```
dlist*element = root;
        do
        {
                element->print();
                element=element->next;
}
        while(element!=root);
```

Dabei ist `root` ein Zeiger auf ein `dlist`-Objekt, das den Anfang der Liste bildet. `root` ist vorzugsweise ein statisches Klassenelement, da für alle Listenelemente nur eine einzige, die gleiche Wurzel, existiert (außer man möchte mehrere Listen behandeln, was aber hier nicht der Fall sein soll). Damit ist `root` sinnvollerweise auch privat, damit dieser Zeiger niemals unsinnige Werte haben kann. Das obige C++ -Fragment könnte dann etwa Teil einer öffentlichen statischen Elementfunktion `dlist::print_all()` sein, die für jedes Listenelement eine Elementfunktion `print()` aufruft. Diese wiederum könnte jedes Datenelement am Bildschirm ausgeben.

Ein Hauptprogamm, das die Funktionsweise der Listenklasse demonstriert, könnte dann folgendermaßen aussehen:

```
main()
{
dlist   A("Objekt A"), B("Objekt B");

        A.set_root();
        dlist::print_all();

        B.inlist(&A);
        dlist::print_all();

        {
        dlist   C("Objekt C");
                C.inlist(&A);
                dlist::print_all();
        }
        dlist::print_all();

        return 0;
}
```

Dabei ist `dlist::set_root()` eine Elementfunktion, die den `root`-Zeiger auf das aktuelle Objekt setzt. Alternativ könnte man auch **alle** Objekte einer Klasse in eine zusammenhängende Liste eintragen. In diesem Fall könnte der `root`-Zeiger bereits im Konstruktor gesetzt werden, ein

explizites Setzen wäre dann nicht mehr nötig. Auch das Eintragen in die Liste (`dlist::inlist()`) könnte dann bereits im Konstruktor erfolgen.

Wenn das Programm korrekt ist, werden mit dem letzten Aufruf von `dlist::print_all()` nur noch die beiden Objekte A und B ausgegeben, da das Objekt C nur innerhalb des Programmblockes existiert. Im konkreten Fall besteht der Datenbereich der doppelt verketteten Liste aus einem einfachen String.

Zu beachten wäre noch, daß der Destruktor überprüfen müßte, ob der `root`-Zeiger nicht gerade auf das aktuell zu vernichtende Objekt zeigt. Falls ja, müßte der `root`-Zeiger auf das nächste Listenelement gesetzt werden bzw., falls die Liste bereits leer ist, auf `NULL`.

3.4 Zusammenfassung und Begriffsbestimmungen:

- Die Verbindung von Datenelementen und Elementfunktionen zu einer Einheit bezeichnet man als *Klasse*. Eine Klasse enthält private und öffentliche Elemente. Die Zugriffsrechte können auf `friend`-Funktionen und `friend`-Klassen ausgedehnt werden. Nur in der Klasse selbst kann entschieden werden, wer auf die Elemente zugreifen darf, eine nachträgliche Vergabe von Zugriffsrechten ist nicht möglich.

- *Ein* **Objekt** *ist eine Variable vom Typ einer Klasse.*
  Vergleiche:

```
int     i;      // i ist eine Variable und vom Typ int
Targa   tga;    // tga ist ein Objekt und vom Typ class Targa
```

  Im Prinzip könnte man auch Grundtypen wie `int` als Klasse bezeichnen und infolgedessen Variablen als Objekte. Allerdings weist eine Klasse noch gewisse essentielle Eigenschaften auf (Fähigkeit zur Vererbung, wird in einem späteren Abschnitt behandelt), die den Grundtypen fehlen. Aufgrund dieses Unterschiedes wird C++ als *hybride Sprache* bezeichnet. In einer vollständig objektorientierten Sprache gibt es diese Unterscheidung nicht.

- *Eine* **Struktur** *ist eine Klasse, in der alle Elemente öffentlich zugänglich sind.* In C++ ist der Begriff der Struktur über den Begriff der Klasse definiert. Der eigentliche Grundtyp ist die `class`, `struct` tritt nur als abgeleiteter Typ auf (und dient der Abwärtskompatibilität zu C):

```
struct X { ... }; := class X { public: ... };
```

- Mithilfe des Schutzkonzeptes können die Elemente einer Klasse vor dem Zugriff von außen geschützt werden. Ein Objekt erscheint damit nach außen hin als ein Ganzes und ist nur über seine öffentlichen Elemente zugänglich. Diese Methodik bezeichnet man auch als *Einkapselung* (engl. *encapsulation*).

4 Datenabstraktion - Überladen von Operatoren

Hier sind Drachen!
(Alte Landkarte)[3]

AUFGABENSTELLUNG:
Es soll ein Programm erstellt werden, das mit dem Verfahren des „Raytracing“ eine beliebige Anzahl von Kugeln graphisch darstellt.

SCHLÜSSELBEGRIFFE:
`operator`, Operatorfunktionen, Zuordnungsreihenfolge und Rangfolge, Ausgabe selbstdefinierter Typen, Konversionsoperatoren, implizite Operatorfunktionen.

Der Untertitel, den auch Bjarne Stroustrup für dieses Kapitel gewählt hat, ist sicher nicht unpassend, denn mit der Möglichkeit, die Bedeutung eines Operators neu definieren zu können, stellt C++ eines der mächtigsten Werkzeuge zur Verfügung. Mit überladenen Operatoren können auch extrem komplexe Problemstellungen einfach und leicht verständlich formuliert werden. Ein ideales Beispiel für die Anwendung überladener Operatoren ist die Vektorrechnung im $\mathbb{R}^3$.

Basis der Vektorrechnung ist eine Struktur (bzw. Klasse), die die drei Vektorkomponenten enthält:

```
struct  vector3
{
        double  x,y,z;
};
```

Um zwei Vektoren zu addieren, würde man in einfachem C++ (bzw. C, wenn man die Referenzen durch Zeiger ersetzt) eine Funktion wie die folgende schreiben:

```
void    add(vector3&result, const vector3&a, const vector3&b)
{
        result.x = a.x + b.x;
        result.y = a.y + b.y;
        result.z = a.z + b.z;
}
```

Die erste Problematik entsteht daraus, daß alle Parameter prinzipiell gleichberechtigt sind und nur aus reiner Willkür das Resultat im ersten Parameter zurückgeliefert wird – in Anlehnung daran, daß auch in einer Zuweisung der **links** stehende Wert verändert wird. Im Gegensatz dazu wird aber oft in mathematischen Ausdrücken das Ergebnis auf der **rechten** Seite ausgegeben – mit der gleichen Berechtigung könnte man daher das Ergebnis auch im letzten Argument liefern. (Vgl. in diesem Zusammenhang die beiden Libraryfunktionen `memcpy()` und `bcopy()`.)

Erschwerend kommt hinzu, daß komplexere Ausdrücke wie $\vec{v} = \vec{a} + \vec{b} + \vec{c} + \vec{d}$ mit der obigen Funktion nicht direkt formuliert werden können – irgendwann geht dann die Übersicht verloren.

KLEINE ÜBUNGSAUFGABE als Fingertraining: Addiere unter Verwendung dieser Funktion vier Vektoren.

[3] Aus: Bjarne Stroustrup, Die C++ Programmiersprache, erste Edition (Deutsch)

Beide Nachteile können vermieden werden, wenn das Ergebnis als **Funktionswert** zurückgegeben wird:

```
vector3 add(const vector3&a, const vector3&b)
{
vector3 result;

        result.x = a.x + b.x;
        result.y = a.y + b.y;
        result.z = a.z + b.z;

        return result;
}
```

Damit kann nun die mehrfache Addition in einem geschrieben werden:

```
vector3 v,a,b,c,d
        v = add(a,add(b,add(c,d)));
```

Wie zuvor, so ist auch diese Variante in C möglich, wenn man die Referenzen durch Zeiger ersetzt. Allerdings ist sie in C unüblich, da bei jedem Funktionsaufruf eine temporäre Variable erzeugt und bei der Wertrückgabe eine Struktur kopiert werden muß. Dies ist recht ineffizient. In C++ stellt das hingegen kein so fundamentales Problem dar, denn `add()` kann hier als `inline`-Funktion definiert werden – der Compiler ist damit imstande, den Funktionscode direkt an der Aufrufposition einzusetzen, alle temporären Variablen wegzuoptimieren und ggf. sogar die drei Vektorkomponenten separat zu behandeln. Ob er das auch wirklich macht, hängt von der Qualität des Compilers ab, auf jeden Fall aber hat er alle dazu notwendigen Informationen. Formal enthält die Funktion `add()` auch als `inline`-Funktion eine lokale Variable, im tatsächlichen Code kann sie jedoch wegoptimiert sein.

Die Funktion `add()` läßt sich in C++ natürlich ebenso als Elementfunktion ausführen, womit der Aufruf abermals übersichtlicher wird:

```
struct  vector3
{
        double  x,y,z;

        vector3 add(const vector3&v)            // nur ein Parameter !
        {
        vector3 result;

                result.x = x + v.x;
                result.y = y + v.y;
                result.z = z + v.z;

                return result;
        }
};
         ...
vector3 v,a,b,c,d;
        v = a.add(b).add(c).add(d);
```

Als Elementfunktion ist nur ein Parameter nötig, denn der andere Vektor ist jeweils implizit über `this` verfügbar.

Diese Version ist in C bereits nicht mehr möglich, aber C++ bietet noch mehr: Anstatt die Funktion `add()` zu nennen, kann sie als `operator+()` bezeichnet werden:

```
struct  vector3
{
        double  x,y,z;

        vector3 operator+(const vector3&v)
        {
        vector3 result;

                result.x = x + v.x;
                result.y = y + v.y;
                result.z = z + v.z;

                return result;
        }
};
```

Vom Namen abgesehen, verläuft der Aufruf wie zuvor:

```
vector3 v,a,b,c,d;
        v = a.operator+(b).operator+(c).operator+(d);
```

Würde man allerdings diese explizite Schreibweise verwenden müssen, brächte sie selbstverständlich keine Vorteile. Stattdessen kann aber für alle Operatorfunktionen auch die implizite Schreibweise verwendet werden:

```
vector3 v,a,b,c,d;
        v = a+b+c+d;
```

So erst entfalten die Operatorfunktionen ihre volle Leistungsfähigkeit. Generell läßt sich jeder Aufruf einer Operatorfunktion für einen Operator „`@`“ abkürzen:

$$\texttt{X.operator@(Y)} \equiv \texttt{X@Y}$$

Dabei gilt es allerdings noch einige Dinge zu beachten:

- Unäre Operatoren müssen ohne Parameter definiert werden, da sie nur **ein** Argument aufweisen, das durch `this` bereits zur Verfügung steht. Binäre Operatoren haben, wie bereits gezeigt, ein einziges Argument; der einzige ternäre Operator `?:` kann nicht überladen werden.
- Der `.`-Operator (Zugriff auf ein Klassenelement) kann nicht überladen werden. Das ist unmittelbar einsichtig, da die explizite Schreibweise dann in eine unendliche Rekursion münden würde. Aus dem gleichen Grund kann auch der Bereichtsoperator `::` nicht überladen werden.
- Operatorfunktionen können, wie alle anderen Funktionen auch, polymorph sein:

```
struct  vector3
{
         ...

        // Multiplikation Vektor mit Skalar w = v · λ
        vector3 operator*(double skalar);

        // Skalarprodukt zweier Vektoren λ = v · w
        double operator*(const vector3&);
};
```

- Mit dem Überladen eines Operators wird lediglich seine `Bedeutung` geändert, nicht aber seine sonstigen Eigenschaften wie Rangfolge, Zuordnungsreihenfolge und Auswertungsreihenfolge. Es können außerdem zu den bereits vorhandenen Operatoren keine neuen hinzu definiert werden. Während die Rangfolge bereits in C eine einsichtig bedeutsame Rolle spielt, wird in C++ zudem die Zuordungsreihenfolge relevant, denn sie bestimmt die Reihenfolge, in der die Operatorfunktionen aufgerufen werden.

