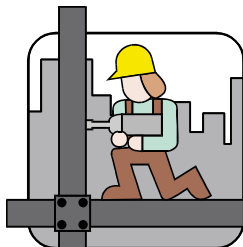


## 2.6 Konstruktoren und Destruktoren

Wenn Sie bereits mit anderen Programmiersprachen gearbeitet haben, wird Ihnen das Konzept der Konstruktoren und Destruktoren zunächst etwas merkwürdig vorkommen — handelt es sich doch um Funktionen, die aufgerufen werden, ohne dass ihr Aufruf im Programmtext steht! Mit der Zeit werden Sie aber die Abläufe verstehen und merken, dass auch da kein Geheimnis dahinter steckt.

### 2.6.1 Überblick über Konstruktoren

**Abbildung 2.12**  
Konstruktoren werden beim Erzeugen eines Objekts aufgerufen.



Ein Konstruktor dient dazu, ein Objekt in einen definierten Anfangszustand zu versetzen, das heißt Speicherplatz für die Attribute bereitzustellen und gegebenenfalls die Attribute mit sinnvollen Anfangswerten zu initialisieren.

#### Besondere Eigenschaften

Konstruktoren haben einige besondere Eigenschaften, die sie von allen anderen Methoden unterscheiden:

- ❑ *Aufrufe von Konstruktoren werden automatisch in das Programm eingefügt.* Jedes Mal, wenn ein Objekt erzeugt wird und die Klasse über einen entsprechenden Konstruktor verfügt, wird dieser aufgerufen. Explizite Konstruktoraufrufe gibt es also nicht.
- ❑ *Konstruktoren tragen denselben Namen wie die Klasse.* Um eine Methode als Konstruktor zu kennzeichnen, müssen Sie ihr in C++ denselben Namen geben, den auch die Klasse hat. Alle anderen Namen können keine Konstruktoren bezeichnen.
- ❑ *Konstruktoren haben keine Rückgabewerte (auch nicht `void`).* Sie dürfen keinen Rückgabetyt bei der Deklaration eines Konstruktors angeben und auch keine `void`-Anweisungen mit Argument verwenden.

Ein einfaches, schon klassisches Beispiel für einen selbst definierten Datentyp ist ein `Datum`, das einen Tag im Kalender repräsentiert. Ich will in diesem Abschnitt eine solche Klasse mit Ihnen aufbauen, um die verschiedenen Arten von Konstruktoren deutlich zu machen. Die Ausgangsversion sei:

```
class Datum
{
private:
    unsigned int t, m, j;

public:
    void setze(unsigned int _t,
               unsigned int _m, unsigned int _j);
    void setzeAufHeute();
    void ausgeben() const;
};
```

Hier haben wir wieder die Klassenstruktur vor uns, die wir im vorletzten Abschnitt kennen gelernt haben. Die Attribute `t`, `m`, `j` (für Tag, Monat und Jahr) sind `private` Datenelemente, können also nur in Methoden desselben Objekts gelesen und verändert werden. Um sie von außen zu setzen, gibt es die öffentliche Methoden `setze()`, die das Datum auf die angegebenen Werte setzt, sowie `setzeAufHeute()`, die das Datumsobjekt mit dem aktuellen Systemdatum initialisiert. Die dritte öffentliche Methode `ausgeben()` ist als konstant deklariert (`const` hinter den runden Klammern); das deutet darauf hin, dass darin nur Attribute gelesen, jedoch nicht verändert werden — was ja auch von einer Ausgabefunktion erwartet werden darf. Konstante Methoden dürfen auch für konstante Referenzen des Objekts aufgerufen werden.

### Arten von Konstruktoren

Je nach Form und Aufgabe unterscheidet man mehrere Arten von Konstruktoren:

- ❑ *Standardkonstruktor* (englisch *default constructor*). Dieser wird bei jeder Erzeugung eines Objekts der Klasse verwendet, wenn kein anderer Konstruktor in Frage kommt.
- ❑ *Allgemeiner Konstruktor* (Seite 98). Dieser kann beliebige Argumente haben und wie eine Methode überladen werden. Es können also mehrere Konstruktoren mit verschiedenen Argumentlisten existieren. Auch Vorgabewerte für die Argumente sind erlaubt.
- ❑ *Kopierkonstruktor* (Seite 102). Dieser dient dazu, ein Objekt dieser Klasse mit einem anderen derselben Klasse zu initialisieren. Er erhält dazu als Argument eine konstante Referenz auf das Objekt.

- *Typumwandlungskonstruktor* (Seite 105). Dieser hat nur ein Argument und dient dazu, einen anderen Datentyp in die Klasse (als Typ gesehen) umzuwandeln.

Im Folgenden wollen wir die verschiedenen Konstruktoren genauer betrachten.

## 2.6.2 Standardkonstruktor

Der Standardkonstruktor, der leider viel zu oft auch im Deutschen mit dem englischen Ausdruck »default constructor« bezeichnet wird, hat keine Argumente. Er wird immer dann aufgerufen, wenn ein Objekt dieser Klasse ohne weitere Angaben erzeugt wird.

*Deklaration* Bei unserem Beispiel lautet die Deklaration:

```
class Datum
{
private:
    unsigned int t, m, j;

public:
    Datum(); // Standardkonstruktor
    void setze(unsigned int _t,
               unsigned int _m, unsigned int _j);
    void setzeAufHeute();
    void ausgeben() const;
};
```

*Definition* Bei einem Datums-Objekt ist es sehr unschön, wenn sich die Attribute in einem undefinierten Zustand befinden, dies gilt auch für den Zeitpunkt unmittelbar nach seiner Erzeugung. Selbst ein »0.0.0« ist als Datum unbrauchbar. Ganz praktisch könnte es dagegen sein, wenn das Objekt gleich mit dem heutigen Datum vorbelegt wird. Genau das erledigt unser Konstruktor:

```
Datum::Datum()
{
    setzeAufHeute();
}
```

Konstruktoren werden oft als `inline`-Methoden implementiert (siehe Seite 88), also direkt in die Klassendeklaration geschrieben. Dann ist auch dem Leser der Header-Datei sofort klar, mit was ein Objekt durch den Konstruktor eigentlich initialisiert wird. So wie der Konstruktor hier steht, würde man ihn am ehesten in die Header-Datei nach der Klassendeklaration schreiben.

