8. Klassen und Objekte

Bislang haben wir uns in diesem Kurs nur mit grundlegenden Befehlen und Strukturen von C++ beschäftigt. Der größte Teil davon ist von Standard-C übernommen und somit nicht spezifisch für C++. In diesem Kapitel soll sich nun mit dem Konzept beschäftigt werden, das einer der wesentlichen Wechsel von Standard-C auf C++ darstellt: das Objektorientierte Programmieren (OOP).

8.1.Entstehung von OOP und Grundlagen

Ursprünglich dienten Computer ausschließlich dazu numerische Aufgaben zu lösen. Ihre Aufgabe war es Rechenaufgaben zu lösen, deren Umfang und Komplexität die Grenzen menschlichen Rechenvermögens erreichte oder gar überschritt. Ein typisches Beispiel für diese Verwendung ist das Berechnen von Logarithmentafeln, eine Aufgabe, für die viele der ersten Computer konstruiert wurden.

Für diese numerischen Zwecke war ein Programmierkonzept völlig ausreichend, das auf der einen Seite einen Satz von Daten vorsah (gegebenen und berechneten) und auf der anderen Seite Funktionen, um diese zu ändern. Hinzu kam, daß die Programme noch relativ klein waren (gemessen an heutigen Maßstäben) und nur für eine bestimmte Nutzungszeit, den sogenannten Software-Lebenszyklus, entworfen wurden.

Mit steigenden Anforderungen an die Computer wuchs aber der Umfang der Programme sehr stark. Auch beschränkten sich die Anwendungen nicht mehr nur auf das Lösen numerischer Probleme, sondern der Computer vollzog einen Wandel zu einem universellen Arbeitsgerät, bis hin zu Multimedia, Künstlicher Intelligenz und Computervermittelter Kommunikation.

Diese Entwicklung erforderte ein Umdenken auch bei dem Programmierkonzept. Es wurde nötig, Software in kleine Sinneinheiten aufzuteilen, die immer wieder verwendet werden können. Dabei stellte sich eine Aufteilung in Datensatz und Funktionalität, wie sie bislang üblich war, als sehr unhandlich heraus. Man strukturierte statt dessen die Programmteile in Funktionseinheiten, die die für sie benötigten Daten gleich mit enthielten, womit das Konzept der OOP geboren war.

Einer der zentralen Aspekte der OOP ist es, Daten mit den zugehörigen Funktionen zusammenzuführen. Diesen Prozeß bezeichnet man als *Kapselung*, die daraus resultierende Datenstruktur nennt sich *Klasse*, eine Variable oder Konstante, die diese Datenstruktur nutzt, nennt sich eine *Instanz* der Klasse. Der wesentliche Unterschied einer Klasse des OOP zu den schon bekannten Datenstrukturen (z.B. ein struct in C) ist, daß eine Klasse zusätzlich zu den Daten auch weitreichende Funktionalitäten übernehmen kann. Dadurch wird sie zu einem *Abstrakten Datentyp*, der zu den

Inhalten (den Daten) auch die Möglichkeiten spezifiziert, mit ihnen umzugehen (die Funktionen). Die Daten einer Klasse werden dabei als *Elemente*, und die Funktionen als *Elementfunktionen* oder *Methoden* bezeichnet.

Der zweite wichtige Aspekt der OOP ist die Möglichkeit der *Vererbung*. Um nicht jedesmal eine komplette neue Klasse programmieren zu müssen, kann man in den objektorientierten Sprachen angeben, daß eine Klasse alle Elemente und Methoden einer anderen Klasse 'ererben' soll, d.h. die neue Klasse enthält von vorneherein alle Daten und Funktionen der alten Klasse. Diese können dann um weitere ergänzt werden oder bei Bedarf auch überschrieben werden. Damit wird eine Beziehung wie z.B. 'ein Apfel ist eine Frucht' realisiert, durch zusätzliche Elemente und Methoden wird die neue Klasse genauer spezifiziert. Die alte Klasse wird dabei als *Basisklasse* bezeichnet, die neue als *abgeleitete* Klasse und die Struktur, die durch die Vererbung entsteht, als *Klassenhierarchie*.

Der dritte grundlegende Aspekt ist der des *Polymorphismus*. Wie erwähnt kann man in einer abgeleiteten Klasse Methoden der Basisklasse überschreiben. Dadurch hat die abgeleitete Klasse eine neue Methode, die aber immer noch den gleichen Namen trägt wie die Methode der Basisklasse. Das Prinzip des Polymorphismus besagt nun, daß zur Compilezeit noch nicht feststehen muß, welcher Klasse einer Hierarchie ein Datum angehört, solange gewährleistet ist, daß die benutzten Elemente und Methoden in allen möglichen Klassen vorhanden sind. Welche Methoden dabei aufgerufen werden, entscheidet sich dann nicht wie sonst üblich zur Compilezeit, sondern erst zur Laufzeit. Dies bezeichnet man auch als *späte* oder *dynamische Bindung* - im Gegensatz zur herkömmlichen *statischen Bindung*.