  Bei von **links zuordnenden** Operatoren (z.B. „+") werden die Operatorfunktionen von **links** her aufgerufen:

  ```
  a+b+c;
  ```

  In expliziter Schreibweise:

  ```
  (a.operator+(b)).operator+(c);
  ```

  Also: Der erste Funktionsaufruf erfolgt für das Objekt `a`, der zweite für den Funktionswert.

  Bei von **rechts zuordnenden** Operatoren (z.B. „=") werden die Operatorfunktionen von **rechts** her aufgerufen:

  ```
  a=b=c;
  ```

  In expliziter Schreibweise:

  ```
  a.operator=(b.operator=(c));
  ```

  Hier ist zuerst das Objekt `b` involviert, danach `a`; der erste Funktionsaufruf erfolgt für das Objekt `b`, der zweite für `a`.

  Das Objekt `c` spielt bei dieser Betrachtung keine Rolle, da von ihm keine Elementfunktion aufgerufen wird, es verhält sich „passiv" und wird nur als Parameter, nicht als „aktives" Objekt, gebraucht.

- Wird ein Operator überladen, so heißt dies nicht, daß seine ursprüngliche Bedeutung verlorengeht. Seine Bedeutung wird nur zusätzlich auf eine bestimmte Klasse von Objekten erweitert, für die er vorher nicht definiert war (die Addition von Strukturen ist normalerweise nicht möglich). Tritt in einem Ausdruck eine Kombination von verschiedenen Objekttypen auf, werden auch verschiedene Operatorfunktionen aufgerufen:

```
vector3 v,w;
double  d,e;
        e = v*w*d;
```

  Mit den obigen Definitionen ergibt der Ausdruck `v*w` einen `double` (Skalarprodukt zweier Vektoren), sodaß für die Multiplikation mit der Variablen `d` der intrinsische Multiplikationsoperator für `double`-Variablen genommen wird. Das Ergebnis ist wieder vom Typ `double`. Um zu verdeutlichen, welcher Operator wann aufgerufen wird, ist es oftmals günstig, sich die explizite Schreibweise zu veranschaulichen:

```
e = (v.operator*(w))*d;
```

  `(v.operator*(w))` liefert einen `double`.
  ANMERKUNG: Auch hier ist die Zuordnungsreihenfolge sehr wichtig. Der Operator „`*`“ ordnet von links zu, `v*w*d` bedeutet also `(v*w)*d`. Der Ausdruck `v*(w*d)` würde in expliziter Schreibweise völlig anders aussehen:

```
v.operator*(w.operator*(d));
```

  Hier wird also **zweimal** eine Elementfunktion aufgerufen: zuerst wird ein Vektor mit einem Skalar multipliziert, das Ergebnis ist ein Vektor. Dieser wird mit einem anderen Vektor multipliziert, was (zufälligerweise) wieder einen Skalar liefert. In diesem Fall ist das Ergebnis also identisch. Allerdings sind Vektoroperationen dreimal langsamer (da jede Rechnung mit drei Komponenten ausgeführt werden muß): Der Ausdruck `v*(w*d)` ist infolgedessen um einiges ineffizienter als `(v*w)*d` ...

- Manche Operatoren treten in mehreren Bedeutungen auf, beispielsweise „`&`“ sowohl unär (als Adreßoperator) als auch binär (als bitweises Und). Beide Operatorfunktionen können definiert werden und beeinflussen sich gegenseitig nicht:

```
class   vector3
{
         ...
        void operator&();                // Adreßoperator überladen
        void operator&(const vector3&); // bitweises Und überladen
};
```

  Anmerkung: Dieses Codefragment ist nur als Demonstration gedacht. Es ist nicht besonders sinnvoll, den Adreßoperator zu überladen; was ein bitweises Und bei Vektoren bedeuten soll, ist ebenfalls fraglich.

- Die Zusammenhänge zwischen den Operatoren bei den intrinsischen Typen sind nicht automatisch auch bei überladenen Operatoren vorhanden. Ist in der Klasse `vector3` der Operator „`+`“ definiert, folgt nicht sofort, daß der Operator „`+=`“ als „`v+=w`“ := „`v=v+w`“ definiert ist. Der Operator „`+=`“ muß explizit definiert werden, damit er verfügbar wird.
- Ein Operator kann alternativ zur Elementfunktion auch als `friend`-Funktion definiert werden. Der implizite `this`-Parameter fällt dann weg und muß explizit angegeben werden. Unäre Operatorfunktionen haben als `friend`-Funktion daher einen Parameter, binäre Operatorfunktionen zwei:

```
class   vector3
{
        double  x,y,z;
         ...
        friend vector3 operator+(const vector3&v,const vector3&w)
        {
        vector3 result;
                result.x = v.x + w.x;
                result.y = v.y + w.y;
                result.z = v.z + w.z;
                return result;
        }
};
```

Der Aufruf einer `friend`-Operatorfunktion läßt sich nach der folgenden Regel für einen binären Operator „`@`“ abkürzen:

$$\texttt{operator@(X,Y)} \equiv \texttt{X@Y}$$

Der Aufruf des wie oben definierten Operators „`+`“ ist damit vollkommen identisch mit der Version als Elementfunktion – *eine Operatorfunktion kann daher* **entweder** *als Elementfunktion* **oder** *als Friendfunktion definiert werden! Beide Varianten sind gleichzeitig nicht möglich.*. Welche Variante günstiger ist, läßt sich allgemein nicht sagen und muß von Fall zu Fall entschieden werden. Im aktuellen Beispiel der Vektorklasse sind beide gleichberechtigt.

Es gibt jedoch Situationen, in denen es **notwendig** ist, eine Operatorfunktion nicht als Elementfunktion zu definieren, zum Beispiel bei der Multiplikation Skalar mal Vektor:

```
vector  v,w;
double  d;
        v = d*w;
```

In expliziter Schreibweise würde der Funktionsaufruf als Elementfunktion bzw. `friend`-Funktion folgendermaßen aussehen:

```
v = d.operator*(w);        (1)
v = operator*(d,w);        (2)
```

Variante (1) ist **nicht** möglich, da ein **double** keine Klasse[4] darstellt, von der aus eine Elementfunktion aufgerufen werden kann. Bleibt also nur Variante (2), die die Definition des Operators `*` als `friend`-Funktion voraussetzt:

[4] Zur Wiederholung: C++ ist eine *hybride* Sprache.

```
class   vector3
{
         ...
        friend vector3 operator*(double d, const vector3&v);
};
```

Anmerkung: Bei der Funktion `operator*(double,vector3)` handelt es sich um eine **andere** Funktion als etwa `operator*(vector3,double)` (Polymorphie!). Die Definition des einen Operators schließt daher die Definition des anderen nicht aus. `operator*(vector3,double)` kann gleichermaßen als Elementfunktion oder als `friend`-Funktion definiert werden, dies allerdings ausschließlich.

- Der Operator „=“ stellt einen Spezialfall dar: er ist **implizit** für jede Klasse als komponentenweise Zuweisung definiert. Wird er **explizit** definiert, *muß* er als Elementfunktion definiert werden.

```
struct  vector3
{
        double x,y,z;

        // Diese Operatorfunktion ist bereits implizit definiert:
        void    operator=(const vector3&v)
        {
                x = v.x;
                y = v.y;
                z = v.z;
        }
};
```

Im Fall der Vektorklasse ist es nicht notwendig, den „=“-Operator neu zu definieren. Eine Neudefinition hat immer dann Sinn, wenn die Zuweisung nicht komponentenweise erfolgen soll.

- Es ist nicht notwendig, daß eine Operatorfunktion, die keine Elementfunktion darstellt, `friend` einer Klasse ist, obwohl dies im allgemeinen der Fall sein wird. Sind alle notwendigen Klassenelemente öffentlich, kann eine Operatorfunktion auch **nachträglich**, außerhalb der Klasse definiert werden:

```
struct  vector3
{
        double x,y,z;
};

inline  vector3 operator+(const vector3&v, const vector3&w)
{
         ...
}
```

Für eine solche globale Operatorfunktion gelten die gleichen Regeln wie für `friend`-Funktionen (im Prinzip sind `friend`-Funktionen ebenfalls global, auch wenn sie innerhalb einer Klasse definiert werden).

Die „nachträgliche“ Definition einer Operatorfunktion wird unter anderem recht häufig verwendet, um für die Klasse `ostream` weitere Ausgabetypen zu definieren:

```
#include <iostream.h>

struct  vector3
{
        double x,y,z;
};

ostream& operator<<(ostream&os, const vector3&v)
{
        os << '(' << v.x << ',' << v.y << ',' v.z << ')';
        return os;
}
```

Die Ausgabe eines Vektors erfolgt damit vollkommen parallel zu den intrinsischen Standardtypen:

```
vector3 v;

        cout << "Der Vektor v hat den Wert " << v << "...\n";
```

Die Möglichkeit, selbstdefinierte Typen bei der Ausgabe gleich wie intrinsiche Typen behandeln zu können, stellt einen der größten Vorzüge der Stream-Klassen gegenüber der C-Ausgabe mit `printf()` dar.

Analog kann auch für die Eingabeklasse `istream` eine Operatorfunktion

```
istream&operator>>(istream&is, vector3&v)
```

definiert werden. Der Parameter `v` ist dabei nicht mehr konstant, da das Argument ja durch die Eingabe verändert werden soll.

- Der Operator „Funktionsaufruf“ kann ebenfalls überladen werden. Als einziger Operator kann er **beliebig** viele Parameter enthalten:

```
struct  vector3
{
         ...
        // Was auch immer dies bedeuten mag ...
        double operator()(vector3&a,vector3&b,vector3&c);
};
```

Der Aufruf erfolgt dann, als ob das Objekt eine Funktion wäre:

```
vector3 v,a,b,c;

        v(a,b,c);        // explizite: v.operator()(a,b,c);
```

Sinnvoll wäre dies beispielsweise, wenn `v` eine 3-Form darstellt, der man drei Vektoren übergibt und die eine Zahl zurückliefert.

Selbstverständlich kann auch diese Operatorfunktion polymorph sein.

