

## Kapitel 19

# Objektorientierte Programmierung



## AUTOMOBILGESCHICHTE SCHREIBT MAN HEUTE IN C++.

TEILEN SIE MIT UNS IHRE LEIDENSCHAFT FÜR ZUKUNFTSTECHNOLOGIEN.

**BMW**  
**GROUP**  
Recruiting



Rolls-Royce  
Motor Cars Limited

Certainly not every good program is object-oriented, and not every object-oriented program is good

Bjarne Stroustrup

## Warum objektorientierte Entwicklung?

„Structured programming appears to fall apart when applications exceed 100,000 lines or so of code“

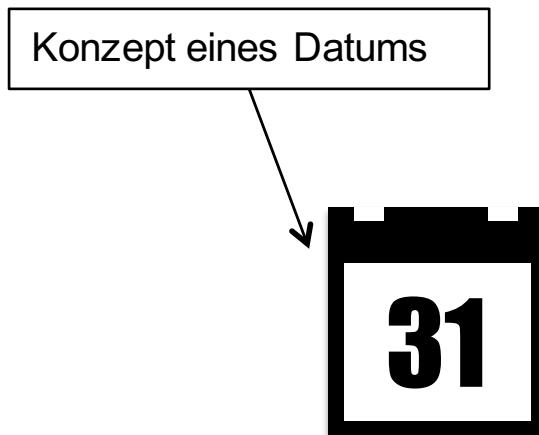
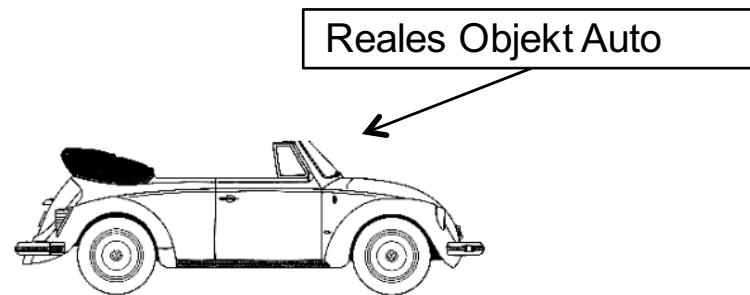
## Typischer Umfang großer Programmsysteme

Jahr	System	SLOC [Mio.]
1993	Windows NT 3.1	4-5
1996	Windows NT 4.0	11-12
2000	Windows 2000	>29
2001	Windows XP	40
2003	Windows Server 2003	50
2000	Debian 2.2	55-59
2002	Debian 3.0	104
2007	Debian 4.0	283
2005	Mac OS X 10.4	86
2003	Linux Kernel 2.6.0	5.2
2009	Linux Kernel 2.6.32	12.6
2007	SAP NetWeaver	238

Anzahl von Millionen  
Codezeilen (SLOC,  
Source Lines Of Code)