Aus den drei Grundprinzipien der OOP folgen bei konsequenter Umsetzung einige weitere Konzepte. Da ist zuerst das aus der Kapselung entstehende Konzept, daß Objekte ein Innen und ein Außen haben. Innen heißt dabei, daß es Elemente und evtl. auch Methoden gibt, die der Nutzer (d.h. der aufrufende Programmteil) dieses Objektes nicht sieht, während das Außen die Menge der Elemente und Methoden ist, die der Nutzer auch sehen kann. Objektorientierte Sprachen haben in der Regel die Möglichkeit, eine solche Unterscheidung zu treffen - in C++ werden Elemente und Methoden des inneren Bereichs *privat (private)* genannt, die des äußeren Bereichs *öffentlich (public)*.

Die Nutzung der privaten Datenstrukturen ist dann nur noch über öffentliche Methoden möglich, bei einem konsequent objektorienten Programmierstil sollten alle Elemente privat sein und der Zugriff auf die Elemente nur über Methoden erfolgen (sogenannte *Zugriffsmethoden*). Damit ist u.a. gewährleistet, daß man später die Datenstrukturen austauschen kann, ohne die Aufrufe der entsprechenden Zugriffsmethoden zu ändern - und damit auch keine Programme ändern muß, die

dieses Objekt nutzen. Außerdem ist es möglich, zusätzliche Effekte mit in die Zugriffsmethoden einzubauen, so z.B. ein Flag, das anzeigt, ob eine Änderung der Daten erfolgte. Wenn ein direkter Zugriff auf die Elemente der Klasse möglich wäre, wäre dieses Flag unzuverlässig.

Aus dem Konzept des Innen und Außen eines Objektes folgt, daß ein Objekt eine Außenseite hat. Diese Außenseite wird gebildet durch die Zugriffsmöglichkeiten auf Methoden und Elemente des Objektes. Sie werden vor allem von den Namen der öffentlichen Elemente und Methoden und der Parameter dieser Methoden bestimmt. Die Menge der Zugriffsmöglichkeiten wird als *Schnittstelle (Interface)* bezeichnet.

Der Aufruf einer Methode eines Objektes wird häufig als *Nachricht (Message)* bezeichnet. Dahinter steckt die intuitive Vorstellung, daß ein Programmteil durch eine Nachricht ein Objekt dazu bringt, etwas zu tun. Das kann dann durchaus auch nur ein Setzen eines Elementes sein. Wesentlich an diesem Konzept der Programmierung ist aber, daß niemals ein externer Programmteil das Objekt verändert, sondern immer nur eine Nachricht an das Objekt schickt, dies selbst zu tun. Damit kann ein Objekt immer die Kontrolle über sein Innen behalten.

8.2.Abstrakte Basisklassen

Ein besonderer Fall für Vererbung tritt dann ein, wenn man zwar eine Reihe von Objekten modellieren möchte, die gemeinsame Eigenschaften haben, aber keine direkten Beziehung zwischen den Objekten findet. Wenn man z.B. eine Klasse haben möchte, die ein Auto beschreibt, und eine andere, die ein Motorrad beschreibt, so findet man sicherlich viele Gemeinsamkeiten, aber weder ist ein Auto ein Motorrad, noch umgekehrt.

Damit macht es keinen Sinn, eine der beiden Klassen von der anderen abzuleiten. Trotzdem wäre es wünschenswert, eine gemeinsame Basisklasse zu haben, die die gemeinsamen Eigenschaften widerspiegelt. Damit könnte man zum einen Programmieraufwand bei der Erstellung der Klassen sparen, da man vieles nur einmal statt mehrfach schreibt.

Was wäre in unserem Beispiel denn eine mögliche gemeinsame Basisklasse? Oder anders gefragt: Was sind denn sowohl Auto als auch Motorrad? Beide sind Maschinen, aber das würde viele gemeinsame Eigenschaften außen vor lassen, so z.B. die Eigenschaft 'kann fahren'. Sinnvoll ist es, eine Klasse zu finden, der beide angehören, aber die möglichst viele (am besten alle) der gemeinsamen Eigenschaften vereint. Dies wäre hier so etwas wie eine Klasse Fahrzeug (im Sinne von Landfahrzeug mit Rädern), denn sowohl Auto als auch Motorrad sind Fahrzeuge im obigen Sinne und alle gemeinsamen Eigenschaften der beiden sollten in dieser Klasse vereinbar sein.

Bei einer Implementierung in OOP könnte man daher eine Fahrzeugklasse modellieren, von der dann die beiden anderen abgeleitet werden. Diese Klasse soll aber nicht instanziert werden, d.h. ihr selbst sollen keine Daten zugehören - sie dient einzig dem Zweck, Basisklasse für andere zu sein. Daher nennt man sie eine *Abstrakte Basisklasse*.