Ein paar Worte noch zum Konstruktor: Mithilfe des Konstruktors kann jederzeit ein Objekt erstellt werden. Enthält die Klasse `vector3` folgende Elementfunktionen:

```
struct  vector3
{
         ...
        vector3(double x, double y, double z);  // Konstruktor
        vector3 operator+(const vector3&v);
};
```

dann sind auch Ausdrücke wie der folgende möglich:

```
vector3 a,b;
        a = b + vector3(1,0,0);
```

Wird ein polymorpher Konstruktor `vector3::vector3(double)` definiert, kann ein `double` direkt als `vector3` verwendet werden:

```
vector3 a,b;
        a = b + 1.0;      // für das zweite Argument wird
                          // vector3::vector3(double) aufgerufen.
```

Dasselbe läßt sich auch erreichen, wenn für den zuerst genannten Konstruktor Defaultparameter definiert werden:

```
struct  vector3
{
         ...
        vector3(double x = 0.0, double y = 0.0, double z = 0.0);
         ...
};
```

Damit kann jeder `double` als `vector3` verwendet werden, nicht angegebene Komponenten werden mit 0 aufgefüllt.

```
vector3 a,b;
        a = b + 1.0;                // impliziter Aufruf des Konstruktors
        a = b + vector3(1.0);       // expliziter Aufruf
        a = b + vector3(1.0,2.0);
        a = b + vector3(1.0,2.0,3.0);
```

Der umgekehrte Weg `vector3` → `double` kann mithilfe der *Konversionsoperatoren* ermöglicht werden:

```
struct  vector3
{
        double  x,y,z;
         ...

        // Konversionsoperator vector3 nach double
        operator double()
        {
                return x+y+z;    // oder was sonst an dieser
                                 // Stelle sinnvoll ist ...
        }
};
```

Damit kann in Situationen, in denen ein `double` erwartet wird, ein `vector3` übergeben werden:

```
void    f(double);
         ...
vector3 v;
        f(v);    // impliziter Aufruf von vector3::operator double()
```

WARNUNG: Aus der Tatsache, daß diese Möglichkeiten hier vorgestellt werden, folgt nicht, daß sie in dieser Form auch sinnvoll sind. Welche Operatorfunktionen und welche Konstruktoren zweckmäßig sind, bleibt als Übungsaufgabe überlassen.

4.1 Die Klasse `vector3`

Neben diesen kleinen Erläuterungen zum Überladen von Operatoren nun aber zur eigentlichen Aufgabenstellung dieses Abschnittes. Wesentlich ist dabei, daß vor allem die Klassenstruktur möglichst vollständig ausgebaut wird, das eigentliche Hauptprogramm ergibt sich dann nahezu trivial. Dazu gehört auch, daß die Klassendefinition in ein separates Headerfile (Vorschlag: `vector3.hpp`) verlegt wird und notwendige nicht-`inline` -Funktionen in einem eigenen Modul (Vorschlag: `vector3.cpp`) abgelegt werden.

Die zur Verfügung gestellten Vektorfunktionen sollten intuitiv erfaßbar sein. Dabei muß man auch die Erwartungshaltung beachten: Von einer **Elementfunktion** `unit()` würde man erwarten, daß sie das Objekt selbst ändert, von einer **`friend`-Funktion** hingegen eher, daß das Objekt selbst unverändert bleibt, aber einen Einheitsvektor zurückliefert:

```
vector3 v, w;
         ...
        v.unit();         // mache aus v⃗ einen Einheitsvektor
         ...
        w = unit(v);      // w⃗ ist der Einheitsvektor in Richtung v⃗
```

4.1.1 Rechenregeln mit Vektoren im $\mathbb{R}^3$

Ein Vektor $\vec{v} \in \mathbb{R}^3$ besteht aus drei Komponenten (v_x, v_y, v_z). Addition und Subtraktion von Vektoren sowie Multiplikation mit bzw. Division durch Skalare (Zahlen) sind komponentenweise

definiert. Daneben liefert die *Skalarprodukt* zweier Vektoren $\vec{v}$, $\vec{w}$ einen Skalar λ:

$$\lambda = \vec{v} \cdot \vec{w} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \cdot \begin{pmatrix} w_x \\ w_y \\ w_z \end{pmatrix} = v_x \cdot w_x + v_y \cdot w_y + v_z \cdot w_z$$

und die *Vektorprodukt* (das „Kreuzprodukt“) zweier Vektoren $\vec{v}$, $\vec{w}$ einen Vektor $\vec{u}$:

$$\vec{u} = \vec{v} \times \vec{w} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \times \begin{pmatrix} w_x \\ w_y \\ w_z \end{pmatrix} = \begin{pmatrix} v_y \cdot w_z - v_z \cdot w_y \\ v_z \cdot w_x - v_x \cdot w_z \\ v_x \cdot w_y - v_y \cdot w_x \end{pmatrix}$$

Über das Skalarprodukt ist der *Absolutbetrag* (die *Länge*) eines Vektors definiert:

$$|\vec{v}| := \sqrt{\vec{v} \cdot \vec{v}}$$

und damit der *Einheitsvektor* $\vec{v}_u$ in Richtung eines Vektors $\vec{v}$:

$$\vec{v}_u = \frac{\vec{v}}{|v|}$$

Das Skalarprodukt zweier Einheitsvektoren $\vec{v}_u$, $\vec{w}_u$ liefert den *Cosinus des eingeschlossenen Winkels* α:

$$\cos\alpha = \vec{v}_u \cdot \vec{w}_u$$

4.2 Das Raytracing - Verfahren

Das Raytracing-Verfahren ist das mächtigste Verfahren der Computergraphik. Auch die komplexesten und unmöglichsten Gebilde können hiermit dargestellt werden. Dabei ist das Grundprinzip ausgesprochen einfach: Das Auge eines Beobachters und ein Punkt auf dem Bildschirm definieren einen Sichtstrahl, eine Gerade. Aufgabe des Raytracing-Verfahrens ist es nun, den Schnittpunkt dieser Geraden mit verschiedenen Objekten zu berechnen. Das einfachste Objekt dabei ist die Kugel. Mithilfe der oben erstellten Vektorklasse läßt sich die Grundstruktur eines Raytracingprogrammes in wenigen Zeilen erstellen.

Die Bildschirmebene wird durch den Mittelpunkt des Bildschirms, $\vec{B}$, und zwei Richtungsvektoren $\vec{X}$ und $\vec{Y}$, die jeweils der waagrechten bzw. senkrechten Kante entsprechen, bestimmt. Ein Beobachter am Standpunkt $\vec{S}$ ergibt sich somit als Sichtstrahl:

$$\vec{g}(\lambda) = \vec{S} + \lambda \left[\vec{B} - \vec{O} + x\vec{X} + y\vec{Y} \right]$$

Dabei sind x, y die Bildschirmkoordinaten, jeweils im Bereich $[-1, +1]$ und λ der *Strahlparameter*. Er gibt an, wie weit ein bestimmtes Objekt vom Beobachter entfernt ist. Generell hat ein Sichtstrahl die Form

$$\vec{g}(\lambda) = \vec{O} + \lambda\vec{d}$$

wobei $\vec{O}$ der *Ursprung* und $\vec{d}$ die *Richtung* des Strahls ist. Für alle Werte von λ gibt $\vec{g}(\lambda)$ Punkte auf der Geraden g an.

Eine Kugel ist die Menge aller Punkte, die von einem Mittelpunkt $\vec{M}$ den gleichen Abstand, den *Radius* R haben. Ein Punkt $\vec{P}$ liegt daher auf der Kugel, wenn gilt:

$$|\vec{P} - \vec{M}| = R$$

bzw. wenn gilt (das Quadrat der obigen Gleichung muß ebenfalls erfüllt sein):

$$\left(\vec{P} - \vec{M}\right)^2 = R^2$$

Nun ist

$$\vec{P} \equiv \vec{g}(\lambda) = \vec{O} + \lambda\vec{d}$$

Eingesetzt in obige Gleichung ergibt sich

$$\left(\vec{O} + \lambda\vec{d} - \vec{M}\right)^2 = R^2$$

gleichbedeutend mit

$$\left(\left[\vec{O} - \vec{M}\right] + \lambda\vec{d}\right)^2 = R^2$$

Nach Ausquadrieren der linken Seite folgt

$$\left[\vec{O} - \vec{M}\right]^2 + 2\lambda\left[\vec{O} - \vec{M}\right] \cdot \vec{d} + \lambda^2\vec{d}^2 = R^2$$

Dies ist eine quadratische Gleichung für den gesuchten Strahlparameter λ der Struktur

$$\lambda^2 + 2p\lambda + q = 0$$

mit

$$p = \frac{\left[\vec{O} - \vec{M}\right] \cdot \vec{d}}{\vec{d}^2}$$

$$q = \frac{\left[\vec{O} - \vec{M}\right]^2 - R^2}{\vec{d}^2}$$

Die beiden Lösungen dieser quadratischen Gleichung sind

$$\lambda_1 = -p - \sqrt{p^2 - q}$$

$$\lambda_2 = -p + \sqrt{p^2 - q}$$

Die Gleichung hat *keine* Lösung, wenn der Ausdruck unter der Wurzel (sog. „Diskriminante") kleiner als 0 ist, d.h. der Sichtstrahl trifft die Kugel nicht, es gibt keinen Schnittpunkt und der Beobachter schaut an der Kugel vorbei. Wenn die Diskriminante genau 0 ist, sind die beiden Lösungen identisch, es gibt genau einen Schnittpunkt, der Beobachter sieht also genau den Rand der Kugel. Falls die Diskriminante größer als 0 ist, gibt es zwei Schnittpunkt, die der Vorder- und der Hinterseite der Kugel entsprechen. Dabei ist immer $\lambda_1 < \lambda_2$ (da die Wurzel immer positiv ist), d.h. der Schnittpunkt bei λ_1 ist derjenige mit der Vorderseite. Solange man nicht transparente Objekte betrachtet, ist also nur λ_1 von Bedeutung.

Der eigentliche Punkt auf der Kugel, den der Beobachter sieht, ergibt sich nun als

$$\vec{P} = \vec{O} + \lambda_1\vec{d}$$

Es ergibt recht nette Farben, wenn einfach der Normalvektor als Farbe dargestellt wird. Ein Vektor im $\mathbb{R}^3$ besteht ja aus drei Komponenten, die jeweils als Rot/Grün/Blau verwendet werden können. Der *Normalvektor* (Vektor senkrecht auf die Oberfläche an jedem Punkt) mit der Länge 1 ergibt sich auf der Kugel als Differenz zwischen Punkt und Mittelpunkt, dividiert durch den Abstand dieser beiden Punkte, dem Radius der Kugel:

$$\vec{n} = \frac{\vec{P} - \vec{M}}{R}$$

Die Farbabbildung von $\vec{n}$ nach Rot/Grün/Blau (Pixelfarbe) kann nun lauten:

$$\text{Rot} = 255 \cdot |n.x|$$

$$\text{Grün} = 255 \cdot |n.y|$$

$$\text{Blau} = 255 \cdot |n.z|$$

Diese Farben werden nun für denjenigen Bildschirmpunkt gesetzt, dessen Koordinaten x, y den ursprünglichen Sichtstrahl bestimmt haben.