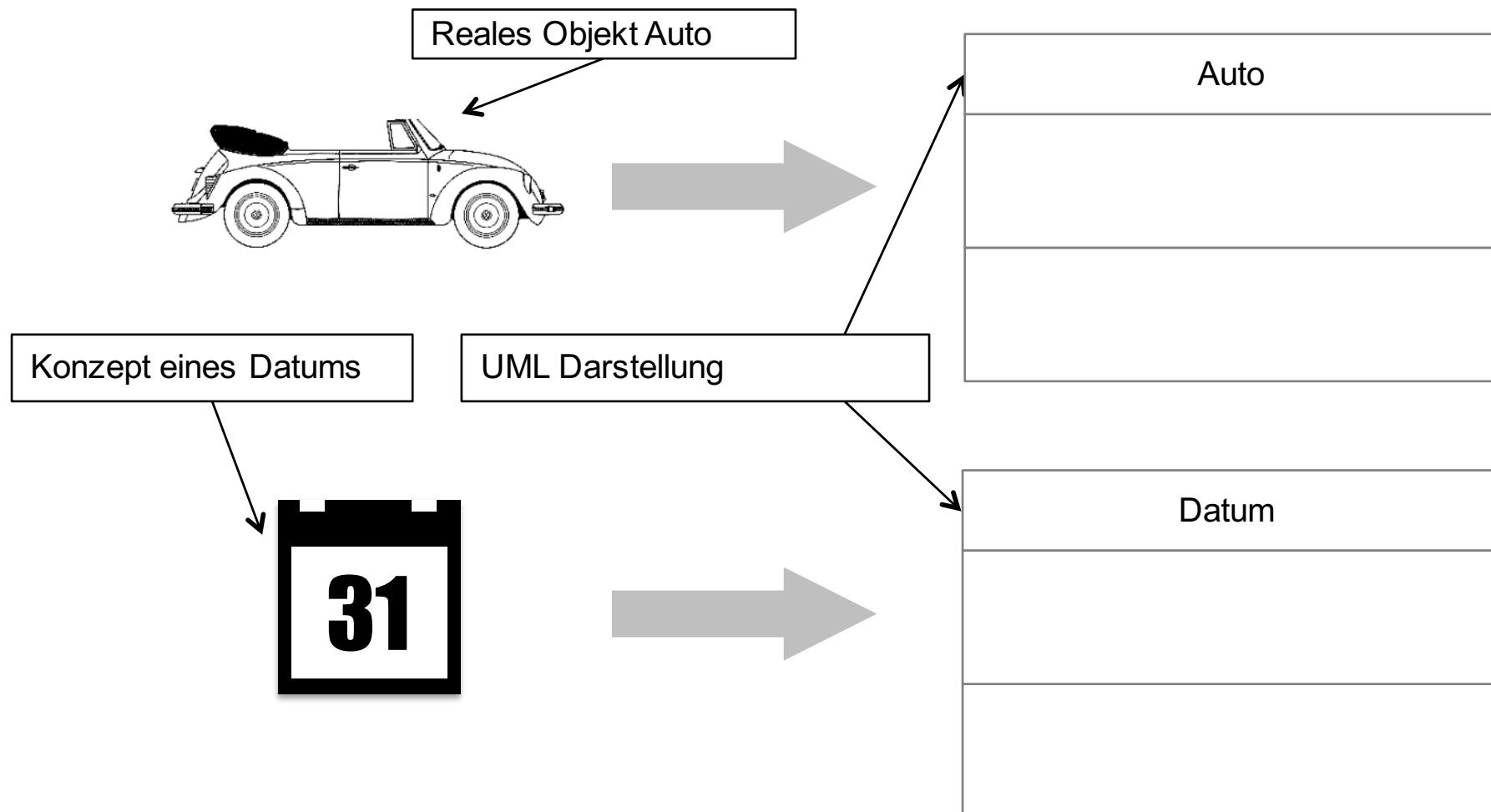
## Was sind Objekte?

Als Objekt bezeichnet man die softwaretechnische Repräsentation eines Gegenstandes oder eines Konzeptes. Der Gegenstand kann dabei real existieren oder auch nur gedacht sein. Objekte leiten sich typischerweise aus Hauptworten ab.