Ich sprach oben davon, dass eine der Aufgaben des Konstruktors sei, Speicherplatz für die Attribute bereitzustellen. Vielleicht fragen Sie sich nun, warum Sie davon in dieser Methode nichts sehen. Ganz einfach: Es passiert automatisch. Ebenso wie bei lokalen Variablen — unabhängig davon, ob ihr Typ einfach oder eine Klasse ist — kann das System selbstständig ermitteln, wie viel Speicher die Attribute benötigen, und diesen entsprechend reservieren. Das ändert sich erst, wenn Sie selbst Felder oder Objekte dynamisch anlegen wollen, das heißt selbst die Anweisung zur Speicherreservierung geben. Auch das kann innerhalb eines Konstruktors geschehen. Doch dazu kommen wir später noch.

*Speicherplatz für die Attribute*

Jetzt sehen wir uns an, wo der Konstruktor aufgerufen wird. In einem Hauptprogramm steht beispielsweise:

*Aufruf des Konstruktors*

```
int main()
{
    Datum heute; // Impliziter Konstruktoraufruf
    heute.ausgeben();
    //...
```

An der Stelle, wo ein Objekt vom Typ `Datum` erzeugt wird, wird auch dessen Konstruktor aufgerufen.

Natürlich können Sie auch mit der Klasse arbeiten und Objekte davon erzeugen, wenn diese keinen Konstruktor hat. Allerdings sind dann die Attribute in einem unbestimmten Zustand — genauso wie Variablen, die Sie nur deklariert, aber nicht initialisiert haben. Ich möchte Ihnen daher empfehlen, bei allen Ihren Objekten zumindest einen Standardkonstruktor zu definieren.

Achten Sie aber darauf, im Konstruktor wirklich nur die nötigsten Initialisierungen vorzunehmen und ihn möglichst klein zu halten. Da er beispielsweise keinen Rückgabewert hat, werden mögliche Fehler nicht ohne Weiteres vom Programm bemerkt. Richten Sie im Zweifelsfall lieber eine zusätzliche Methode ein, in der dann kritische Initialisierungen vorgenommen werden können.



## Hintergrund

Die Einfügung des Konstruktoraufrufs klappt übrigens auch bei Verschachtelungen. Wenn Sie beispielsweise eine Klasse haben wie:

```
class Tabelleneintrag
{
private:
    Datum datum;
    //...

public:
```

```

    Tabelleneintrag();
    // ...
};

```

so wird der Konstruktor von `Datum` automatisch bei der Abarbeitung des Konstruktors von `Tabelleneintrag` aufgerufen, und zwar noch vor der ersten Anweisung im Funktionskörper von `Tabelleneintrag::Tabelleneintrag()`.

### 2.6.3 Allgemeine Konstruktoren

Allgemeine Konstruktoren haben Argumente und können wie normale Methoden überladen werden, das heißt, es darf mehrere Konstruktoren mit unterschiedlichen Parameterlisten geben. Auch Vorgabewerte für die Parameter sind erlaubt.

Wir wollen unsere Klasse um eine vollständige Initialisierung erweitern sowie um eine, die nur den Tag enthält.

```

class Datum
{
public:
    Datum();
    Datum(unsigned int _t,
           unsigned int _m, unsigned int _j);
    Datum(unsigned int _t);
    // ...
};

```

Mit dem ersten neuen Konstruktor wollen wir einfach alle drei Datumsteile setzen; da wir für diese Aufgabe schon die Methode `setze()` vorgesehen haben, können wir sie auch gleich verwenden:

```

Datum::Datum(unsigned int _t,
             unsigned int _m, unsigned int _j)
{
    setze( _t, _m, _j);
}

```

*Unterscheidung von  
Parametern und Attributen*

Die Unterstriche sind eine persönliche Konvention von mir; auf diese Weise kann ich Parameter der Methode besser von Attributen unterscheiden. Manche Programmierer machen es aber auch genau umgekehrt und beginnen die Namen aller Attribute mit einem Unterstrich oder einem `m_` von »member«. Wie Sie es damit halten, überlasse ich ganz Ihnen; nur sollten Sie mit Ihrer Konvention konsequent sein und diese überall durchhalten. Denn der Zweck solcher Regeln ist ja letztlich, den Code für sich und andere möglichst lesbar zu halten.

*Einheitliche Initialisierung*

Die gerade gezeigte Vorgehensweise empfiehlt sich übrigens auch allgemein: Wenn Sie zulassen möchten, dass Ihr Objekt entweder von einem

allgemeinen Konstruktor oder einem expliziten Methodenaufruf initialisiert wird, verwenden Sie einfach im Konstruktor auch diese Methode. Auf diese Weise haben Sie den Code konsistent gehalten und die unnötige Verdopplung von Anweisungen vermieden.

Unser zweiter neuer Konstruktor soll nur den Tag als Parameter haben, wobei Monat und Jahr dieselben wie für den heutigen Tag sein sollen.

*Konstruktor mit einem Parameter*

```
Datum::Datum(unsigned int _t)
{
    setzeAufHeute();
    t = _t;
}
```

Was hier noch fehlt, ist eine Plausibilitätsprüfung, denn nicht jede Zahlenkombination ist ja ein gültiges Datum. Vielleicht überlegen Sie sich einmal selbst, wie eine solche aussehen könnte.

Auch allgemeine Konstruktoren werden automatisch bei der Definition eines Objekts aufgerufen. Die Argumente geben Sie dabei in Klammern hinter dem Objektnamen an.

*Aufruf allgemeiner Konstruktoren*

```
int main()
{
    // Standardkonstruktor
    Datum heute;
    // Konstruktor mit 3 Argumenten
    Datum ostern(4,4,1999);
    // Konstruktor mit 1 Argument
    Datum gestern(20);
    // ...
}
```

Der Compiler sucht dabei anhand der Anzahl und Typen der Parameter, welchen Konstruktor er aufrufen muss.