4.2.1 Bestimmung der Bildschirmvektoren

Die in der Grundgleichung des Raytracing

$$\vec{g}(\lambda) = \vec{S} + \lambda \left[\vec{B} - \vec{S} + x\vec{X} + y\vec{Y} \right]$$

auftretenden Größen $\vec{S}$,$\vec{B}$,$\vec{X}$,$\vec{Y}$ sind durch den Beobachter bestimmt. Ein Beobachter ist durch die folgenden Größen definiert:

$\vec{S}$	Standpunkt (`vector3`)	z.B. $(0, -10, 0)$
$\vec{B}$	Blickpunkt (`vector3`)	z.B. $(0, 0, 0)$
φ	waagrechter Bildwinkel (`double`)	z.B. 45°
$\vec{o}$	Richtung „oben“ (`vector3`)	z.B. $(0, 0, 1)$

Während $\vec{S}$ und $\vec{B}$ daraus direkt eingesetzt werden können, müssen $\vec{X}$ und $\vec{Y}$ noch berechnet werden, und zwar gemäß

$$\vec{Y} = \vec{o} \cdot \tan\varphi \frac{\text{MaxY}}{\text{MaxX}}$$

$$\vec{X} = \left(\vec{B} - \vec{S} \right) \times \vec{o} \cdot \tan\varphi$$

Die Vektoren $\vec{S}$,$\vec{B}$,$\vec{X}$,$\vec{Y}$ sind für alle Bildpunkte x, y konstant.

4.2.2 Berechnung des Lichteinfalls

Während obiges Programm zwar recht nette Bilder liefern kann, so ist es doch unphysikalisch, da Objekte üblicherweise nicht von selbst leuchten, sondern von Lichtquellen angestrahlt werden. Diesen Lichteinfall kann man leicht nachahmen. Dazu nehme man an, eine punktförmige Lichtquelle befinde sich am Punkt $\vec{L}$. Der Richtungsvektor von einem Aufpunkt $\vec{P}$ zur Lichtquelle ist dann $\vec{L} - \vec{P}$. Der Cosinus des Winkels, mit dem ein Lichtstrahl von der Lichtquelle am Punkt $\vec{P}$ auftrifft, ergibt sich als Skalarprodukt des Einfallseinheitsvektors mit dem Normalvektor der Oberfläche:

$$\cos\alpha = \left(\vec{L} - \vec{P} \right)_u \cdot \vec{n}$$

Die Lichtintensität am Punkt $\vec{P}$ ist dann genau proportional zum Cosinus des Einfallswinkels. Die Rot/Grün/Blau-Werte am Punkt $\vec{P}$ brauchen daher nur noch mit $\cos\alpha$, einem Wert zwischen 0 und $+1$ (negative Werte entsprechen der Schattenseite und sind daher zu ignorieren), multipliziert werden, um eine realistische Darstellung einer Kugel zu erreichen.

5 Vererbung, virtuelle Funktionen und abstrakte Klassen

Aufgabenstellung:
Das Raytracing-Programm aus dem vorigen Abschnitt soll zur Darstellung beliebiger graphischer Objekte erweitert werden.

Schlüsselbegriffe:
Vererbung, Basisklassen, abgeleitete Klassen, `virtual`, pure virtual, abstrakte Klassen.

Im vorigen Abschnitt wurde das Grundprinzip des Raytracings vorgestellt. Dabei wurde vor allem die Klasse `vector3` betrachtet und die Operatoren, die die einfache Handhabung der `vector3`-Objekte ermöglichen. Allerdings sind die Vektoren nicht die einzigen Objekte, die in der Programmstruktur auftreten. Objektorientiertes Programmieren bedeutet ja, die beteiligten Objekte als Gesamtheit zu betrachten, und so ist es naheliegend, noch eine Klasse `Gerade3` als Zusammenfassung von Ursprungspunkt und Richtungsvektor zu definieren

```
class Gerade3 { vector3 O, d; } ;
```

und eine Klasse `Sphere3` als Zusammenfassung von Mittelpunkt und Radius:

```
class Sphere3 { vector3 M; double R } ;
```

Für jeden Bildpunkt wird dann eine `Gerade3` aus den x, y-Koordinaten konstruiert. Die Schnittpunktsberechnung soll in einer Elementfunktion der Klasse `Sphere3` ausgeführt werden:

```
double  Sphere3::intersect(const Gerade3&g);
```

Damit können die Datenelemente `M` und `R` in der `intersect`-Routine `private` sein und das Objekt `Sphere3` als abgeschlossenes System behandelt werden. Die Funktion `intersect()` soll den kleinsten positiven Schnittpunkt der Geraden mit der aktuellen Kugel liefern. Nur positive Werte für den Geradenparameter sind sinnvoll, die Funktion kann also irgendeinen negativen Wert liefern, wenn kein Schnittpunkt existiert.

Wenn mehrere Kugeln existieren, so ist es Aufgabe des Hauptprogrammes (oder einer entsprechenden „globalen“ Schnittpunktsroutine), von allen Kugeln den kleinsten positiven Schnittpunkt zu finden. Um alle Kugeln, die zu einem Bild beitragen sollen, der Reihe nach abfragen zu können, bietet sich deren Zusammenschluß in einer doppelt verketteten Liste an, wie in 3 vorgestellt. Der „Datenbereich“ eines Listenobjektes besteht also in diesem Fall aus Mittelpunkt und Radius:

```
class   Sphere3
{
        Sphere3 *prev,*next;

        vector3 M;
        double  R;

public:  ...              // Konstruktoren, Listenfunktionen, etc.

        double  intersect(const Gerade3&g);
};
```

Die globale Intersect-Funktion könnte dann folgendermaßen aussehen:

```
double  Sphere3::global_intersect(const Gerade3&g)
{
double  mindist = -1;
        if (!root) return mindist;;

dlist*element = root;
        do
        {
        double  dist = element->intersect(g);
                if (dist>0 && (mindist<0 || dist<mindist))
                        mindist = dist;

                element=element->next;
        }
        while(element!=root);

        return dist;
}
```

Das Hauptprogramm kann somit aus einer beliebig langen Liste von Objekten den minimalen Schnittpunkt ermitteln:

```
        for(y ... ) for(x ... )
        {
        g3      Strahl( ... );
        double  mindist = Sphere3::global_intersect(Strahl);
                if (mindist<0) continue;
                 ...
        }
```

Diese Information ist im Prinzip schon genug, um den Schnittpunkt bestimmen zu können. Für ein Beleuchtungsmodell ist allerdings auch der Normalvektor der Oberfläche am Schnittpunkt von Bedeutung. Die Information „Geradenparameter“ muß daher um die Information des Normalvektors erweitert werden. Am einfachsten geschieht dies dadurch, daß die globale Intersect-Funktion anstelle eines `double` eine Datenstruktur zurückliefert, die alle notwendigen Informationen enthält (alternativ könnte die Intersect-Funktion zusätzliche Referenzparameter erhalten, denen dann die Werte zugewiesen werden).

Der Normalvektor könnte nun mit jeder Schnittpunktsberechnung ermittelt werden. Da aber nicht für jeden Schnittpunkt die Oberflächennormale berechnet werden muß (sondern nur für den nächsten), ist es effizienter, nur das Objekt zu merken, das vom Strahl getroffen wurde, und die Berechnung des Normalvektors in eine separate Funktion zu verlegen:

```
vector3 Sphere3::Normale(const vector3&wo)
{
        return unit(wo-M);
}
```

Bei einer Kugel ergibt sich der Normalvektor als Differenzvektor zum Mittelpunkt.

```
struct  Intersection_info
{
        double  dist;           // Geradenparameter des Schnittpunktes
        Sphere3 *obj;           // Zeiger auf das getroffene Objekt

        Intersection_info()
        :dist(-1.0), obj(0)
        {}
};
```

Die globale Intersect-Funktion von oben kann leicht um die zusätzliche Information erweitert werden:

```
Intersection_info Sphere3::global_intersect(const g3&g)
{
Intersection_info I;
         ...
                        I.dist = dist;
                        I.obj = element;
         ...
        return I;
}
```

Im Hauptprogramm ist es nun ein Leichtes, den Normalvektor zu ermitteln:

```
         ...
        Intersection_info I = Sphere3::global_intersect(Strahl);

                if (I.dist<0) continue;

        // Ermittlung des Schnittpunktes
        vector3 P = Strahl.O + I.dist * Strahl.d;

        // Ermittlung des Normalvektors der Oberfläche
        vector3 n = I.obj->Normale(P);
```

Für ein Beleuchtungsmodell muß dann nur noch die Position einer (oder mehrerer Lichtquellen) bekannt sein. Wenn $\vec{L}$ die Position einer Lichtquelle ist und $\vec{P}$ der Schnittpunkt, so ist $\vec{l} = (\vec{L} - \vec{P})_u$ die Richtung zur Lichtquelle und die Lichtintensität am Aufpunkt $\vec{P}$ kann mit folgender Formel berechnet werden (**Phong**'sches-Beleuchtungsmodel):

$$\vec{C} = \vec{C}_{\text{diffuse}} \cos\alpha + \vec{C}_{\text{speculiar}} \cos^{\gamma}\beta$$

Dabei sind $\vec{C}$, $\vec{C}_{\text{diffuse}}$ und $\vec{C}_{\text{speculiar}}$ „Farbvektoren", also Rot/Grün/Blau-Werte im Bereich $[0, 1]$. α ist der Einfallswinkel des Lichtes und kann direkt aus dem Skalarprodukt von Normalvektor $\vec{n}$ und Lichteinfallsrichtung $\vec{l}$ berechnet werden. β ist der Winkel zwischen Lichteinfallsrichtung und dem an der Oberfläche reflektierten Sichtvektor $\vec{d}$ (der Richtungsvektor des Sichtstrahls). Für den an einer Oberfläche mit Normalvektor $\vec{n}$ reflektierten Vektor ergibt sich die einfache Formel:

$$\vec{r} = \vec{d} - 2(\vec{n} \cdot \vec{d}) \cdot \vec{n}$$

$\cos\beta$ ist schließlich das Skalarprodukt von $\vec{r}$ und $\vec{l}$. Die Größe γ ist, wie $\vec{C}_{\text{diffuse}}$ und $\vec{C}_{\text{speculiar}}$, eine Objekteigenschaft und gibt an, wie rauh eine Oberfläche ist. Je größer, desto „spiegeliger" erscheint eine Oberfläche. Der Farbvektor $\vec{C}_{\text{speculiar}}$ ist für nichtmetallische Materialien immer weiß, für farbige Metall (Gold, Kupfer) entspricht er der diffusen Objektfarbe $\vec{C}_{\text{diffuse}}$.