8.3.Beispiel

Als Beispielhierarchie nehmen wir eine Hierarchie von graphischen Objekten, da dies eine der anschaulichsten Verwendungen der Objektkonzepte ist. Ganz nebenbei war auch das erste objektorientierte Anwendungssystem 'Sketchpad' ein Graphiksystem - ein Vorläufer heutiger CAD-Systeme. Stellen wir uns also eine Reihe graphischer Objekte auf einem Bildschirm vor: Punkte, Rechtecke und Kreise. Zunächst stellt man fest, daß zwischen diesen Objekten keine 'A ist B'-Beziehung zu finden ist. Da aber all diese Objekte Eigenschaften wie z.B. eine Position, dargestellt durch zwei Koordinaten, und eine Farbe gemeinsam haben, erscheint es sinnvoll, sich eine abstrakte Basisklasse zu überlegen, von der man dann alle anderen ableitet. Die Methoden, die diese Klasse aufnimmt, sind zuerst die zu den Elementen gehörenden Zugriffsmethoden.

Außerdem sollten alle graphischen Objekte in der Lage sein, ihren Inhalt wiederzugeben, d.h. sich zu zeichnen. Auch eine Methode zum Löschen könnte sinnvoll sein. Des weiteren würde es für ein Graphiksystem wünschenswert sein, eine Option zum Verschieben der Objekte anzubieten - die zugehörige Methode müßte das Objekt löschen, die Koordinaten neu setzen und danach das Objekt wieder zeichnen. Dazu können die schon vorhandenen Methoden zum Zeichnen und Löschen genutzt werden.

Von dieser Klasse können dann die anderen abgeleitet werden. Der Punkt braucht keine weiteren Eigenschaften, das Rechteck soll zusätzlich noch eine Breite und eine Länge haben, außerdem eine Ausrichtung. Ähnlich wie das Verschieben aller Graphikobjekte, soll eine Funktion zum Drehen vorhanden sein. Im Gegensatz dazu hat der Kreis nur das zusätzliche Element Radius.

Damit ergeben sich dann folgende Klassen. Eventuelle Parameter werden dabei in Klammern angegeben, Rückgabewerte werden ignoriert.

Beispiel 1. Beispiel-Klassenhierarchie von Graphikobjekten

Klasse ***Graphikobjek***t:

- Elemente: ***XPosition***, ***YPosition***, ***Farbe***
- Methoden: ***leseX, leseY, leseFarbe, setzeX(neuesX), setzeY(neuesY), setzeFarbe(neueFarbe), zeichne, lösche, verschiebe(neuesX, neuesY***)

Klasse ***Punkt***, abgeleitet von Graphikobjekt, keine zusätzlichen Elemente und Methoden.

Klasse ***Rechteck***, abgeleitet von Graphikobjekt, zusätzlich:

- Elemente: ***Richtung, Breite, Länge***
- Methoden: ***leseRichtung, leseLänge, leseBreite, setzeRichtung(neueRichtung), setzeLänge(neueLänge), setzeBreite(neueBreite), drehe(neueRichtung)***

Klasse ***Kreis***, abgeleitet von Graphikobjekt, zusätzlich:

- Elemente: ***Radius***
- Methoden: ***leseRadius, setzeRadius(neuerRadius)***

Von diesen könnte man natürlich noch weitere ableiten, z.B. ein ausgefülltes Rechteck, das dann ein Rechteck ist (d.h. dessen Klasse von der Rechteckklasse abgeleitet wird) - es hätte dann ein zusätzliches Element für die Innenfarbe mit den entsprechenden Zugriffsmethoden.

Was haben wir damit gemacht? Wir haben das Konzept der Kapselung genutzt, um anschauliche Sinneinheiten - die Objekte - zu bilden. Dabei haben wir einem Objekt verschiedene Eigenschaften zugeordnet, zum einen Elemente, die den Zustand eines Objektes beschreiben, zum anderen Methoden, die Möglichkeiten bieten, das Objekt zu verändern oder andere Effekte auszulösen, hier z.B. das Zeichnen eines graphischen Objektes.

Außerdem haben wir das Konzept der Vererbung genutzt, um nicht jedesmal alle Daten und Funktionen neu angeben zu müssen, bzw. dann bei einer Implementierung auch neu programmieren zu müssen. Damit spart man sich Programmieraufwand und bei entsprechender Nutzung wird der Programmcode übersichtlicher, da weniger Elemente genutzt werden. So werden von den 15 Elementen und Funktionen, die die Klasse Kreis hat, nur drei explizit angegeben.

Dabei ist es vielleicht nicht ganz offensichtlich, warum dazu eine abstrakte Basisklasse benutzt wurde, wo sie doch genau die Eigenschaften hat, die die Klasse Punkt auch hat. Warum sollte man nicht einfach die Klasse Graphikobjekt in Punkt umtaufen und alle anderen von dieser ableiten, man würde damit doch eine Klasse sparen? Zum einen widerspräche das dem Konzept der Vererbung, da ein Rechteck oder ein Kreis sicherlich kein Punkt ist. Doch könnte man natürlich auch diese Interpretation der Vererbung aufgeben, da sie hier ja nicht sinnvoll erscheint.