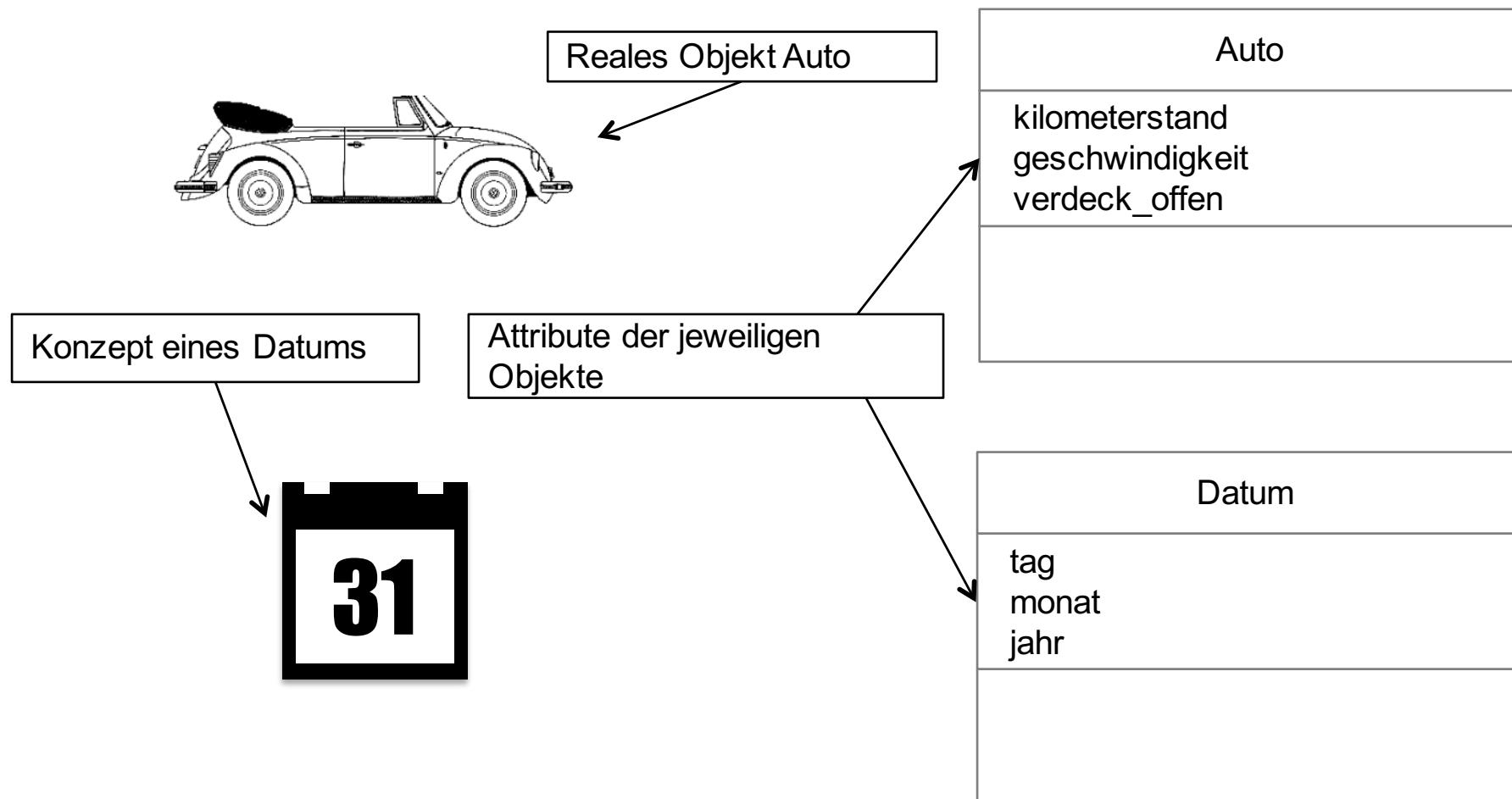
## Darstellung von Objekten

In der Erstellung und Modellierung von Softwaresystemen werden Objekte sehr häufig in der sogenannten Unified Modelling Language (UML) dargestellt. Ein Objekt wird dabei durch ein Rechteck mit drei Feldern repräsentiert. Im oberen Feld steht der Name des Objektes.