Die beteiligten Objekte können im Hauptprogramm statisch

```
Sphere3 S(vector3(.7,.4,0), 0.1);
        S.inlist(&S_root);
```

oder dynamisch

```
Sphere3 *S = new Sphere3(vector3(-.7,-.4,0), 0.1));
        S->inlist(&S_root);
```

erzeugt werden. Dabei ist `S_root` ein existierendes Objekt, für das `Sphere3::set_root()` aufgerufen wurde. Die zweite Zeile ließe sich noch „kompaktifizieren":

```
        (new Sphere3(vector3(-.7,-.4,0), 0.1))->inlist(&S_root);
```

womit die Zwischenvariable `Sphere*S` eliminiert werden kann. Allerdings muß man sich so einen Mechanismus überlegen, wie die dynamisch erzeugten Objekte wieder vernichtet werden können, wenn sie nur noch in der Liste existieren. Beispielsweise könnte man dynamisch erzeugte Listenobjekte besonders markieren und eine spezielle Funktion erstellen, die die Liste nach solchen Objekten durchsucht und diese dann vernichtet (siehe in diesem Zusammenhang insbesondere 5.4). Da die hier vorgestellten `Sphere3`-Objekte jedoch „harmlos" sind (d.h. keine offenen Dateien u.ä. enthalten), ist ein derartiger Aufwand hier nicht notwendig und man könnte beispielsweise in einer Schleife eine beliebige Anzahl dynamischer Objekte mit einem Zufallsgenerator positionieren.

5.1 Ableiten von Klassen - Vererbung

Kugeln sind nicht das einzige geometrische Objekt, des weiteren gibt es auch noch jede Menge anderer Gebilde. Im Prinzip müßte man für alle diese Objekte gleich vorgehen wie bei den Kugeln: Eine Datenstruktur erstellen, die notwendigen Elementfunktionen einfügen, im Hauptprogramm eine Liste dieser Objekte durchgehen und dann den kleinsten positiven Schnittpunkt aller Objektlisten auswählen.

Die Klasse `Ebene` (bestimmt durch Ursprung und zwei Richtungsvektoren) beispielsweise wäre somit von der folgenden Form:

```
class   Ebene
{
        Ebene *prev,*next;

        vector3 Origin, X, Y;

public:  ...                // Konstruktoren, Listenfunktionen, etc.

        double  intersect(const Gerade3&g);
        vector3 Normale(const vector3&wo);
};
```

All diesen Objekttypen sind jedoch einige Eigenschaften gemeinsam, nämlich die Verbdindung in einer doppelt verketteten Liste und Möglichkeit, den Schnittpunkt mit einer Geraden berechnen zu können. Um diese Gemeinsamkeiten verdeutlichen zu können, bietet C++ den *Vererbungsmechanismus* bzw. die Bildung von *Basisklassen* an. Eine Basisklasse enthält die Gemeinsamkeiten ähnlicher Klassen, also hier in erster Linie die Listenstruktur:

```
class   Object3D
{
        Object3D *prev,*next;

public: Object3D();
        ~Object3D();

        void inlist(Object3D*);
        void exlist();
};
```

Eine Klasse kann nun die in der Basisklasse definierten Elemente (Daten, Funktionen) *erben*, indem sie von ihr *abgeleitet* wird:

```
class   Sphere3 :  public Object3D
{
        vector3 M;
        double  R;
public:  ...             // Konstruktoren etc.

        double  intersect(const Gerade3&g);
        vector3 Normale(const vector3&wo);
};

class   Ebene :  public Object3D
{
        vector3 Origin, X, Y;

public:  ...             // Konstruktoren

        double  intersect(const Gerade3&g);
        vector3 Normale(const vector3&wo);
};
```

Man unterscheidet „öffentliches“ und „privates“ Ableiten. Privates Ableiten bedeutet, daß die Ableitung nur für Element- und `friend`-funktionen bzw. `friend`-Klassen sichtbar ist, andernfalls ist die Ableitung nach außen hin für alle Verwendungen offensichtlich. Privat abgeleitete Klassen können einzelne Elemente der Basisklasse als öffentlich deklarieren, sodaß die Eigenschaften, die von der Basisklasse geerbt werden, genau festgelegt werden können. Zumeist werden Klassen jedoch öffentlich abgeleitet.

Eine abgeleitete Klasse „erbt“ alle Funktionen und Daten von ihrer Basisklasse, ein Zeiger auf eine abgeleitete Klasse kann – implizit – in einen Zeiger auf die Basisklasse konvertiert werden. D.h. konkret, daß für Objekte der abgeleiteten Klasse `Sphere3` die Elementfunktion `inlist(Object3D*)` definiert ist und dieser Funktion die Adresse eines `Sphere3`-Objektes übergeben werden kann:

```
Sphere3 S1( ... ), S2( ... );

        S1.inlist(&S2);
```

Also genau wie zuvor, als wäre die Listenstruktur direkt in der Klasse `Sphere3` definiert worden. Allerdings können auch Zeiger auf `Ebene`-Objekte in Zeiger auf die Basisklasse `Object3D` konvertiert werden:

```
Sphere3 S1( ... );
Ebene   E1( ... );

        S1.inlist(&E1);
```

Die Liste kann somit aus verschiedenen Objekten bestehen, die nur durch die gemeinsame Eigenschaft der „Liste" verbunden sind.

5.2 Virtuelle Funktionen

Anstatt für jeden Objekttyp eine eigene Liste und eine eigene globale Intersect-Funktion zu erstellen, kann nun eine einzige Funktion geschrieben werden, die alle Objekte einer Liste behandelt. Diese Funktion steht nun allerdings vor dem Problem, erkennen zu müssen, welche `::intersect()`-Elementfunktion sie aufrufen muß, den sie selbst hat nur Kenntnis von den Gemeinsamkeiten aller Objekte, nicht aber deren spezifischen Eigenschaften. Ein Zeiger auf die Basisklasse kann auch nicht automatisch in einen Zeiger auf eine abgeleitete Klasse konvertiert werden, denn dazu müßte bekannt sein, von welchem Typ ein spezielles Objekt ist. Eine Möglichkeit, dies zu bewerkstelligen, wäre es, ein spezielles Datenfeld in die Basisklasse aufzunehmen, das angibt, auf welchen Typ es sich tatsächlich bezieht:

```
class   Object3D
{
        Object3D *prev, *next;
protected:
        enum { SPHERE3, EBENE } type;

public:  ...
};

class   Sphere3 :  public Object3D
{
         ...
public: Sphere3( ... )
        {
                 ...
                type = SPHERE3;
        }
};

class   Ebene :  public Object3D
{
         ...
```

```
public: Ebene( ... )
        {
                 ...
                type = EBENE;
        }
};
```

An dieser Stelle wurde der dritte Schutzbegriff, `protected`, verwendet. `protected` stellt eine Zwischenstufe zwischen `public` und `private` dar: Abgeleitete Klassen dürfen auf `protected`-Elemente zugreifen.

Die globale Intersect-Funktion müßte nun je nach Objekttyp entscheiden, welches Objekt sie tatsächlich vor sich hat:

```
Intersection_info Object3D3::global_intersect(const g3&g)
{
Intersection_info I;
        if (!root) return I;

Object3D*element = root;
        do
        {
        double  dist;
                switch(element->type)
                {
                case SPHERE3:
                        { Sphere3*obj = (Sphere3*)element;
                        dist = obj->intersect(g);
                        break;
                        }

                case EBENE:
                        { Ebene*obj = (Ebene*)element;
                        dist = obj->intersect(g);
                        break;
                        }

                default:  dist = -1;    // Unbekannter Objekttyp!
                }

                if (dist>0 && (I.dist<0 || dist<I.dist))
                {
                        I.dist = dist;
                        I.obj = element;
                }
                element=element->next;
        }
        while(element!=root);

        return I;
}
```

Wiewohl diese Vorgangsweise zwar für das konkrete Problem die geeignete Lösung darstellt, so weist sie doch den gravierenden Nachteil auf, daß in der globalen Intersect-Funktion alle möglichen Objekttypen bekannt sein müssen. Will man eines Tages einen neuen Objekttyp hinzufügen, so muß auch die globale Intersect-Routine um die entsprechende Kenntnis des neuen Typs erweitert werden.

Das sollte natürlich möglichst verhindert werden. Eine Möglichkeit bestünde darin, anstelle des Typenfeldes, im Beispiel hier als `enum` ausgeführt, einen Funktionszeiger zu verwenden, der für jeden Objekttyp auf eine spezifische Funktion verweist. Die globale Intersect-Funktion kann dann über den Funktionszeiger immer die richtige Funktion aufrufen.

Genau diesen Mechanismus unterstützt C++ mithilfe der *virtuellen Funktionen.* Eine virtuelle Funktion wird in der Basisklasse mit dem Vorsatz `virtual` deklariert:

```
class   Object3D
{
         ...
        virtual double          intersect(const Gerade3&g);
        virtual vector3         Normale(const vector3&wo);
};
```

Wird nun in einer abgeleiteten Klasse eine Elementfunktion gleichen Namens und gleichen Typs (Parameter, Rückgabewert) definiert, dann wird nicht die Elementfunktion der Basisklasse, sondern diejenige der abgeleiteten Klasse aufgerufen, wie das folgende schematische Beispiel zeigen soll:

```
#include <stdio.h>
class   Basis
{
public: virtual void    wer_bin_ich() { puts("Die Basisklasse"); }
};

class   Abgeleitet :  public Basis
{
public: void    wer_bin_ich() { puts("Die abgeleitete Klasse"); }
};

void    f(Basis*obj)
{
        obj->wer_bin_ich();
}

main()
{
Basis   B;
Abgeleitet A;
        f(&B);
        f(&A);
        return 0;
}
```

Obwohl die Funktion `f(Basis*obj)` nur Kenntnis von der Basisklasse hat, ruft sie dennoch immer die „richtige" Elementfunktion auf. Die Intersect-Funktion ist damit so einfach wie in der ersten Variante:

```
Intersection_info Object3D::global_intersect(const g3&g)
{
Intersection_info I;
        if (!root) return I;

Object3D*element = root;
        do
        {
        double  dist = element->intersect(g);
                if (dist>0 && (I.dist<0 || dist<I.dist))
                {
                        I.dist = dist;
                        I.obj = element;
                }
                element=element->next;
        }
        while(element!=root);

        return I;
}
```

Allerdings bezieht sie sich so auf *beliebige* Objekttypen, zu den bestehenden können jederzeit neue hinzugefügt werden, ohne daß man die globale Intersect-Routine in irgendeiner Weise modifizieren müßte.

5.3 Abstrakte Klassen

Die Klasse `Object3D` dient nur als Basis für die Gemeinsamkeiten der spezifischen Objekttypen. Ein Objekt vom Typ `Object3D` selbst hat eigentlich keinen Sinn. Die virtuelle Intersect-Funktion könnte daher immer -1 liefern, für den Fall, daß ein unwissender Anwender ein Objekt vom Typ `Objekt3D` definiert:

```
double  Object3D::intersect(const Gerade3&g)
{
        return -1;      // Kein Schnittpunkt möglich.
}
```

Die virtuellen Funktionen der Basisklassen können gleichsam „Defaultwerte“ für Funktionen liefern, die in einer abgeleiteten Klasse optional berechnet werden können. Wird eine virtuelle Funktion in einer abgeleiteten Klasse nicht neu definiert („überladen“), so wird die Funktion der Basisklasse verwendet.