## Hintergrund

Manchmal möchte man auch vorschreiben, dass Objekte nur über einen allgemeinen Konstruktor erzeugt werden dürfen, das heißt nur mit Angabe eines Parameters. Sie können dieses Ziel unter anderem dadurch erreichen, dass Sie den Standardkonstruktor weglassen. Fehlt dieser nämlich, wenn gleichzeitig ein anderer allgemeiner Konstruktor vorhanden ist, bricht der Compiler mit einem Fehler ab, sobald eine Instanz der Klasse erzeugt werden soll.

Besonders elegant ist diese Möglichkeit jedoch nicht, zumal der Anwender der Klasse die resultierende Fehlermeldung nicht sofort im Sinne des Entwicklers interpretieren würde. Ein anderer Weg ist die Deklaration des Standardkonstruktors als `private`. Auf diese Weise ist aus Sicht des Compilers ein solcher Konstruktor vorhanden, wird also nicht als fehlend gemeldet. Versucht ein Benutzer der Klasse

*Privater Konstruktor*

jedoch, eine Instanz ohne Angabe eines Parameters zu erzeugen, meldet der Compiler, dass der dazu notwendige Standardkonstruktor nicht öffentlich ist und daher von außen nicht aufgerufen werden darf. Dem Benutzer bleibt somit nichts anderes übrig, als einen der öffentlichen allgemeinen Konstruktoren zu verwenden und einen Parameter anzugeben.

*Anwendung:  
Wrapper-Klassen*

Diese Vorgehensweise bietet sich beispielsweise bei so genannten *Wrapper-Klassen* an, also Klassen, die keine eigene Funktionalität haben, sondern nur einer anderen Klasse eine neue Schnittstelle geben. Nehmen wir etwa an, Sie hätten obige Klasse `Datum` fertig implementiert und auch in einigen anderen Funktionen und Klassen eingesetzt; nun wollen Sie Ihren Code in einem anderen Projekt wieder verwenden. Dummerweise schreiben die Arbeitsrichtlinien für dieses Projekt aber vor, dass alle Bezeichner auf Englisch sein müssen. Anstatt nun die Methoden umzubenennen und dabei zu riskieren, etwas zu vergessen, schreiben Sie einfach einen Wrapper.

```
class Date
{
private:
    Datum* dat;
    Date(); // privater Konstruktor

public:
    Date(Datum* _dat) :
        dat(_dat) {}
    void set(unsigned int _t,
            unsigned int _m, unsigned int _j)
        { dat->setze(_t, _m, _j); }
    void setToToday()
        { dat->setzeAufHeute(); }
    void print() const
        { dat->ausgeben(); }
};
```

Hier kann keine Instanz erzeugt werden, ohne dass die Elementvariable `dat` initialisiert wird. Da die anderen Methoden auch direkt auf sie zugreifen, ist diese Restriktion sinnvoll und notwendig. Will ein Benutzer der Klasse mittels der Zeile

```
Date myDate;
```

ein Objekt vom Typ `Date` erzeugen, so erhält er vom Compiler die schlichte Meldung: »*Date::Date() is private within this context.*«

## 2.6.4 Initialisierung mit Listen

Beim Aufruf eines Konstruktors wird noch vor dem Betreten des Methodenkörpers Speicherplatz für die Datenelemente bereitgestellt. Anschließend wird erst der Code in diesem Block abgearbeitet; darin haben wir bisher die Datenelemente mit Werten versehen.

Auf diese Weise wird aber doppelt auf die Attribute zugegriffen: einmal bei der Erzeugung und einmal bei der Zuweisung. C++ bietet die Möglichkeit, beides in einem Schritt zusammen zu erledigen. Dazu verwendet man eine so genannte *Initialisierungsliste*. Dabei geben Sie hinter der schließenden runden Klammer der Argumentliste, getrennt durch einen Doppelpunkt, die Attribute an, wobei die zu verwendenden Werte in runden Klammern dahinter stehen. Bei unserem Beispiel hat das etwa folgende Form:

```
Datum::Datum(unsigned int _t,  
            unsigned int _m, unsigned int _j)  
: t(_t), m(_m), j(_j)  
{  
}
```

Dabei sind die Größen `t`, `m` und `j` die Attribute (deklariert im `private`-Abschnitt der Klasse `Datum`), die wir durch diesen Konstruktor mit den übergebenen Werten belegen. Eine Prüfung, ob die angegebenen Werte in einem gültigen Bereich liegen, findet dabei allerdings nicht statt.

Wie Sie sehen, kann in diesem Fall der Methodenkörper sogar ganz leer sein. Beim Aufruf des Konstruktors wird zuerst die Initialisierungsliste abgearbeitet, und zwar in der Reihenfolge, wie die Attribute in der Klasse deklariert sind — und nicht, wie sie in der Liste stehen! Generell sollten Sie daher die Datenelemente in Ihren Initialisierungslisten stets in derselben Reihenfolge aufführen wie in der Klasse. (Besonders kritisch wird dies aber erst, wenn ein Datenelement auf ein anderes angewiesen ist.)

## Hintergrund

Eine Klasse darf nicht nur Variablen und Funktionen enthalten, von denen bisher immer die Rede war, sondern auch Konstanten und Referenzen. Bei diesen stellt sich das Problem der Initialisierung noch wesentlich drängender. Denn normalerweise müssen Konstanten und Referenzen gleich bei ihrer Deklaration initialisiert werden, etwa (bei bekannter Klasse `Kreis`):

```
const unsigned int MAX_SIZE = 1000;  
// ...  
Kreis aktuellerKreis;  
Kreis& k = aktuellerKreis;
```

Sind sie aber Bestandteil einer Klasse, so ist es nicht erlaubt, sie gleich bei ihrer Deklaration zu initialisieren, denn Zuweisungen innerhalb der Klassendeklaration sind in C++ nicht möglich. Folgender Code ist also nicht gültig:

```
class Kreis;  
  
class Kreisliste  
{  
private:
```

*Initialisierung von  
Konstanten und Referenzen*

```

    const unsigned int MAX_SIZE = 1000; // nicht gültig!
    Kreis aktuellerKreis;
    Kreis& k = aktuellerKreis; // nicht gültig!
// ...
};

```

Diese Klasse müssen Sie zunächst ohne die Initialisierung der Konstanten und Referenzen, die sie enthält, deklarieren. (Dann brauchen Sie eigentlich das Attribut `aktuellerKreis` gar nicht mehr, oder?)

```

class Kreisliste
{
private:
    const unsigned int MAX_SIZE;
    Kreis& k;
    // ...
public:
    Kreisliste(Kreis& _k);
    // ...
};

```

Hier kommt jetzt die Initialisierungsliste ins Spiel. Denn Konstanten und Referenzen *müssen* durch eine solche Liste initialisiert werden — ein anderer Weg ist nicht erlaubt! Beim Konstruktor schreiben Sie dann beispielsweise:

```

Kreisliste::Kreisliste(Kreis& _k)
    : MAX_SIZE(1000), k(_k) // eventuell weitere
{
    // ...
}

```

Eine solche Initialisierung müssen Sie übrigens bei jedem Konstruktor dieser Klasse angeben.