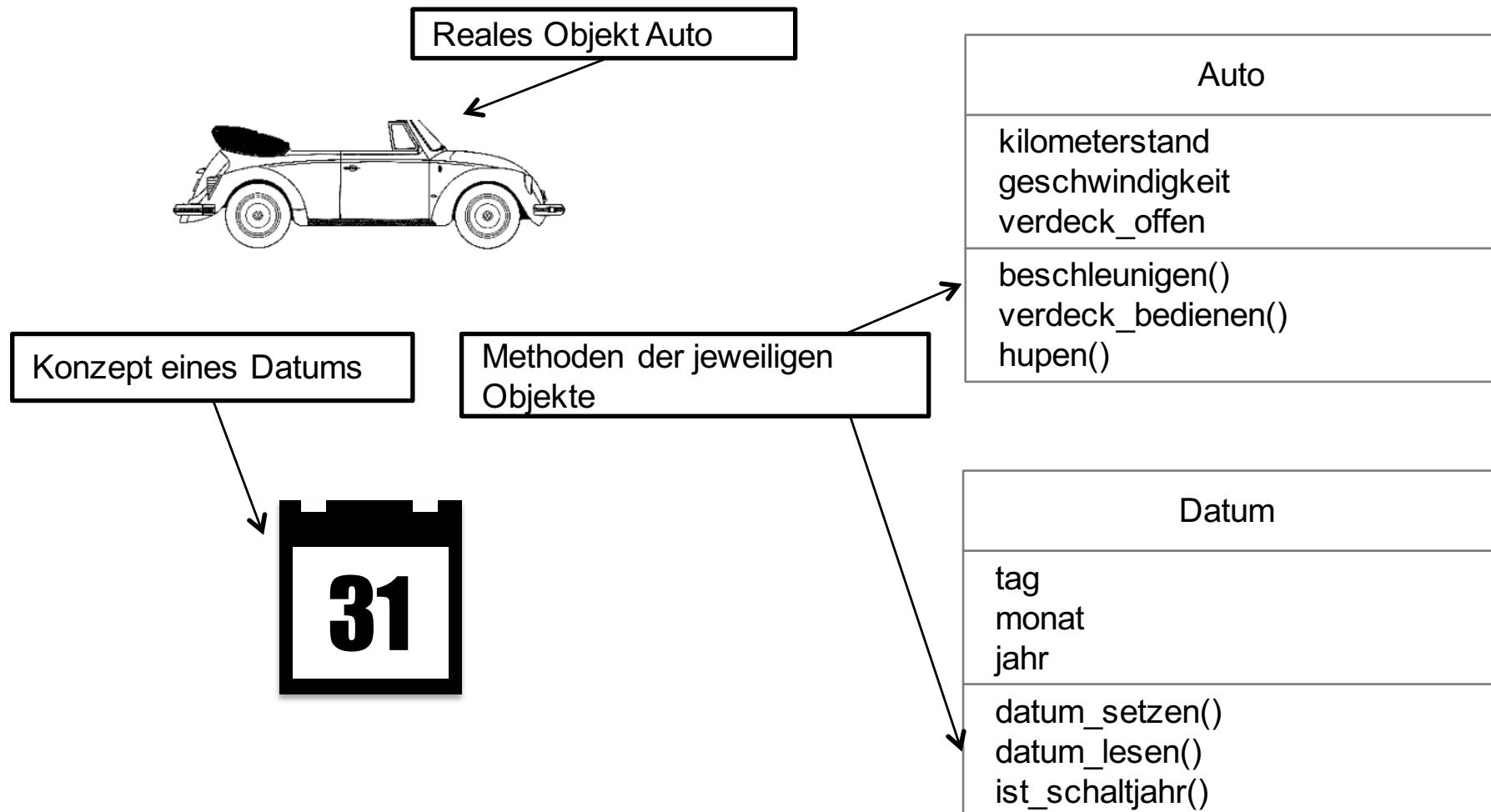
## Der statische Zustand von Objekten

Ein Objekt befindet sich zu jedem Zeitpunkt in einem genau definierten Zustand. Dieser Zustand wird durch die Attribute des Objektes beschrieben. Diese Attribute enthalten alle Daten, um den Zustand des Objektes als Modell vollständig und konsistent zu beschreiben. Die hier angegebenen Beispielen stellen eine beispielhafte und noch unvollständige Beschreibung dar und müssten weiter detailliert werden.