Das Überladen einer virtuellen Funktion kann in C++ jedoch auch *erzwungen* werden, indem die Funktion in der Basisklasse als *pur virtuell* deklariert wird:

```
class   Object3D
{
         ...
        virtual double          intersect(const Gerade3&g) = 0;
         ...
};
```

Eine virtuelle Funktion mit dem Zusatz „`= 0`“ ist *pur virtuell*, eine Klasse, die pur virtuelle Funktionen enthält, ist *abstrakt*. Abstrakte Klassen existieren nur als Typ, aber es können keine Objekte vom Typ einer abstrakten Klasse erstellt werden. Der Versuch, ein Objekt vom Typ `Object3D` zu erzeugen, führt zu einer Fehlermeldung des Compilers. Zeiger und Referenzen auf abstrakte Klassen sind jedoch weiterhin möglich. Wird in einer abgeleiteten Klasse eine pur virtuelle Funktion nicht überladen, so ist auch diese Klasse abstrakt. Um ein Objekt vom Typ einer abstrakten Klasse erzeugen zu können, müssen daher alle pur virtuellen Funktionen der Basisklasse überladen werden.

5.4 Virtuelle Destruktoren

Wenn, wie im aktuellen Beispiel, Objekte abgeleiteter Klassen dynamisch erzeugt werden, aber nicht explizit vernichtet werden, sondern beispielsweise über eine Liste auf Objekte der Basisklassen erfaßt und der Reihe nach gemeinsam vernichtet werden sollen, tritt die Problematik auf, daß `delete` auf einen Zeiger auf die Basisklasse angewendet werden muß, in Wahrheit aber auf ein abgeleitetes Objekt zeigt. Damit nun alle Datenstrukturen der abgeleiteten Klasse auch korrekt eliminiert werden, ist es notwendig, den Destruktor der Basisklasse als virtuell zu deklarieren:

```
struct  A
{
        virtual ~A();    // Virtueller Destruktor
};
```

Damit wird der Destruktor der abgeleiteten Klasse bei der Vernichtung des Objektes der Basisklasse korrekt aufgerufen. Im Gegensatz zu gewöhnlichen virtuellen Funktionen wird allerdings nach dem Aufruf des Destruktors der abgeleiteten Klasse noch der Destruktor der Basisklasse aufgerufen, da ja die Elemente der Basisklasse nur von dieser korrekt abgebaut werden können (eine gewöhnliche virtuelle Funktion würde die Funktion der Basisklasse vollständig ersetzen):

```
#include <stdio.h>

struct  A
{
        virtual ~A() { puts("A::~A()"); }
};

struct  B : A
{
        ~B() { puts("B::~B()"); }
};

main()
{
B       b;

        return 0;
}
```

Anmerkung: Virtuelle Konstruktoren sind nicht möglich. Übungsfrage: Warum nicht?

5.5 Anhang

Bestimmung des Schnittpunktes einer Geraden mit einer Ebenen: Ein Punkt auf der Ebenen ist durch zwei Parameter μ und ν bestimmt. Die Schnittpunktsgleichung lautet daher:

$$\vec{g}_O + \lambda \cdot \vec{g}_d = \vec{O} + \mu \vec{X} + \nu \vec{Y}$$

Diese Gleichung kann leicht auf die Form

$$\vec{g}_O - \vec{O} = -\lambda \cdot \vec{g}_d + \mu \vec{X} + \nu \vec{Y}$$

gebracht werden. Allgemein liefert das Kreuzprodukt eines Vektors mit sich selbst Null, durch Bilden des Kreuzproduktes obiger Gleichung mit $\vec{Y}$ kann daher $\nu \vec{Y}$ eliminiert werden:

$$\left(\vec{g}_O - \vec{O}\right) \times \vec{Y} = -\lambda \cdot \vec{g}_d \times \vec{Y} + \mu \vec{X} \times \vec{Y}$$

Die Gleichung muß nun nur noch mit $\vec{X} \times \vec{Y}$ multipliziert (im Sinne des Kreuzproduktes) werden, damit auch μ verschwindet:

$$\left[\left(\vec{g}_O - \vec{O}\right) \times \vec{Y}\right] \times \left(\vec{X} \times \vec{Y}\right) = -\lambda \cdot \left(\vec{g}_d \times \vec{Y}\right) \times \left(\vec{X} \times \vec{Y}\right)$$

Diese Gleichung für λ besteht eigentlich aus drei unabhängigen Gleichungen, jeweils für die x, y und z-Komponente. Alle drei Gleichungen liefern dasselbe Ergebnis für den gesuchten Geradenparameter λ, es kann jedoch sein, daß die Zahlenwerte dergestalt sind, daß sich in einer Komponente eine Division durch 0 ergäbe. In diesem Fall ist die Lösung aus derjenigen Gleichung zu bestimmen, deren Division ausführbar ist. Ist keine der drei Gleichungen lösbar, dann gibt es keinen Schnittpunkt, d.h. die Gerade ist parallel zur Ebene.

6 Templates

AUFGABENSTELLUNG:
Die bisher verwendeten Konstrukte und Algorithmen, insbesondere das Konzept der doppelt verkettete Liste, sollen so formuliert werden, daß sie leicht wiederverwendbar sind.

SCHLÜSSELBEGRIFFE:
`template`'s (Funktionen und Klassen).

Ziel von C++ ist es, die Programmiersicherheit zu erhöhen. Dazu ist notwendig, Sprachstrukturen zu bieten, die den Einsatz des Präprozessors unnötig machen. In C wird der Präprozessor immer dann eingesetzt, wenn die Sprache selbst an ihre Grenzen gelangt. Zitat: „Es zeigt immer einen Fehler an, wenn Macros eingesetzt werden – entweder von seiten der Programmiersprache oder von seiten des Programmierers.“ (Bjarne Stroustrup, The C++ Programming Language, Seite 138, Section 4.7 Macros). Die Macros des Präprozessors stellen eine Fehlerquelle dar, da sie nicht an die Syntaxkontrolle von C/C++ gebunden sind und zu vollkommen unerwarteten Ergebnissen führen können.

Die meisten Anwendungen der `#define`-Direktive können in C++ mit spracheninternen Mitteln umgesetzt werden: Mit `const` definierte Variablen werden wie `#define` in C an der Stelle ihrer Verwendung eingesetzt und Rechnungen, so möglich, bereits vom Compiler ausgeführt. Die aus Effizienzgründen verwendeten Macros mit Parametern können durch `inline`-Funktionen ersetzt werden. Sie unterliegen dann der Syntaxkontrolle durch den Compiler und haben im Gegensatz zu Macros keine unvorhersehbaren Seiteneffekte.

Dennoch gibt es Situationen, in denen eine `inline`-Funktion ein Macro nicht ersetzen kann. Gerade daß ein Macro nicht der Kontrolle durch den Compiler unterliegt, wird oft verwendet, wenn beispielsweise ein und dasselbe Macro für unterschiedliche Typen verwendet werden soll, wie beispielsweise beim beliebten Maximums-Macro:

```
        #define max(a,b)((a)>(b)?(a):(b))
```

Auch wenn dieses Macro recht gefährlich ist (die Parameter können ja komplizierte Ausdrücke oder Funktionsaufrufe sein, die dann mehrmals ausgewertet werden, z.B. `max(++a,++b)`), wird es aufgrund seiner Einfachheit oft verwendet.

6.1 Templatefunktionen

Für Situationen, in denen Operationen **typenunabhängig** ausgeführt werden sollen, sind in C++ die `template`-Funktionen vorgesehen. Bei der Definition der `template`-Funktion wird der Typ für einige oder alle Parameter durch einen unbestimmten Typ, das *Template-Argument*, ersetzt. Wird eine solche Funktion aufgerufen, erstellt der Compiler eine Funktion, indem er die entsprechenden Typen, die beim Aufruf verwendet werden, einsetzt. Eine `template`-Funktion funktioniert wie ein Makro (engl. „template“ = dt. „Schablone“), verhält sich aber wie eine gewöhnliche Funktion. Das Maximumsmakro sieht als `template`-Funktion so aus:

```
template <class T> inline T max(T a, T b)
{
        return a>b?a:b;
}
```

`T` ist das Template-Argument. Für `T` wird je nach Aufruf ein unterschiedlicher Typ eingesetzt. Dies kann eine Klasse sein, für die der Kleiner-Operator definiert (überladen) wurde, oder auch ein intrinsischer Standardtyp wie `int` oder `double`. In der Parameterliste muß mindestens ein

Parameter vom Typ des Template-Argumentes vorhanden sein, damit der Compiler anhand des Funktionsaufrufes die Funktion erstellen kann.

Eine Templatefunktion kann verwendet werden, als ob sie für den angesprochenen Typ definiert wäre:

```
int     f();
int     g();

main()
{
int     i, j, k;
double  d, e, f;

        i = max(j,k);           // Aufruf der Funktion int max(int,int)
        d = max(e,f);           // double max(double,double)
        i = max(1,6);           // int max(int,int)

        i = max( f(), g() );    // f() und g() werden nur einmal
                                // aufgerufen! Vgl. mit C - #define !
         ...
}
```

Tritt in einer Templatefunktion ein Syntaxfehler auf, so wird dieser erst beim Aufruf der Funktion gemeldet, nicht schon bei der Definition – Ursache kann ja beispielsweise eine fehlende Operatorfunktion für eine bestimmte Klasse sein.

ANMERKUNG: Es ist kann günstiger sein, eine Templatefunktion mit Referenzargumenten auszuführen:

```
template <class T> inline T&max(T&a, T&b)
{
        return a>b?a:b;
}
```

Bei Klassen mit aufwendigen Copy-Konstruktoren spart man somit das Erzeugen temporärer Objekte als Parameter und eines Rückgabewertes als Kopie des Maximums beider Parameter. Hinzu kommt, daß so der Rückgabewert der Funktion einen *Lvalue* darstellt, d.h. ihm etwas zugewiesen werden kann:

```
int     i = 10, j = 20;
        max(i, j) = 0;  // j wird auf 0 gesetzt.
```

Der Aufruf `max(++a, ++b)` ist weiterhin möglich und führt zu keinen Seiteneffekten wie im Fall des `#define`-Makros. ÜBUNGSFRAGE: Warum ist mit dieser Funktionsdefinition der Aufruf `max(a++, b++)` *nicht* möglich?