Auf diese Weise bekommen Sie also die Möglichkeit, zusätzliche Arten von Elementen in Ihre Klassen aufzunehmen.

### 2.6.5 Kopierkonstruktor

Ein Kopierkonstruktor hat die Aufgabe, ein Objekt mit einem anderen derselben Klasse zu initialisieren. Dazu hat er als Parameter eine konstante Referenz auf dieses Objekt. Für unsere Datumsklasse könnte das beispielsweise lauten:

```

Datum::Datum(const Datum& _datum)
: t(_datum.t), m(_datum.m), j(_datum.j)
{
    cout << "Hier ist der Kopierkonstruktor!"
          << endl;
}

```

Sie können diesen Konstruktor aufrufen wie andere auch, etwa:

```
Datum d1;
Datum d2(d1);
```



**Abbildung 2.13**  
Ein Kopierkonstruktor erzeugt ein Objekt nach der Vorlage eines anderen.

Es gibt aber noch eine — anfangs eventuell verwirrende — Möglichkeit, den Kopierkonstruktor aufzurufen, nämlich in Form einer Zuweisung:

*Aufruf in Form einer Zuweisung*

```
Datum d1;
Datum d2 = d1;
```

Und obwohl hier das Gleichheitszeichen an eine Zuweisung denken lässt, *ist es keine Zuweisung, sondern eine Initialisierung!* Wenn Sie aber bedenken, dass auch Variablen von Standardtypen, die Sie gleich bei der Deklaration initialisieren, mit dem Gleichheitszeichen ihre Werte erhalten, ist diese Syntax nur konsequent. Trotzdem ist die hier vorgenommene strenge Unterscheidung zwischen Zuweisung und Initialisierung gerade für den Anfänger oft schwierig. Daher nochmals ein Beispiel:

```
Raumfahrzeug ufo;
Raumfahrzeug transporter;

// Zuweisung, kein Kopierkonstruktor:
transporter = ufo;

// Initialisierung mit Kopierkonstruktor:
Raumfahrzeug jaeger = ufo;
```

Sie können sich also merken: Der Kopierkonstruktor tritt nur dann in Aktion, wenn ein *neues Objekt* erzeugt wird, aber *nicht*, wenn ein bereits bestehendes einen neuen Wert erhält.

## Hintergrund

Braucht eigentlich jede Klasse einen Kopierkonstruktor? Diese Frage ist durchaus berechtigt, ist doch die Initialisierung eines Objekts mit einem anderen derselben Klasse eine gängige Anweisung. Und da die Situation so oft vorkommt, erzeugt der Compiler selbst einen Kopierkonstruktor, wenn der Autor der Klasse keinen

*Sinn des Kopierkonstruktors*

bereitstellt. In diesem wird das neue Objekt erzeugt, indem alle Datenelemente Bit für Bit kopiert werden. Das klappt bei Standarddatentypen immer, womit Sie sich notieren können: *Bei Klassen, deren Attribute nur Standarddatentypen haben, ist ein selbst definierter Kopierkonstruktor nicht nötig.* Ist ein Attribut wiederum selbst ein Objekt, so wird beim Kopieren dessen Kopierkonstruktor aufgerufen und so weiter.

Ein Klasse braucht jedoch mindestens immer dann einen Kopierkonstruktor, wenn sie dynamisch angelegten Speicherplatz verwaltet. Denn der automatisch erzeugte kopiert im Allgemeinen nur den Anfangspunkt dieses Speichers, so dass das kopierte Objekt einen Verweis auf denselben Speicherbereich erhält wie das ursprüngliche. Und das will man beim Kopieren ja vermeiden! Daher gilt auch die Faustregel: Immer wenn für eine Klasse ein Kopierkonstruktor erforderlich ist, braucht sie auch einen Zuweisungsoperator. Mehr Details dazu erkläre ich Ihnen aber später, wenn wir den Umgang mit dynamisch angelegtem Speicher genauer unter die Lupe nehmen (ab Seite 206).

### Objekte als Rückgabewerte

Manchmal ist es sinnvoll, wenn eine Funktion oder eine Methode ein Objekt zurückliefert. Dies kann einmal in Form einer Referenz geschehen, etwa:

```
Datum& Log::getLogDate();
```

Das Problem dabei ist jedoch, dass sich diese Referenz auf ein Objekt beziehen muss, das auch nach Ende der Funktion noch gültig ist. Wenn also eine Methode eine Referenz auf ein Attribut zurückliefert, ist das völlig in Ordnung (solange dadurch nicht der Zugriffsschutz zu sehr unterwandert wird ...). Wenn aber die Funktion ein Objekt zurückliefern will, das sie lokal erst erzeugt hat, ist die Referenz völlig untauglich.