Doch auch hier ist wie so oft bei OOP der Sinn des Konzeptes erst dann sichtbar, wenn man sich Gedanken zu eventuellen Änderungen macht. Wenn man später

Elemente oder Methoden zu der Klasse Punkt ergänzen möchte, da man weitere Eigenschaften eines Punktes modellieren möchte, die aber die anderen Objekte nicht haben sollen, dann ist dies ohne eine abstrakte Basisklasse nicht möglich, da sonst alle Eigenschaften des Punktes an die anderen Klassen vererbt würden. Bei einer sauberen objektorientierten Programmierung mit abstrakter Basisklasse kann man diese zusätzlichen Eigenschaften eines Punktes in die Klasse Punkt eintragen, während Ergänzungen für alle Klassen in die abstrakte Basisklasse Graphikobjekt eingetragen werden können.

Das einzige der drei Grundkonzepte, das wir noch nicht genutzt haben, ist das des Polymorphismus. Doch auch dieses Konzept kann hier Verwendung finden, und zwar bei den Methoden verschiebe und drehe. Was sollen diese Methoden machen? Sie sollen das entsprechende Objekt zuerst löschen, dann die Position bzw. Richtung des Objektes ändern und es dann wieder neu zeichnen. Dazu ist es zweckmäßig, die Methoden zeichne und lösche der Objekte zu nutzen.

Doch wenn man mit dem herkömmlichen Konzept der statischen Bindung in der Klasse Graphikobjekt die Methode verschiebe einführt, wird der Compiler eine Methode bauen, die ein abstraktes Graphikobjekt löscht und später wieder zeichnet. Dies kann aber nicht sinnvoll sein, da dies ja kein sichtbares Objekt sein soll.

Man könnte zwar stattdessen einfach einen Punkt zeichnen und löschen, aber dies würde dann auch in den Klassen Rechteck und Kreis passieren: es würde ein Punkt gelöscht und später gezeichnet. Dieses Problem kann man dadurch umgehen, daß man den Compiler anweist, die Methoden zeichne und lösche dynamisch zu binden. Dann werden nicht mehr die Methoden benutzt, die zur Compilezeit gültig sind (d.h. die von Graphikobjekt), sondern die der zur Laufzeit jeweils gültigen Klasse. Wenn dann die Methoden zeichne und lösche von den abgeleiteten Klassen überschrieben werden, ändert sich die Funktionalität der Methode verschiebe gleich mit.

8.4.Die Datenstruktur class

Objektorientierte Programmierung wird in C++ mit Hilfe der Datenstruktur class realisiert. Diese Datenstruktur ist der des schon gelernten struct ähnlich und realisiert die Implementierung einer Klasse im Sinne der OOP. Eine ganz einfache Klassendefinition sieht z.B. so aus:

Beispiel 2. Definition einer Klasse

```
class datum {
    private:
      int tag,monat,jahr;
    public:
      datum(){ tag=0; monat=0; jahr=0; };
    void setze(int t, int m, int j)
    {
      tag=t; monat=m; jahr=j;
```

```
    }
    void loesche()
    {
      tag=0; monat=0; jahr=0;
    }
    void ausgabe();
    int korrekt();
};
```

Hierbei fallen gegenüber dem Datentyp *struct* zwei neue Aspekte auf: zum einen die beiden Schlüsselwörter private und public, zum anderen die Tatsache, daß nicht nur Datentypen, sondern auch Funktionen (bzw. hier: Methoden) mit aufgeführt werden - teilweise mit Definitionen, teilweise auch nur die Deklarationen. Besondere Bedeutung hat dabei die erste Methode, die den Namen der Klasse trägt.

8.5.private und public

Die Schlüsselwörter private und public bezeichnen das Konzept der Kapselung – Objekte haben ein Innen und ein Außen in dem Sinne, daß sie auf einen Teil ihrer selbst nur selber zugreifen können, während andere Bereiche von allen Programmteilen zugreifbar sind. Damit ist eine Einteilung möglich, zum einen in von fremden Zugriff geschützte Bereiche und zum anderen in einen Teil, der für die Kommunikation mit anderen Programmteilen nötig ist.

Der Teil einer Klasse, der hinter *private:* steht, ist in den Instanzen der Klasse (d.h. den Daten, die diese Klasse als Typ haben) für andere Programmteile nicht zugreifbar, ein Zugriff auf die dort deklarierten Elemente und Methoden ist nicht zulässig.

Im Gegensatz dazu sind die Teile der Klasse, die hinter *public:* stehen, in den jeweiligen Instanzen öffentlich zugänglich, d.h. für andere Programmteile sichtbar und nutzbar. So könnte man den Tag von heute dadurch verstellen, daß man **heute.setze(*Tag,Monat,Jahr*)** aufruft. Das dann immer das ganze Datum verstellt werden muß, kann durchaus Absicht sein. Falls ein einzelnes Verstellen möglich sein soll, müßten entsprechende Methoden implementiert werden.

Die öffentlichen und privaten Bereiche sind dabei in einer Klasse beliebig mischbar, es kann nach einem öffentlichen Bereich wieder ein privater folgen und umgekehrt.

Das Schlüsselwort *private:* kann hier auch weggelassen werden, da die Voreinstellung für *class* ist, daß Elemente und Methoden privat sind. Erst nach *public:* werden sie öffentlich.