Zusätzlich zu den Templatefunktionen können für bestimmte Typen explizite Ausführungen bereit gestellt werden, beispielsweise eine Maximumsfunktion für Strings, d.h. den Typ `char*`, da für diesen Typ der <-Operator durch die Libraryfunkton `strcmp()` zu ersetzen ist:

```
char*&max(char*&a, char*&b)
{
        return strcmp(a,b)>0?a:b;
}
```

Damit sind – zusätzlich zu den „gewöhnlichen" Maximumsfunktionen – auch Aufrufe mit Strings möglich, was mit einem C-Makro bei weitem nicht erreicht werden kann:

```
        printf("%s\n", max("abadabadu", "feuerstein") );
```

6.2 Templateklassen

Neben den Templatefunktionen sind auch Templateklassen möglich. Ähnlich wie eine Templatefunktion erst mit ihrem Aufruf erstellt wird, wird eine Templateklasse erst mit der Definition eines Objektes erstellt. Innerhalb einer Templateklasse kann das Templateargument wie ein beliebiger Typ verwendet werden:

```
template <class T> class Array_with_range_check
{
        T       *data;
        int     n;

public: Array_with_range_check(int elements)
        {
                data = new T[n=elements];
        }

        ~Array_with_range_check()
        {
                delete [] data;
        }

        T&operator[](int index) const
        {
                if (index<0 || index>=n)
                {
                        cerr    << "Indexüberschreitung, Index " << index
                                << " ist nicht erlaubt!\n";
                        exit(3);
                }
                return data[index];
        }
};
```

Beim Erstellen eines Objekts der Templateklasse müssen die Templateargumente explizite angegeben werden:

```
main()
{
```

```
Array_with_range_check<char> s(100);
Array_with_range_check<double> d(23);

        s[425] = 0;      // Dieser Zugriff wird überprüft !

        return 0;
}
```

Templatetypen wie `Array_with_range_check<char>` können wie alle anderen Typen verwendet werden. Auch ein `int` kann ein Templateargument sein:

```
template <int n, class T> class fixed_Array_with_range_check
{
        T       data[n];

public:  T&operator[](int index) const
        {
                ...     // wie oben
        }
};
```

Bei der Erstellung eines Templatetypes muß für das `int`-Argument ein konstanter Zahlenwert angegeben werden:

```
main()
{
fixed_Array_with_range_check<100,char> s;
fixed_Array_with_range_check<23,double> d;

         ...
}
```

Hierbei ist der Typ `fixed_Array_with_range_check<100,char>` ein anderer **Typ** als der ähnliche Typ `fixed_Array_with_range_check<101,char>`. Beide Typen können nicht ineinander konvertiert werden, sofern nicht besondere Vorkehrungen (Konversionsoperatoren) getroffen werden.

Sinn und Zweck von Templateklassen ist die Realisierung von Konzepten, die nicht für einen bestimmten Typ ausgelegt sind, wie beispielsweise das Array mit Bereichsüberprüfung.

6.3 Übungsaufgabe

Konstruiere weitere praktische Templateklassen wie beispielsweise ein Stack-Template sowie ein Template für doppelt verkettete Listen.

7 Weiteres ...

7.1 Überladen der Operatoren new und delete

Wie `+` und `*` und viele andere, sind auch `new` und `delete` *Operatoren*. Sie sind **keine** Schlüsselwörter sondern unterliegen allen Gesetzen, die für Operatoren gelten, wie Rangfolge, Zuordnungsreihenfolge, Auswertungsreihenfolge. `new` und `delete` können für eine bestimmte Klasse überladen werden, um eine andere als die vorgebene Art der Speicherverwaltung zu verwenden. Die Standardlibraryfunktionen sind auf maximale Inhomogenität der Speicherobjekte optimiert; kann für eine Klasse aber vorausgesetzt werden, daß immer Blöcke gleicher Größe verwendet werden müssen, kann die Speicherverwaltung für diesen Spezialfall einfacher und effizienter organisiert werden.

Soll der Operator `new` überladen werden, so muß sein Funktionswert immer vom Typ `void*` sein – die Adresse des neu angeforderten Speicherbereiches –, das erste Argument muß vom Typ `size_t` sein (im wesentlichen ein `unsigned`). Wird ein Objekt der zugehörigen Klasse per `new` angefordert, wird an die Operatorfunktion die Größe des benötigten Speichers übergeben. Kann dort kein Speicher zur Verfügung gestellt werden, darf sie `0` zurückliefern, um dem Aufrufer den Fehler zu signalisieren. Der Konstruktor für das angeforderte Objekt wird dann **nicht** aufgerufen.

Die Operatorfunktion `new` hat somit keinerlei Kenntnis von dem Objekt selbst, dessen Speicher sie bereitstellen soll. Ähnlichrd gilt für den Operator `delete`, seine Parameter sind ebenfalls fix. Beide Operatorfunktion dürfen **nicht** virtuell sein (da sie keine echten Elementfunktionen darstellen).

```
class   X
{
public: void* operator new(size_t size)
        {       puts("Operator new"); return 0; }

        void operator delete(void*,size_t size)
        {       puts(Operator delete"); }
};

void    f()
{
X*      x = new X;
        delete x;
}
```

Wird ein Array von Objekten angefordert, z.B. `new X[10]`, wird hierfür *immer* der globale Operator `new` verwendet. Es gibt dzt. in C++ keine Möglichkeit, den `new`-Operator für mehrere Objekte in einer Klasse zu überladen (analog bei `delete []`).

Wird der `new/delete` Operator *privat* in einer Klasse deklariert, kann damit die dynamische Speicheranforderung für Objekte dieser Klasse verboten werden. Aus dem vorher genannten Grund kann die Anforderung eines Objektarrays jedoch nicht unterbunden werden.

7.2 Stream-Manipulatoren

Was könnte die folgende Funktionsdefinition für einen Sinn haben ?

```
ostream& operator<<(ostream&os, void (*f)(ostream&))
{
        f(os);
        return os;
```

```
}
```

Wie man nach etwas Nachdenken herausfinden wird, handelt es sich bei dem Funktionsparameter `f` um einen Zeiger auf eine Funktion, und zwar eine solche, die ein `ostream`-Objekt als Parameter erwartet. Ganz analog zur Definition des Ausgabeoperators für `vector3`-Objekte kann nun auch die Adresse einer solchen Funktion in einer Ausgabeanweisung verwendet werden, beispielsweise die der schon in `<iostream.h>` vordefinierten Funktion `flush()`, die den Dateipuffer leert und alle Daten physisch auf die Festplatte/Diskette oder den Bildschirm schreibt:

```
        cout << "Sehr wichtiger Text !\n" << flush << "...  unwichtig.";
```

BEACHTE: `flush` ist die Adresse der Funktion `void flush(ostream&)`. Die Operatorfunktion `<<` wird nun für die Adresse der Funktion aufgerufen, diese wiederum ruft `flush()` mit `cout` als Parameter auf.

Neben `flush()` stehen noch ein paar andere Funktionen zur Verfügung:

`endl`	Zeilenende schreiben und `flush()` aufrufen
`ends`	Zeichencode 0 schreiben (für String-Streams)
`hex`	Zahlenausgabe auf hexadezimal schalten
`oct`	Zahlenausgabe auf oktal schalten
`dec`	Zahlenausgabe auf dezimal schalten
`ws`	Eingabestreams: Leerzeichen ignorieren

Diese und ähnliche Funktionen werden als *Streammanipulatoren* bezeichnet.

7.3 Virtuelle Klassen

Angenommen, mehrere Objekte, wie sie in etwa für 3D-Graphik gebraucht werden, werden in einer doppelt verketten Liste verwaltet. Von der Basisklasse mit der entsprechenden Listenstruktur wird ein Objekt `Kugel` und ein Objekt `Quader` abgeleitet. Ein `Würfel` hat nun Eigenschaften, die in diesen beiden Typen bereits enthalten sind – er kann in eine Kugel eingebettet werden, was die prinzipielle Behandlung vereinfacht, andererseits besteht auch er wie ein `Quader` aus sechs Flächen. Also einfach von beiden Klassen ableiten?

```
class   Object
{
        Object *next, *prev;
         ...
};

class   Kugel :  public Object
{
         ...
};

class   Quader :  public Object
{
         ...
};

class   Würfel :  public Kugel, public Quader
{
         ...
```

```
};
```

Hierbei tritt jedoch ein elementares Problem auf: Ein `Würfel` enthält mit der obigen Klassenhierarchie *zwei* `prev` und *zwei* `next`-Zeiger, einmal über den Vererbungsweg der `Kugel`, einmal über den `Quader`. Das darf natürlich logisch nicht sein, denn ein Objekt kann nur einmal in einer Objektliste sein, nicht mehrmals. Der Ausweg besteht darin, sowohl die `Kugel` als auch den `Quader` *virtuell* von der Basisklasse abzuleiten:

```
class   Kugel :  virtual public Object { ... };

class   Quader :  virtual public Object { ... };
```

Die Basisklasse `Object` ist damit in der Klasse `Würfel` nur einmal anstatt doppelt enthalten (im Konstruktor der Klasse `Würfel` muß nun explizit der Konstruktor der `Object`-Klasse aufgerufen werden, soferne kein Defaultkonstruktor existiert). In diesem Kontext wird die Basisklasse `Object` als **virtuell** bezeichnet. Mithilfe einer virtuellen Basisklasse entsteht aus einer Baumstruktur eine Rautenstruktur, die Vererbungsäste treffen sich nicht nur bei der zuletzt abgeleiteten Klasse, sondern auch bei der Basisklasse.

WARNUNG: Durch virtuelle Vererbung können Mehrdeutigkeiten entstehen, die nicht jeder Compiler zu melden imstande ist!

7.4 Zeiger auf Klassenelemente

Angenommen, ein allgemeiner Algorithmus verwendet speziell eines von mehreren gleichartigen Klassenelementen, beispielsweise könnte eine Funktion genau eine Komponente eines 3D-Vektors auf eine bestimmte Zahl setzen (als einfaches Beispiel). Eigentlich müßte eine solche Funktion für jede Komponente neu implementiert werden. Das kann aber mithilfe der Zeiger auf Klassenelemente verhindert werden. Ein solcher Zeiger kann nicht unabhängig von einem Objekt ausgewertet werden, sondern nur **innerhalb** eines Objektes auf eine Substruktur verweisen.

Zur Auswertung dienen die Operatoren `.*` und `->*` (für Objekte/Referenzen auf Objekte bzw. Zeiger auf Objekte). Um die Anwendung dieser Sprachstruktur zu demonstrieren, soll das folgende Beispiel dienen:

```
struct  vector3
{
        double  x,y,z;
};

void set(double val, vector3*field, int count, double vector3::*k)
{
        for(vector3*end = field+count; field<end; field++)
                field->*k = val;
}

main()
{
vector3 vfield[10];

        set(3.141592, vfield, 10, &vector3::x);
        set(2.718281, vfield, 10, &vector3::y);
        set(0.999999, vfield, 10, &vector3::z);
```

```
        ...
}
```

Im obigen Beispiel wird im Feld `vfield[]` in jedem Element die x-Komponente auf 3.141592, die y-Komponente auf 2.718281 und die z-Komponente auf 0.999999 gesetzt. Der Ausdruck `&vector3::x` ist vom Typ *Zeiger auf ein Klassenelement* und bezieht sich – im Gegensatz zu gewöhnlichen Zeigern – auf einen Typ, nicht auf ein konkretes im Speicher befindliches Datenobjekt.

7.5 Fehlerbehandlung (exceptions)

Beispiel: Arrayklasse, `operator[]`. Vom Hauptprogramm aus wird ein Array falsch indiziert. Wie soll die Funktion `operator[]` reagieren?