Dabei ist es doch auch möglich, ein ganzes Objekt als Ergebnis einer Funktion oder Methode zurückzugeben. Sie müssen lediglich seine Klasse als Rückgabebetyp der Funktion angeben. Sehen wir uns folgendes Beispiel an, um zu erkennen, was dabei so alles vor sich geht:

```
1: Datum Log::getLogDate()
2: {
3:     Datum d(29,2,2000); // zu Testzwecken fix
4:     return d;         // Objekt wird zurückgegeben
5: }
6:
7: int main()
8: {
9:     Log logObject;
10:    Datum date = logObject.getLogDate();
11:    // ...
12: }
```

In Zeile 1 finden Sie den Kopf der Methode `getLogDate()` (die für unsere Zwecke immer dasselbe Datum liefert). Dort ist — wie erwartet — als Typ des Rückgabewerts die Klasse `Datum` vermerkt. Ein Objekt davon wird in Zeile 3 angelegt; dabei wird natürlich der entsprechende Konstruktor aufgerufen. In der vierten Zeile startet der Rücksprung. Diesen müssen wir zusammen mit dem Aufruf in Zeile 10 betrachten, um die Vorgänge zu verstehen. Dort wird ein neues Objekt der Klasse `Datum` angelegt und gleichzeitig initialisiert — ein Fall für den Kopierkonstruktor. Dieser erhält als Argument gerade das lokale Objekt `d` aus `getLogDate()`. Anschließend, am Ende der Methode in Zeile 5, wird `d` erst gelöscht. Und am Ende von `main()`, in Zeile 12, wird natürlich auch das Objekt `date` wieder entfernt.

Das heißt also: Vor dem endgültigen Ende und dem Aufräumen der Funktion wird — im Kontext des Aufrufers! — erst der Kopierkonstruktor (oder ein Zuweisungsoperator) aufgerufen, der das Objekt, das die Funktion zurückgibt, an seinen Bestimmungsort bringt.

### 2.6.6 Typumwandlungskonstruktor

Ein Typumwandlungskonstruktor ist eine spezielle Form des allgemeinen Konstruktors. Er dient dazu, andere Datentypen in die jeweilige Klasse umzuwandeln. Damit ist dann eine Regel für die implizite Typkonvertierung erklärt und Sie können an allen Stellen, wo eigentlich ein Objekt der Klasse erwartet würde, einen Wert dieses Typs angeben.

Sehen wir uns dazu gleich ein Beispiel an. Bei unserer Datumsklasse ist es sehr viel einfacher, eine Zeichenkette zur Initialisierung anzugeben als drei `int`-Zahlen. Daher fügen wir folgenden Konstruktor hinzu:

```
class Datum
{
public:
    Datum(const string& _datumstring);
    // ... Rest wie auf Seite 95 f.
};
```

Dann können Sie die Typumwandlung in die gewünschten Bahnen lenken und einfach schreiben:

```
int main()
{
    Datum d1("29.02.2000");
```

Wenn Ihnen auf Anhieb klar ist, dass diese Syntax funktioniert, sehen wir uns gleich folgende Anweisung an:

```
Datum d2;
// ...
d2 = "03.03.2001";
```

Das ist eigentlich eine Kurzschreibweise für zwei getrennte Schritte. Zunächst wird ein temporäres Objekt unter Zuhilfenahme des Typumwandlungskonstruktors erzeugt und anschließend wird dieses dem bestehenden, nämlich `d2`, zugewiesen.

Noch deutlicher wird das an folgendem Beispiel: Nehmen Sie an, Sie haben eine Funktion, um eine Log-Nachricht auszugeben. Diese erhält neben der Nachricht selbst noch das Datum als Argument:

```
log_message(const Datum& _date,
           const string& _message);
```

Dann dürfen Sie dank des Typumwandlungskonstruktors schreiben:

```
int main()
{
    log_message("21.02.2001", "Jetzt geht's los!");
    // ...
}
```

Denn intern wird dieser Aufruf vom Compiler in folgenden Block umgewandelt:

```
int main()
{
    {
        Datum temp("21.02.2001");
        log_message(temp, "Jetzt geht's los!");
    }
    // ...
}
```

Auf diese Weise können Sie leicht eine automatische Typumwandlung bei der Parameterübergabe erlauben und damit die Flexibilität Ihrer Klasse weiter erhöhen. Es gibt übrigens auch die umgekehrte Variante, nämlich ein Objekt in einen anderen Typ umzuwandeln; mehr dazu erfahren Sie im Abschnitt »Typumwandlungsoperator« ab Seite 318.

## Hintergrund

### *Explizite Konvertierung*

So praktisch diese automatische Umwandlung auch sein mag — manchmal möchte man gerade diese vermeiden. Denn dabei kann es passieren, dass der Compiler temporäre Objekte erzeugt, an die der Programmierer gar nicht gedacht hat. Da Erzeugung und Vernichtung solcher Objekte aber auch Zeit und Speicherplatz kosten, sollte man versuchen, die vollständige Kontrolle zu behalten und unerwünschte Automatismen gar nicht erst zuzulassen. Außerdem könnten Typverletzungen durch ein Zuviel an automatischer Konvertierung unentdeckt bleiben.

Aus diesem Grund wurde im ANSI/ISO-Standard von C++ das Schlüsselwort `explicit` eingeführt. Sie schreiben es in die Klassendeklaration vor Konstruktoren, die nur ein Argument haben (oder bei denen die weiteren Argumente wegen

der Vorgabewerte verzichtbar sind). Beispielsweise könnten wir für unser Datumsbeispiel eine eigene Klasse `Jahr` einführen. Diese hätte neben einer besseren Möglichkeit zur Typprüfung auch den Vorteil, dass wir dabei auf eine fehlende Jahrhundertangabe reagieren könnten.

```
class Jahr
{
public:
    explicit Jahr(unsigned int _j);
    unsigned int getJahr() const;
private:
    unsigned int j;
};

Jahr::Jahr(unsigned int _j)
{
    if (_j < 10)
        j = _j + 2000;
    else
        if (_j < 100)
            j = _j + 1900;
        else
            j = _j;
}
```

Damit sind von folgenden Anweisungen einige nicht erlaubt:

```
SpeicherJahr(Jahr _j); // Beispielfunktion
```

```
Jahr f()
{
    Jahr heuer(2000); // erlaubt

    SpeicherJahr(2000); // nicht erlaubt!!
    SpeicherJahr(Jahr(2000)); // explizit: erlaubt

    return 99; // nicht erlaubt!!
}
```

In der Klasse `Datum` speichern wir die Jahreszahl statt als Ganzzahl nun als Objekt vom Typ `Jahr`. Dabei müssen wir natürlich auch den Konstruktor anpassen.

```
class Datum
{
public:
    Datum(unsigned int _t,
          unsigned int _m, Jahr _j);
    // ...
};
```