8.6.Unterschiede zu *struct*

Das Klassenkonzept in C++ ist als Erweiterung des struct aus Standard-C zu sehen. In C++ gehen die Ähnlichkeiten aber noch weiter. Hier sind die Konstrukte class und struct gleichwertig. Alles, was man mit Klassen machen kann, kann in C++ auch über

ein struct gemacht werden, einschliesslich der noch zu erläuternden Konstruktoren und Destruktoren.

Damit unterscheidet sich C++ deutlich von Standard-C, wo es zwar den gleichen Datentyp schon gibt, der dort aber nicht diese Möglichkeiten bietet. C++ ist diesbezüglich zwar abwärtskompatibel zu Standard-C, aber ein struct aus C++ muß nicht unbedingt in Standard-C verwendbar sein, da dort vor allem keine Methoden erlaubt sind.

Einen Unterschied zwischen class und struct gibt es in C++ aber auch. Dieser ist so gering, daß er leicht übersehen werden kann, falls man ein struct zu einer vollwertigen Klasse ausbaut: ***die Voreinstellung für den Zugriff auf Elemente und Methoden ist in einem struct der öffentliche Zugriff, nicht der private.*** Falls man im Beispiel auf das *private:* am Anfang verzichten würde, und class durch struct ersetzen würde, wären die drei Integerelemente der Klasse öffentlich, ein Zugriff auf sie also erlaubt.

8.7.Methoden

Eine *Methode* ist im Sinne der OOP eine Funktion, die Element eines Objektes ist. Daher werden Methoden teilweise auch als *Elementfunktionen* bezeichnet. In C++ wird eine Methode eines Objektes genauso wie ein Datum eines Objektes deklariert, die Unterscheidung zwischen Datum und Funktion erfolgt hier - genauso wie immer - über die nachfolgende Parameterliste in runden Klammern, die natürlich auch leer sein kann, falls keine Parameter übergeben werden. Ein Beispiel - vorerst noch nicht ganz im Sinne der OOP, da alle Elemente öffentlich sind:

Beispiel 3. Methoden - public

```
class Uhrzeit
 {
   public:
     int stunde,minute;
     int korrekt();
 };
```

Auf die Methode kann man dann auch wie auf die Daten mit dem Punkt-Operator "." zugreifen. Im Beispiel sähen die Zugriffe auf Daten und die Methode dann so aus:

Beispiel 4. Aufruf von Methoden

```
// [...]
 Uhrzeit jetzt;
 // [...]
 cout<<"Jetzt ist es "<<jetzt.stunde<<":"<<jetzt.minute<<endl;
 if (!jetzt.korrekt()) cout<<"Diese Uhrzeit gibt es nicht!!"<<endl;
 // [...]
```

Das vorgestellte Programmfragment wird natürlich nur laufen, wenn die Methode korrekt() nicht nur deklariert, sondern auch definiert ist. Dazu gibt es zwei Möglichkeiten, zum einen eine Deklaration direkt mit der Definition innerhalb des

Objektes und zum anderen eine spätere Definition außerhalb des Objektes. Im ersten Fall sähe dies so aus:

Beispiel 5. Defition einer Methode innerhalb der Klassendefinition.

```
class Uhrzeit
  {
    public:
      int stunde,minute;
      int korrekt()
      {
      return((stunde>=0)&&(stunde<24)&&(minute>=0)&&(minute<60));
      }
};
```

Im zweiten Fall käme irgendwann nach der Deklaration dann später die Definition, wobei hier die Zugehörigkeit zu der Klasse mit dem *Scope-Operator* :: angezeigt wird:

Beispiel 6. Defition einer Methode ausserhalb der Klassendefition – mit Hilfe des *scope*-Operators

```
class Uhrzeit
{
    public:
      int stunde,minute;
      int korrekt();
};

int Uhrzeit::korrekt()
{
    return((stunde>=0)&&(stunde<24)&&(minute>=0)&&(minute<60));
}
```

Eine Methode sollte man aber nur dann innerhalb der Deklaration einer Klasse definieren, wenn sie sehr kurz ist. Gedacht ist diese Form der Definition vor allem für die Zugriffsmethoden, die ja häufig nur einen Wert zurückgeben oder setzen. Daher werden derart definierte Methoden auch immer so interpretiert, als ob sie mit dem Schlüsselwort *inline* gekennzeichnet wären. Es wird bevorzugt also kein Funktionsaufruf generiert, sondern die Funktion wird so compiliert, als ob sie im aufrufenden Programm selbst stände.

8.8.Konstruktoren

Eine besondere Form der Methode ist der Konstruktor. Wie schon erwähnt ist ein Konstruktor eine besondere Methode, die den Namen der Klasse trägt. Er wird nicht wie andere Methoden aufgerufen, sondern wird bei einer Instanzierung der Klasse implizit aufgerufen, d.h. dann, wenn ein Datum vom Typ dieser Klasse definiert wird.