- Einen Laufzeitfehler („Runtime-Error") auslösen. Es ist natürlich nicht sehr erfreulich, wenn in einem bereits fertigen, komplexen Programm während der Bedienung, nach der mühsamen Bearbeitung von vielleicht wichtigen Daten, das Programm aufgrund eines zwar elementaren, aber im Grunde doch harmlosen Fehlers, abrupt beendet wird, ohne Möglichkeit, die Daten doch noch vorher abzuspeichern. Schließlich: Alle Fehler kann auch der beste Programmierer nicht von vorneherein ausschließen. In Anbetracht dessen, daß irgendwo ein falscher Funktionsaufruf stattfand, der sowieso erkanntermaßen keinen Schaden anrichten kann, ist der abrupte Abbruch des gesamten Programmes vielleicht doch möglicherweise etwas „überstürzt"!?

- Die Funktion kann ein ungültiges Element zurückliefern, in der Hoffnung, der Aufrufer testet den erhaltenen Funktionswert immer auf Richtigkeit. Aber abgesehen davon, daß Programmierer erfahrungsgemäß zu bequem sind, jederzeit alles zu überprüfen, kostet dies auch Laufzeit. Und was soll geschehen, wenn alle Werte gleich möglich sind, also gar kein ungültiger, abtestbarer Wert existiert!?

- Die Funktion könnte irgendeinen zufälligen gültigen Wert zurückliefern, beispielsweise eine Referenz auf ein statisches Element. Der Aufrufer hat somit zwar keine Möglichkeit, die Richtigkeit zu überprüfen, kann aber auch keinen Schaden anrichten. Im schlimmsten Fall passiert „nichts", das Programm als Ganzes ist also „stabil". Aber Progammfehler zu finden ist so praktisch unmöglich...

Diese Problematik war auch den C++ - Designern bewußt und es wurden die *Exceptions* (*Ausnahmen*) eingeführt. Damit kommen drei neue Schlüsselwörter ins Spiel: `throw`, `try` und `catch`. Mit `throw` kann eine Exception ausgelöst werden. An `throw` wird ein Objekt übergeben, das Informationen über die Art der Ausnahme enthält. Dazu wird meist eine *Exception Class* definiert, die auch leer sein kann, da der Typ allein bereits eine ausreichende Information darstellt:

```
class   string
{
        char*s;
        int len;

public: class   Range_error {};

         ...

        char& operator[](int i)
        {
                if (i<0 || i>=len)
```

```
                        throw Range_error();

                return s[i];
        }
};
```

Ein falsch indizierter String löst nunmehr die Exception `Range_error` aus. Defaultmäßig bewirkt dies den Programmabbruch. Ist dies jedoch nicht gewünscht, was der Aufrufer eher beurteilen kann als die Funktion `string::operator[]()` selbst, kann diese Exception in einem `try`-Block mit der `catch`-Anweisung abgefangen werden:

```
main()
{
char    c;
string  s(100);

        try
        {
                c = s[101];
        }

        catch( string::Range_error )
        {
                cout << "Der obige String wurde völlig falsch indiziert !\n";
        }
}
```

Dabei muß der falsche Zugriff nicht unbedingt direkt im `try`-Block geschehen. Die Exception wird auch dann abgefangen, wenn eine Funktion aufrufen wurde, in der der falsche Zugriff geschah. Der jeweils letzte `try`-Block ist für eine Exception maßgebend. Objekte, die in einer verschachelt aufgerufenen Funktion lokal erzeugt werden, werden vor der Ausführung der `catch`-Anweisung ordnungsgemäß destruiert. Der Exception-Mechanismus gewährleistet ein korrektes Aufräumen lokaler Objekte auch beim Abbruch von tief rekursiven oder verschachtelten Funktionen.

Mithilfe der Exceptions kann eine Libraryfunktion erstellt werden, ohne Annahmen darüber zu treffen, wie der Aufrufer im Fehlerfalle reagieren möchte. Unternimmt dieser nichts, führen Exceptions zum Programmabruch.

Exceptions können auch zur Kommunikation von Funktionen untereinander verwendet werden, allerdings ist dies eher ein Mißbrauch und, da der Stack immer sauber aufgeräumt und der nächstäußere `try`-Block gesucht werden muß, relativ ineffizient.

7.6 mutable - Datenelemente

In einem konstanten Objekt kann kein Datenelement verändert werden. Wirklich keines? Auch konstante Datenobjekte befinden sich im gewöhnlichen Arbeitsspeicher, und der Schreibschutz wird nur vom Compiler zur Verfügung gestellt, nicht von der Hardware, denn irgendwann muß ja auch ein konstantes Objekt initialisiert werden. Auf den Speicher kann man aber prinzipiell immer auch schreibend zugreifen, und wenn der Compiler das nicht direkt gestattet, dann eben indirekt über einen Zeiger (dessen Typ natürlich explizit von einem `const object*` in einen `object*` konvertiert werden muß).

Auch konstante Objekt können somit, entsprechenden Zeigeraufwand vorausgesetzt, beschrieben werden. Damit man derartigen „Unfug“ wenigstens sofort erkennen kann und nicht erst

kryptische Tricks verwendet werden, hat man das Schlüsselwort `mutable` eingeführt. Ein `mutable`-Datenelement kann auch in einer konstanten Datenstruktur verändert werden.

Eine sinnvolle Anwendung von `mutable` wäre es, das Ergebnis einer aufwendigen Berechnung in einem ansonsten konstanten Objekt zu merken und es ggf. später bei einer ähnlichen Berechnung wieder zu verwenden. In der Anwendung „von außen" erscheint das Objekt als konstant, nur die Bearbeitung läuft schneller ab im Vergleich zur Version ohne internes Merken des Ergebnisses.

7.7 Runtime-Type-Checking

Ein ab und zu auftretendes Problem: Eine Funktion, die auf einer Basisklasse operiert, möchte wissen, von welchem Typ ein Objekt denn nun wirklich ist. Zwar sollte eine solche Notwendigkeit wegen der Möglichkeit virtueller Funktionen kaum auftreten, aber manchmal gibt es keine andere Wahl. Die erste Idee wäre, eine Typenidentifikation in die Basisklasse aufzunehmen. Abgeleitete Klassen müssen diese dann auf einen eindeutigen Wert setzen.

Eine bessere Methode stellen die Operationen des „Dynamischen Typecastings" dar. Im Gegensatz zum statischen Typecasting werden diese Operationen überprüft und lösen im Fehlerfall eine Exception „`bad_cast`" aus oder liefert einen `NULL`-Zeiger, so der Compiler noch der Exceptions nicht mächtig ist.

```
class   Base            {};
class   Derived :  public Base {};

void    f(Base*b)
{
Derived * D = dynamic_cast<Derived*>(b);
         ...
}
```

Der Ausdruck „`dynamic_cast<Derived*>(b)`" reagiert wie oben beschrieben, wenn `b` **nicht** auf ein Objekt vom Typ `Derived` zeigt sondern beispielsweise nur auf ein Objekt der Basisklasse.

7.8 Namespaces und New Style Headers

Je größer und komplexer Programme werden, desto eher treten Namenskonflikte zwischen Funktionen und/oder Variablen auf. Es ist beispielsweise keine gute Idee, eine globale Variable einfach nur „`i`" zu nenne, weil Variablen mit einem solchen Namen massenhaft auftreten. Diese Problematik, daß mit immer größeren Programmen immer weniger eindeutige Bezeichnung zur Verfügung stehen, nennt man die *Namespace-Pollution.* Mithilfe der Einbettung von globalen Variablen und Funktionen in statische Klassenelemente läßt sich diese Problematik etwas in den Griff bekommen.

Um das Problem systematisch anzugehen, kann in Zukunft ein Namespace explizit definiert werden:

```
namespace special
{
        int     x,y;
}

main()
{
        special::x = 8;
        special::y = 9;
}
```

Das Schlüsselwort „`namespace`“ wirkt ähnlich wie eine Klassendefinition, definiert aber eben nur einen Namespace und keine eigene Klasse – ähnlich statischen Klassenelementen. Parallel dazu kann ein Namespace als „aktuell“ ausgewählt werden, hierzu dient „`using`“:

```
main()
{
        using special;

        x = 8;  // hier wird special::x beschrieben
        y = 9;  // dto.
}
```

Alle Standardlibraryfunktionen werden in Zukunft dem speziellen Namespace „`std`“ zugeordnet werden, sodaß jeder Anwender etwa eine eigene Funktion `malloc()` schreiben kann, ohne damit in einen Namenskonflikt mit der Libraryfunktion zu geraten, die dann eben `std::malloc()` heißen wird. Defaultmäßig ist allerdings der Namespace `std` in Verwendung, sodaß die Kompatibilität zu alten Programmen gewährleistet bleibt.

Parallel dazu werden *New Style Headers* ohne Dateinamenserweiterung eingeführt, um eine Unterscheidung zum „alten“ C++ zu ermöglichen; das sieht dann so aus:

```
#include <iostream>
#include <stdio>
```

Der **Inhalt** eines solchen New Style Headers ist relativ trivial, z.B. der von `<stdio>`:

```
namespace std
{
#include <stdio.h>
}

using   std;
```

Die in `<stdio.h>` definierte Funktion `printf()` wird also durch Verwendung des New Style Headers `<stdio>` zur Funktion `std::printf()`.

7.9 Ausblick

Was die ferne Zukunft bringen wird, kann niemand vorhersagen. Man kann nur hoffen, daß C++ nicht komplexer, sondern eher einfacher wird (mit der jungen Programmiersprache `JAVA` versucht man, die Leistungen von C++ ohne den Ballast ihrer Komplexität zu übernehmen). Auf lange Sicht werden aber wohl natürliche, menschliche Sprachen die Computersprachen auch in der Programmierung ablösen. Schon jetzt ist C++ auf dem besten Weg in diese Richtung, mit etwas gutem Willen kann man Datenobjekte mit Substantiven und Elementfunktionen mit Verben identifizieren, auch wenn dieser Vergleich nicht jeder detailierten Betrachtung standhält. Basisklassen könnte man als Adjektive ansehen, denn sie beschreiben die Eigenschaften der Objekte.

Wie dem auch sei, in 100 Jahren wird man nur noch mitleidig darüber lächeln können, womit man sich in den Anfangszeiten des elektronischen Urzeitalters abgab ...

Index

`inline`, 18
`friend`, 23
`inline`, 23
`mutable`, 62
`namespace`, 62
`using`, 63

abstrakt, 51
Ausnahmen, 60

Basisklasse, virtuell, 59
Basisklassen, 46
Bereichsoperator, 17

class, 23
Copy-Konstruktor, 19

Defaultkonstruktor, 19
Definition, 17, 18
Deklaration, 17, 18
Destruktor, 20

Elementfunktionen, 15
Exceptions, 60

Gültigkeitsbereiches, 17

Java, 63

Klasse, virtuell, 59
Konstruktor, 18
Konstruktorsyntax, 20

Laufzeitfehler, 60

memberfunction, 15
Modul, 18

Namespace-Pollution, 62

private, 22
public, 22

Runtime-Error, 60

Streammanipulatoren, 58

Vererbungsmechanismus, 46