Auf diese Weise können wir beispielsweise vermeiden, dass die Schnittstelle falsch benutzt wird. Gerade beim Datum gibt es nämlich sehr unterschiedliche Schreibweisen. Ohne das Objekt `Jahr` wäre folgende Erzeugung auch richtig gewesen:

```
Datum d1(2000,11,20);
```

Dieser Fehler kann nicht vom Compiler bemerkt werden, sondern erst zur Laufzeit von der Bereichsüberprüfung der Methode `setze()`. Aber selbst der bislang korrekte Aufruf

```
Datum d2(20,11,2000);
```

führt nun zu einem Fehler. Denn dieser würde eine implizite Konvertierung von einer `int`-Zahl in ein `Jahr`-Objekt voraussetzen, die wir ausdrücklich untersagt haben. Die Erzeugung muss nun lauten:

```
Datum d2(20,11,Jahr(2000));
```

Ähnliches kann man auch noch mit dem Monat machen, um die Schnittstellen ganz abzusichern. Für unser kleines Beispiel sind solche »Wrapper« sicherlich übertrieben; an kritischen Stellen Ihrer Programme können Sie auf diese Weise jedoch viel an Sicherheit und damit an Qualität gewinnen.

### 2.6.7 Destruktoren

Genauso wie es Methoden gibt, die bei der Erzeugung eines Objekts aufgerufen werden, gibt es auch welche, die bei seiner Vernichtung in Aktion treten.

**Abbildung 2.14**  
Destruktoren räumen  
nicht mehr benötigte  
Objekte auf.



Ein Destruktor übernimmt dann die Aufräumarbeit, wenn das Objekt nicht mehr benötigt wird. Ebenso wie bei den Konstruktoren gilt: Gibt der Autor der Klasse keinen Destruktor vor, erzeugt der Compiler automatisch einen. Sie werden in Ihren Klassen Destruktoren hauptsächlich dann verwenden, wenn Sie mit dynamisch verwaltetem Speicher arbeiten und das Objekt den Speicher, den es für sich in Anspruch genommen hat, am Ende seiner Lebenszeit wieder freigeben muss. Mehr dazu werde ich Ihnen daher erklären, wenn wir über dynamische Speicherverwaltung reden (ab Seite 206).

Ein weiteres Aufgabenfeld für Destruktoren sind offene Datei- oder Datenbankverbindungen, die beim Löschen des Objekts geschlossen werden müssen.

### Syntax des Destruktors

Der Destruktor hat denselben Namen wie die Klasse, allerdings mit einer vorangestellten Tilde ~, zum Beispiel:

```
class Datum
{
public:
    Datum(); // Standardkonstruktor
    ~Datum(); // Destruktor
    // ...
};

inline Datum::~Datum()
{
    cout << "Hier ist der Destruktor!" << endl;
}
```

Jede Klasse kann höchstens einen Destruktor haben. Dieser hat keinerlei Rückgabewert (wie der Konstruktor) und keine Argumente. Da er meist sehr wenig tut, bietet es sich an, ihn als `inline` zu deklarieren.

### Gültigkeitsbereiche von Objekten

Für Objekte, die innerhalb eines Blocks oder einer Funktion angelegt werden, endet ihre Gültigkeit (ihre Lebensdauer) mit der schließenden Klammer »}«. Zu diesem Zeitpunkt wird auch der Destruktor aufgerufen. Wenn Sie ein Objekt außerhalb aller Funktionen deklarieren, nennt man es *global* (dasselbe gilt auch für Variablen). Globale Objekte sind in jeder Funktion sichtbar und dort wie lokale Objekte verwendbar. Für globale Objekte wird der Konstruktor vor der ersten Anweisung in `main()` und der Destruktor nach der Freigabe aller in `main()` instanziierten Objekte aufgerufen.

```
#include <iostream>
#include "Datum.hh"

Datum heutigesDatum__; // globales Objekt

int main()
{
    cout << "Heute ist der ";
    heutigesDatum__.ausgeben();
    return 0;
}
```

Globale Objekte sind fast immer Notlösungen. War man in C noch ziemlich großzügig, was den Umfang an globalen Variablen anging, so gibt es in

C++ fast immer bessere Lösungen. Wenn es aber einmal doch eine globale Variable sein muss, sollten Sie sie entsprechend kennzeichnen, etwa mit dem Suffix `_g` oder zwei angehängten Unterstrichen wie in `heutigesDatum__`.

### 2.6.8 Beispiel: Benutzerinformationen

In diesem Beispiel wollen wir eine Klasse für Benutzer Ihres Linux-Rechners erstellen. Darin sollen der Login-Name, der echte Name und die Gruppen-Zuordnung gespeichert werden. Dazu müssen wir uns zunächst ansehen, wie Benutzer überhaupt durch Linux verwaltet werden.

#### Die Benutzerverwaltung unter Linux

Auf einem Einzelplatzsystem sind die Informationen über die eingetragenen Benutzer in der Datei `/etc/passwd` abgelegt. Dazu gehören der Benutzername, eine eindeutige ID, IDs für die Zugehörigkeit zu Benutzergruppen, das Passwort, der Name des Home-Verzeichnisses, die Shell, die beim Login gestartet wird, sowie eine textuelle Beschreibung, im Allgemeinen der vollständige Name des Benutzers. Bei einem vernetzten Rechner können diese Informationen auch zentral abgelegt sein und über den Network Information Service (*NIS*) zugänglich gemacht werden (früher als gelbe Seiten — *yellow pages* — bekannt). Diese Daten (außer dem Passwort) werden übrigens auch vom Kommando *finger* ausgegeben.

Das System stellt für den Zugriff auf diese Daten aus C- und C++-Programmen verschiedene Funktionen zur Verfügung:

- ❑ `getuid()` liefert die User-ID des aktuellen Benutzers, der das Programm startet.
- ❑ `getpwuid()` erwartet eine Benutzer-ID und gibt dazu alle Daten aus `/etc/passwd` zurück. Dies geschieht in Form einer Struktur (gekennzeichnet durch das Schlüsselwort `struct`); das ist auch nur eine Klasse, bei der alle nicht anders gekennzeichneten Elemente `public` sind. Die Struktur `passwd` ist in `usr/include/pwd.h` definiert.
- ❑ `getpwnam()` liefert dasselbe Ergebnis wie `getpwuid()`, hat jedoch als Argument den Login-Namen des Benutzers.