Genauso wie ein Konstruktor implizit aufgerufen wird, so wird er auch vom Compiler geliefert, falls er nicht explizit angegeben wird. Dies muß schon deswegen geschehen, weil durch den Konstruktor der für das zu instanzierende Objekt nötige Speicherplatz reserviert wird. Aber ein Konstruktor kann noch viele weitere Funktionen

übernehmen. So können in einem Konstruktor Daten gesetzt werden, kann Speicherplatz reserviert werden oder kann gezählt werden, wieviele Instanzen von diesem Objekt vorhanden sind.

Wie ein Konstruktor deklariert wird, haben wir schon am anfänglichen Beispiel gesehen. Er wird wie andere Methoden definiert, trägt aber als Namen den Namen der Klasse - im Beispiel sieht er so aus:

Beispiel 7. Konstruktordefinition

```
datum::datum()
{
    tag=0; monat=0; jahr=0;
};
```

Ein Konstruktor, der wie dieser keine Parameter hat, heißt *Standard-* oder *Defaultkonstruktor*.

Aber auch Parameter können genutzt werden, so könnte man den Konstruktor der Beispielklasse mit einem zusätzlichen der Form:

Beispiel 8. Konstruktor mit Parametern.

```
datum(int T, int M, int J)
{
    tag=T; monat=M; jahr=J;
}
```

überladen. Die Parameter für den Konstruktor würden dann bei der Instanzierung der Klasse, d.h. bei der Definition eines Datums der Klasse angegeben:

Beispiel 9. Instanzierung der Klasse und Aufruf des Konstruktors

```
datum petersGeburtstag=datum(22,11,1971);
```

oder kürzer:

Beispiel 10. Kürzere Schreibweise für den Konstruktor-Aufruf.

```
datum petersGeburtstag(22,11,1971);
```

Dabei können wie üblich die Mechanismen der default-Werte ausgenutzt werden:

Beispiel 11. default-Parameter in einem Konstruktor.

```
datum(int T=0, int M=0, int J=0)
{
    tag=T; monat=M; jahr=J;
}
```

In den vorherigen Beispielen wurden die Elemente einer Klasse immer *innerhalb* des Konstruktorblocks initialisiert (also ihnen ein Wert zugewiesen), also

Beispiel 12. Initialisierung von Klassenelementen.

```
datum(int T, int M, int J)
{
    tag=T; monat=M; jahr=J;
}
```

Für einfache Typen (integer, double usw. ist das nicht weiter ein Problem. Sind aber Klassen selber wieder Element einer Klasse, so bedeutet diese Schreibweise, daß beim Aufruf des Konstruktors zuerst die Defaultkonstruktoren der Elemente aufgerufen werden und anschließend eine Zuweisung gemacht wird. Bei komplexeren Klassen möchte man vielleicht den Aufruf des Default-Konstruktors vermeiden.

Die bessere Schreibweise ist Konstruktoraufruf und Zuweisung für Klassenelemente durch den Aufruf eines entsprechenden Konstruktors mit Parametern zu ersetzen. Dies ist sicherer, schneller und einfacher zu kodieren. In obigem Beispiel würde dann geschrieben werden:

Beispiel 13. Aufruf von Konstruktoren im Konstruktor.

```
datum(int T, int M, int J) : tag(T), monat(M), jahr(J) {}
```

In diesem Fall ist dann der Anweisungsfall des Konstruktors sogar leer. Es können natürlich bei Bedarf zustätzliche Anweisungen angegeben werden.

Der *Standardkonstruktor* oder auch *Defaultkonstruktor* ist ein Konstruktor ohne Parameter. Er sorgt dafür, daß eine Klasse wie die Standardtypen definiert werden kann:

Beispiel 14.Aufruf von einem Standardkonstruktor.

```
TYP NameDerInstanz;
```

Durch diese Kurzform der Definition fügen sich Objekte in C++ in die gewohnten Programmierschemata aus Standard-C nahtlos ein. Man könnte zwar auch ein Datum mit Hilfe des Standardkonstruktors durch

Beispiel 15. Das gleiche wie oben, aber umständlicher.

```
datum gestern= datum();
```

aufrufen, aber diese Form wäre sicherlich unpraktisch und auch ungewohnt.

Falls eine Klasse keine Definition eines Konstruktors enthält, wird ein Standardkonstruktor vom Compiler geliefert. Dieser initialisiert die Klasse im wesentlichen dadurch, daß die Standardkonstruktoren der Elemente aufgerufen werden - Instanzen von Basistypen werden dabei auf Null gesetzt. Der Standardkonstruktor wird aber nicht mehr vom Compiler generiert, wenn ein anderer Konstruktor definiert ist. Dann muß der Programmierer der Klasse diesen selber liefern, ansonsten wird davon ausgegangen, daß ein Standardkonstruktor nicht gewünscht wird. Dies kann ja durchaus der Fall sein, wenn eine Klasse nicht ohne Parameter sinnvoll initialisierbar ist - so ist die Definition des Standardkonstruktors der aktuellen Beispielklasse ja auch nicht unbedingt sinnvoll.