Der Umgang mit diesen Funktionen erfordert etwas Zeiger-Syntax, auf die ich erst ab Seite 190 zu sprechen kommen werde. Sie können also entweder dort nachschlagen oder mir zunächst einmal glauben, dass die Anweisungen korrekt sind.

## Die Klassendeklaration

Als Daten aus der Passwortdatei interessieren uns hier nur der Login-Name, der echte Name sowie die Gruppen-ID. Zur Speicherung der Texte verwenden wir die Klasse `string`, von der Sie an dieser Stelle eigentlich kaum etwas wissen müssen; die Zuweisung mit `=` ist sehr offensichtlich und die Ausgabe mit `cout` auch unproblematisch. (Wenn Sie doch mehr wissen wollen: Seite 282.)

*Attribute*

Im öffentlichen Teil statten wir die Klasse mit einem Standardkonstruktor aus, der die Daten lediglich initialisiert, und zwei allgemeinen Konstruktoren, wobei der eine eine Benutzer-ID und der andere einen Benutzernamen erwartet. Ferner wollen wir auch eine nachträgliche Initialisierung mit einem Benutzernamen zulassen sowie das Objekt auf den aktuellen Benutzer setzen können und auch die enthaltenen Daten auf den Bildschirm ausgeben.

*Methoden*

Unter diesen Methoden sind zwei fast identisch, was vielleicht auf den ersten Blick nicht so offensichtlich ist. Ich meine den allgemeinen Konstruktor, der die Attribute mit den Daten eines Benutzers belegt, von dem die ID bekannt ist, und das Setzen auf den aktuellen Benutzer. Denn von diesem erhält man nämlich auch die ID, wenn man die Funktion `getuid()` ruft. Somit lohnt es sich, die Funktionalität in eine private Methode zu stecken und diese aus den beiden öffentlichen aufzurufen.

Die Initialisierungsmethoden haben als Rückgabewert `true` oder `false`, je nachdem, ob sie fehlerfrei arbeiteten oder nicht. Somit erhalten wir folgende Deklaration:

```
class Benutzer
{
private:
    string login;
    string echterName;
    long   gruppenId;
    bool   init(uid_t _benutzerId);

public:
    Benutzer() :
        gruppenId(0) {}
    Benutzer(const string& _benutzerName)
        { init(_benutzerName); }
    Benutzer(uid_t _benutzerId)
        { init(_benutzerId); }
    bool init(const string& _benutzerName);
    bool setzeAufAktuellen();
    void ausgeben() const;
};
```

Im Standardkonstruktor müssen wir nur die Gruppen-ID mit einem Wert belegen, da die anderen beiden Attribute selbst Objekte sind und deren Standardkonstruktor aufgerufen wird.

### Benutzerinformationen aus der ID

Die private `init()`-Methode arbeitet folgendermaßen:

```
bool Benutzer::init(uid_t _benutzerId)
{
    // Ausgabevariable deklarieren
    struct passwd* benutzer_info = 0;

    // Benutzer-Info holen
    benutzer_info = getpwuid(_benutzerId);

    // Falls ungleich: Benutzer existiert
    if (benutzer_info)
    {
        // Daten kopieren und ausgeben
        login = benutzer_info->pw_name;
        echterName = benutzer_info->pw_gecos;
        gruppenId = benutzer_info->pw_gid;
        cout << "Benutzer " << login
              << " heißt " << echterName << endl;
    }
    else
    {
        // Fehler melden
        cerr << "Benutzer mit Id " << _benutzerId
              << " nicht gefunden!" << endl;
        login = "";
        echterName = "";
        gruppenId = 0;
        return false;
    }

    return true;
}
```

Die Syntax mag Ihnen etwas schwer durchschaubar vorkommen, davon aber abgesehen ist der Aufbau dieser Funktion recht einfach. Die Informationen werden abgefragt und anschließend ausgewertet. Existiert der Benutzer, werden die Daten kopiert und teilweise ausgegeben; existiert er nicht, dann erscheint eine Fehlermeldung.

Der Aufbau der anderen `init()`-Methode ist fast identisch. Nur wird dort statt `getpwuid()` die Funktion `getpwnam()` aufgerufen. Ich lasse das Listing also hier weg.

*Weitere Methoden*

Die beiden anderen Methoden der Klasse sind ziemlich elementar. Dabei ist `setzeAufAktuellen()` so klein, dass wir sie gleich als `inline` deklarieren können (siehe auch Seite 88).

```
inline bool Benutzer::setzeAufAktuellen()
{
    return init(getuid());
}
```

Die Ausgabemethode setzen wir als konstant, da sie keine Datenelemente des Objekts verändert, sondern nur ausgibt. (Konstante Methoden hatten wir auf Seite 84 besprochen.)

```
void Benutzer::ausgeben() const
{
    cout << "Benutzer:      " << login << endl;
    cout << "Echter Name:  " << echterName << endl;
    cout << "Gruppe:        " << gruppenId << endl;
    cout << endl;
}
```

## Einsatz der Klasse

Anwenden können wir die Klasse auf verschiedene Arten:

*Verwendung der Klasse*

```
int main()
{
    bool ergebnis;
    // Standardkonstruktor
    Benutzer u;

    // Konstruktor mit Argument
    // UID 0 entspricht root
    Benutzer root(0);

    cout << "Leerer Benutzer: " << endl;
    u.ausgeben();

    cout << "Objekt mit Root: " << endl;
    root.ausgeben();

    ergebnis = u.init("markus");
    if (ergebnis)
    {
        cout << "Initialisierter Benutzer markus: "
```

```

        << endl;
        u.ausgeben();
    }

    ergebnis = u.setzeAufAktuellen();
    if (ergebnis)
    {
        cout << "Aktueller Benutzer: " << endl;
        u.ausgeben();
    }
    return 0;
}