Ein weiterer Spezialfall der Konstruktoren sind Konstruktoren zur *Typumwandlung*, d.h. solche, die einen Parameter eines Typs aufnehmen, um damit die zu erzeugende Instanz des Objektes zu initialisieren. Auf diese Art kann eine Instanz des

Parametertyps in eine Instanz des Objektes umgewandelt werden, zu der der Konstruktor gehört. Dies kann z.B. bei einer Klasse für komplexe Zahlen geschehen, um reelle Zahlen umzuwandeln:

Beispiel 16. Konstruktor zur Umwandlung von Zahlen: von real zu komplex.

```
class complex
  {
    private:
      float real, imag;
    public:
      complex(float r) { real=r; imag=0; }
  }; //class complex
```

Dieses Beispiel hätte allerdings noch das Problem, daß kein Standardkonstruktor mehr vorhanden wäre, der aber sicherlich wünschenswert wäre. Eine einfache Möglichkeit diesen und einen weiteren, der sowohl Real- als auch Imaginäranteil einer komplexen Zahl aufnimmt, zu erzeugen, erhält man durch:

Beispiel 17. Derselbe Konstruktor wie oben, aber auch mit default-Werten.

```
complex(float r=0, float i=0) { real=r; imag=i; }
```

Solche Standardparameter, wie in Kapitel Funktionen beschrieben, können den Sourcecode stark verkleinern, man sollte dabei aber aufpassen, daß die Übersichtlichkeit nicht leidet.

8.9.Statische Elemente

Die Elemente eines Objektes haben eine Eigenschaft, die nicht immer wünschenswert sein muß: sie sind *objektspezifisch*, d.h. sie gelten für jedes Objekt einer Klasse einzeln und nie für alle zusammen. Es kann aber sinnvoll sein, *klassenspezifische* Elemente zu haben, so z.B. für einen gemeinsamen Zähler wie eine Kunden- oder Seriennummer. Solche Werte lassen sich nur schwer in allen Objekten gleichzeitig verwalten und man würde viel mehr Speicher benötigen, wenn jedes Objekt einen solchen Wert speichern müßte.

In C++ können Elemente ebenso wie Methoden daher als klassenspezifisch deklariert werden. Sie werden dann als *statisch* bezeichnet, da sie sich ja genauso wie statische Variablen verhalten sollen, nur daß sie hier nicht mehr statisch bzgl. mehrerer Aufrufe einer Funktion sind, sondern daß sie statisch bzgl. mehrerer Instanzierungen einer Klasse sind. Aufgrund dieser Ähnlichkeit wird auch das gleiche Schlüsselwort genutzt: static. Statische Funktionen dürfen dabei keine nicht-statischen Elemente verändern.

Die Definition und Initialisierung eines statischen Elementes erfolgt dabei ausserhalb der Klasse mit Hilfe des Scopeoperators ::. Im Beispiel auf der folgenden Seite sähe eine Initialisierung dann so aus:

Beispiel 18. Defition eines statistischen Elements.

```
int gast::zaehler=0;
```

Statische Elemente und Methoden lassen neben den normalen objektbezogenen Aufrufen (Objekt.Methode) auch klassenbezogene Aufrufe zu (Klasse::Methode). So kann der Zähler der Beispielklasse auch über

Beispiel 19. Aufruf des Zählers unabhängig von den Instanzen der Klassen.

```
gast::zeigeAnzahl();
```

aufgerufen werden, egal ob Instanzen existieren oder nicht.

Hier nun die Teile der schon erwähnten Beispielklasse, die Gäste zählt und nach dem 100. Gast vor Überfüllung warnt:

Beispiel 20. Klasse mit einem static Element.

```
class gast
  {
    private:
      static int zaehler;
      // [weitere Elemente]
    public:
      gast();
      // [weitere Methoden]
      static void zeigeAnzahl(){ cout<<"Gäste :"<<zaehler<<endl; }
  }; // class gast

gast::gast(){
    if (zaehler<100) {
      cout<<"Hallo neuer Gast"<<endl;
      zaehler++;
      // [weitere Initialisierung]
    }
    else
      cout<<"Zuviele Gäste!"<<endl;
  } // gast()
```

8.10. Destruktoren

Neben den Konstruktoren gibt es auch noch das Gegenstück: die Destruktoren. Sie werden aufgerufen, wenn der Compiler das Objekt löscht, d.h. dann, wenn er den zugehörigen Speicherplatz freigibt. Sie dienen zum einen dazu, Speicherplatz, den der Konstruktor angefordert hat wieder freizugeben (siehe Kapitel Pointer), können aber auch andere Funktionen wie z.B. das Schließen von Dateien ausführen. Er wird ähnlich wie der Konstruktor auch als Methode deklariert, die den Namen der Klasse trägt - nur hier steht noch eine Tilde vor dem Namen (~). Diese Tilde steht als Symbol für die Komplementierung - der Destruktor ist also auch schreibtechnisch das Gegenteil des Konstruktors. Üblicherweise steht der Destruktor direkt unter den Konstruktoren, er hat niemals Parameter oder einen Rückgabewert.