```

*Bildschirmausgabe*

Die Ausgabe hängt dabei natürlich von Ihrem System ab. Wenn es dort beispielsweise keinen Benutzer markus gibt, werden Sie eine Fehlermeldung statt Benutzerinformationen erhalten. Bei mir war auf dem Bildschirm zu lesen:

```

Benutzer root heißt root (Superuser)
Leerer Benutzer:
Benutzer:
Echter Name:
Gruppe:      0

Objekt mit Root:
Benutzer:    root
Echter Name: root (Superuser)
Gruppe:      0

Benutzer markus heißt Markus Zimmermann
Initialisierter Benutzer markus:
Benutzer:    markus
Echter Name: Markus Zimmermann
Gruppe:      100

Benutzer thomas heißt Thomas Wieland
Aktueller Benutzer:
Benutzer:    thomas
Echter Name: Thomas Wieland
Gruppe:      100

```

Sie sollten an diesem Beispiel Möglichkeiten und Grenzen des Einsatzes von Konstruktoren kennen lernen. Sie können damit ein Objekt gleich bei seiner Erzeugung mit sinnvollen Werten belegen lassen, wie hier mit den Benutzerinformationen. Allerdings ist es im Konstruktor kaum möglich, Fehler abzufangen und zu behandeln. Sie sollten daher eigentlich nur solche Methoden im Konstruktor aufrufen, die garantiert nicht schief gehen können. Unser Design von oben war in diesem Sinne nicht optimal.

## 2.6.9 Zusammenfassung

Folgende Gesichtspunkte aus diesem Abschnitt sollten Sie festhalten:

- ❑ Ein *Konstruktor* dient dazu, ein Objekt in einen definierten Anfangszustand zu versetzen, das heißt Speicherplatz für die Attribute bereitzustellen und gegebenenfalls die Attribute mit sinnvollen Anfangswerten zu initialisieren. Aufrufe von Konstruktoren werden automatisch in das Programm eingefügt.
- ❑ Konstruktoren sind spezielle Methoden, die denselben Namen wie die Klasse haben. Sie verfügen über keinerlei Rückgabewerte (auch nicht `void`).
- ❑ Man unterscheidet zwischen *Standardkonstruktor*, der bei jeder Erzeugung eines Objekts der Klasse verwendet wird, wenn kein anderer angegeben ist, *allgemeinem Konstruktor*, der beliebige Argumente haben darf und überladen werden kann, *Kopierkonstruktor*, mit dem ein Objekt der Klasse durch ein anderes initialisiert wird, und *Typumwandlungskonstruktor*, der zur Konvertierung anderer Datentypen in die Klasse dient.
- ❑ Der *Destruktor* übernimmt die Aufräumarbeit für ein nicht mehr gültiges Objekt. Sein Name besteht aus dem Namen der Klasse mit vorangestellter Tilde »~«. Sein häufigster Zweck ist die Freigabe von dynamisch angelegtem Speicher.

## 2.6.10 Übungsaufgaben

1. Beantworten Sie folgende Fragen:
  - ❑ In welchen Situationen braucht man einen Kopierkonstruktor?
  - ❑ Welche Typen wandelt der Typumwandlungskonstruktor um?
  - ❑ Wie behandeln Sie Fehler, die in Konstruktoren auftreten?
2. Der Kopierkonstruktor für eine Klasse `x` wird entweder in der Form `x::x(x& a)` oder (im Allgemeinen besser) als `x::x(const x& a)` deklariert. Warum nimmt man statt der Referenz auf `x` nicht einfach das Objekt `x` und deklariert `x::x(x a)`?
3. Erweitern Sie die Klasse `Rational` aus Abschnitt 2.5.10 (Seite 92) um einen Kopier- und einen Typumwandlungskonstruktor, der Ganzzahlen vom Typ `long` akzeptiert. Welche zusätzlichen Verwendungsmöglichkeiten ergeben sich dadurch?

4. Schreiben Sie zwei Klassen A und B, die in ihren Konstruktoren und Destruktoren jeweils einen Text ausgeben (etwa »Hier ist Konstruktor von A«). Erweitern Sie diese Klassen zu einem Programm, in dem
- A ein Attribut vom Typ B hat,
  - es ein globales Objekt vom Typ A gibt,
  - es eine Instanz von A in `main()` gibt,
  - es eine Instanz von A in einem untergeordneten Block der Funktion `main()` gibt.

Beobachten Sie, wann welche Konstruktoren und Destruktoren aufgerufen werden und erklären Sie das Verhalten dieses Programms.

## 2.7 Vererbung und Polymorphismus

Eines der bedeutsamsten Konzepte der objektorientierten Programmierung ist die *Vererbung*. Sie hilft Ihnen bei der logischen Gliederung Ihrer Klassen und Objekte und erleichtert die Wiederverwendung von bestehendem Code. In diesem Abschnitt werden Sie erfahren, was es damit auf sich hat und worauf Sie achten müssen.

### 2.7.1 Basisklassen und abgeleitete Klassen

Sie wissen jetzt schon, dass Sie ein größeres objektorientiertes Programm nicht einfach beginnen sollten, indem Sie die erste Zeile Code eintippen. Der erste Schritt (oder einer der ersten Schritte) ist immer die Analyse, mit welchen Objekten das Programm eigentlich umgehen soll. Bilden Sie davon eine Abstraktion, gelangen Sie zu den Klassen. Wenn Sie diesem Ansatz folgen, werden Sie bei vielen Ihrer Programme auf das Problem stoßen, dass Sie zwei Objekte vorliegen haben, die sich zwar weitgehend ähneln, aber eben doch an einigen Stellen voneinander abweichen, wie das Raumfahrzeug und der Satellit in Abbildung 2.15. (Im mittleren Drittel der Kästen stehen die Attribute, im unteren die Methoden.)

Die Lösung besteht nun darin, Gemeinsamkeiten der Klassen in einer neuen Klasse zusammenzufassen, der *Basisklasse* (oder *Oberklasse*). Diese ist dann der »Stammvater« der anderen. Ihre Nachfahren, die *abgeleiteten Klassen* (oder *Unterklassen*), besitzen alle Attribute und Methoden der Basisklasse sowie zusätzliche eigene. Eine abgeleitete Klasse erbt also die Eigenschaften und das Verhalten ihrer Oberklasse, genauso wie Eltern ihren Kindern große Nasen und Vorlieben für Süßigkeiten vererben.

Ein Beispiel dazu sehen Sie in Abbildung 2.16 auf Seite 118. Dort ist der Zusammenhang noch aus einem anderen Blickwinkel dargestellt. In der