Ein Destruktor in der vorgestellten Klasse zum Zählen von Gästen könnte einen Gast wieder abmelden, d.h. die Anzahl der Gäste verringern. Dazu müßte in der Deklaration der Klasse auch der Destruktor deklariert werden, in dem man folgende Zeile nach dem Konstruktor ergänzt:

```
~gast();
```

Eine mögliche Implementierung sähe dann so aus:

Beispiel 21. Destruktor

```
gast::~gast()
  {
    if (zaehler>0)
    {
      zaehler--;
      cout<<"Auf Wiedersehen!"<<endl;
    }
    else
      cout<<"Schwerer Fehler!!"<<endl; //sollte nicht passieren
  } //~gast()
```

8.11. Kopierkonstruktor

Ein weiterer Konstruktor, der im Zusammenhang mit Zählern sinnvoll ist, ist der *Kopierkonstruktor* bzw. neudeutsch *Copykonstruktor*. Er dient dazu, ein Objekt zu kopieren, d.h. ein neues Objekt genau den gleichen Inhalt zu geben wie einem schon vorhandenen. Dabei wird bei der Instanzierung des neuen Objektes das alte als Parameter übergeben. Der Aufruf des Kopierkonstruktors sähe für die Beispielklasse so aus:

Beispiel 22. Aufruf eines Kopierkonstruktors.

```
gast neuerGast(alterGast);
```

Alternativ läßt sich dies natürlich auch mit dem Zuweisungsoperator machen:

Beispiel 23. Alternativ zu Kopierkonstruktor, aber gefährlich und unschön.

```
gast neuerGast;
neuerGast = alterGast;
```

Bei dieser Vorgehensweise können aber Probleme auftauchen. Der Compiler kopiert bei einer Zuweisung mit dem Zuweisungsoperator standardmäßig das alte Objekt elementweise in das neue Objekt. Falls dabei Elemente selbst zusammengesetzt sind, entsteht dabei eine Rekursion. Dabei wird allerdings nicht der Zähler mit hochgezählt, um dies zu gewährleisten muß man einen Kopierkonstruktor definieren.

Dabei erzeugt ein Gleichheitszeichen bei der Deklaration (etwa wie folgt)

Beispiel 24. Impliziter Aufruf eines Kopierkonstruktors.

```
gast neuerGast = alterGast;
```

keinen Aufruf des Zuweisungsoperators, sondern des Kopierkonstruktors. Diese Unterscheidung ist wichtig, wenn man beide Methoden explizit programmiert. Der Zuweisungsoperator wird nur benutzt, wenn das Objekt bereits initialisiert wurde.

8.12. Konstante Objekte

Wie andere Daten auch können Objekte als konstant deklariert werden, indem bei Ihrer Deklaration das Schlüsselwort const vorangestellt wird. Dabei ist aber eine Besonderheit zu beachten: Um wirklich sicherstellen zu können, daß sich das Objekt

wirklich nicht ändert, können im Normalfall keine Methoden des Objektes mehr ausgeführt werden.

Falls man aber möchte, daß Methoden auch bei konstanten Objekten ausgeführt werden können, da sie keine Elemente verändern, so kann man dies dem Compiler explizit mitteilen, indem man ihnen ebenfalls das Schlüsselwort const voranstellt. Hier bedeutet es nicht, daß die Methode konstant ist, sondern daß sie das Objekt konstant läßt. Ein typisches Beispiel wäre eine Ausgabemethode, die nur die Elemente des Objektes auf dem Bildschirm ausgibt, wie die Methode zeigeAnzahl im Beispiel, sie könnte auch wie folgt deklariert werden (das static bleibt einfach mit bestehen):

Beispiel 25. Static Methode

```
static int zeigeAnzahl() const { cout<<"Gäste :"<<zaehler<<endl; }
```

Zum Ende muss man noch erwähnen, daß in C++ alle Variablen wie Objekte initialisiert werden können. So kann eine Integervariable z.B. folgendermaßen erklärt werden:

Beispiel 26. Initialisierung einer Variable wie Klasse.

```
int a(5);
```

8.13. Aufteilung in Dateien

Die Aufteilung von Objekten in Source- und Headerfiles ist besonders interessant, da Objekte mit ihrem Deklarationsteil eine besonders elegante Schnittstelle liefern. Wie auch bei Funktionen wird dieser in das headerfile geschrieben, während die Implementierung des Objektes, d.h. die Definitionen der Methoden, in das Sourcefile kommt.

Dabei ist eventuell noch eine Ausnahme für Methoden denkbar, die implizit als inline programmiert wurden, indem sie mit der Deklaration des Objektes auch definiert wurden. Aber auch hier bietet sich ein Wechsel zu einer Trennung an, um die Aufteilung konsequent durchzuhalten. Methoden, die inline compiliert werden sollen, können dabei ja explizit mit dem Schlüsselwort inline versehen werden. Spätestens dann, wenn man Code kommerziell vertreiben möchte, wird eine Implementierung im headerfile auch unerwünscht werden, da dieses ja mit ausgeliefert werden muß.

8.14. Aufgaben