## Das Verhalten von Objekten

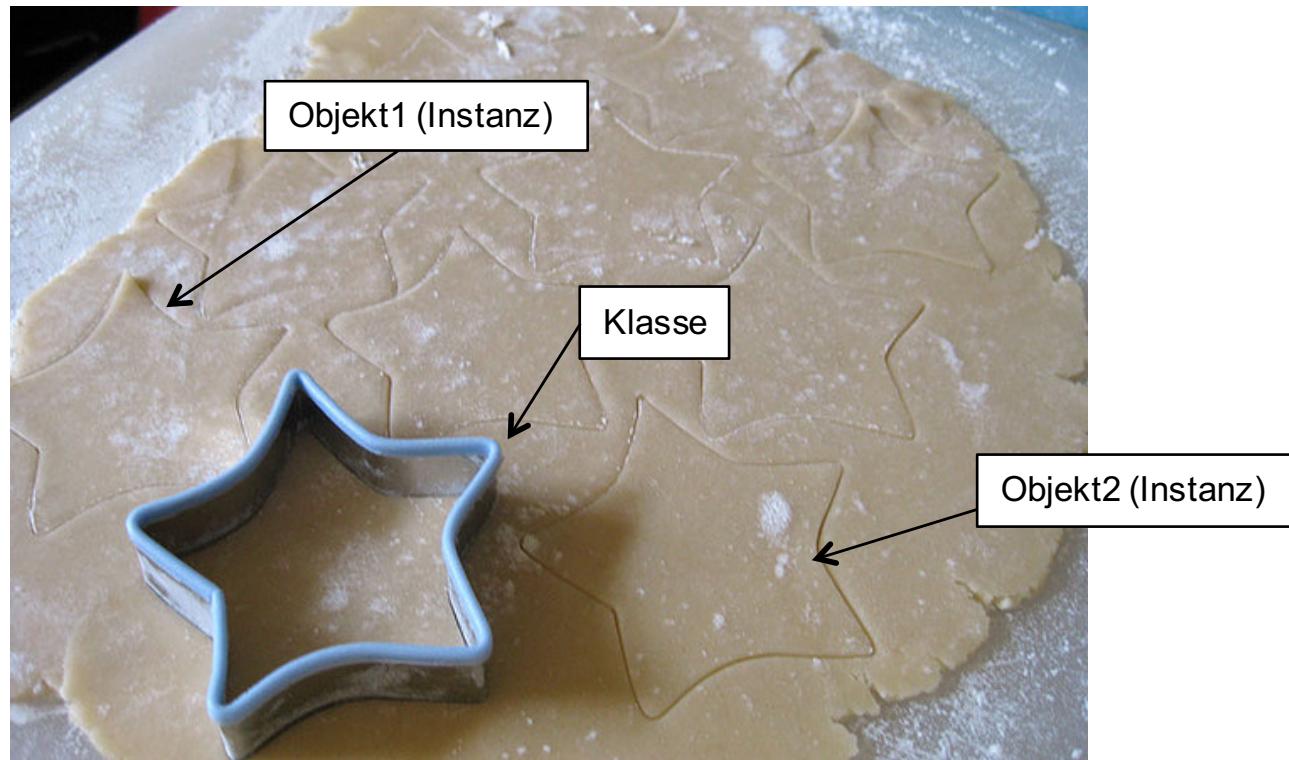
Einen Zustand konnten wir bereits mit Datenstrukturen gut abbilden. Objekte haben zu ihrem Zustand aber noch ein dynamisches Verhalten. Die Methoden eines Objektes beschreiben alle Operationen, die das Objekt ausführen kann und modellieren dessen dynamisches Verhalten. Auf die Attribute wird nur über die Methoden zugegriffen, um die Konsistenz zu sichern.



## Präzisierung der Begriffe Klasse und Objekt

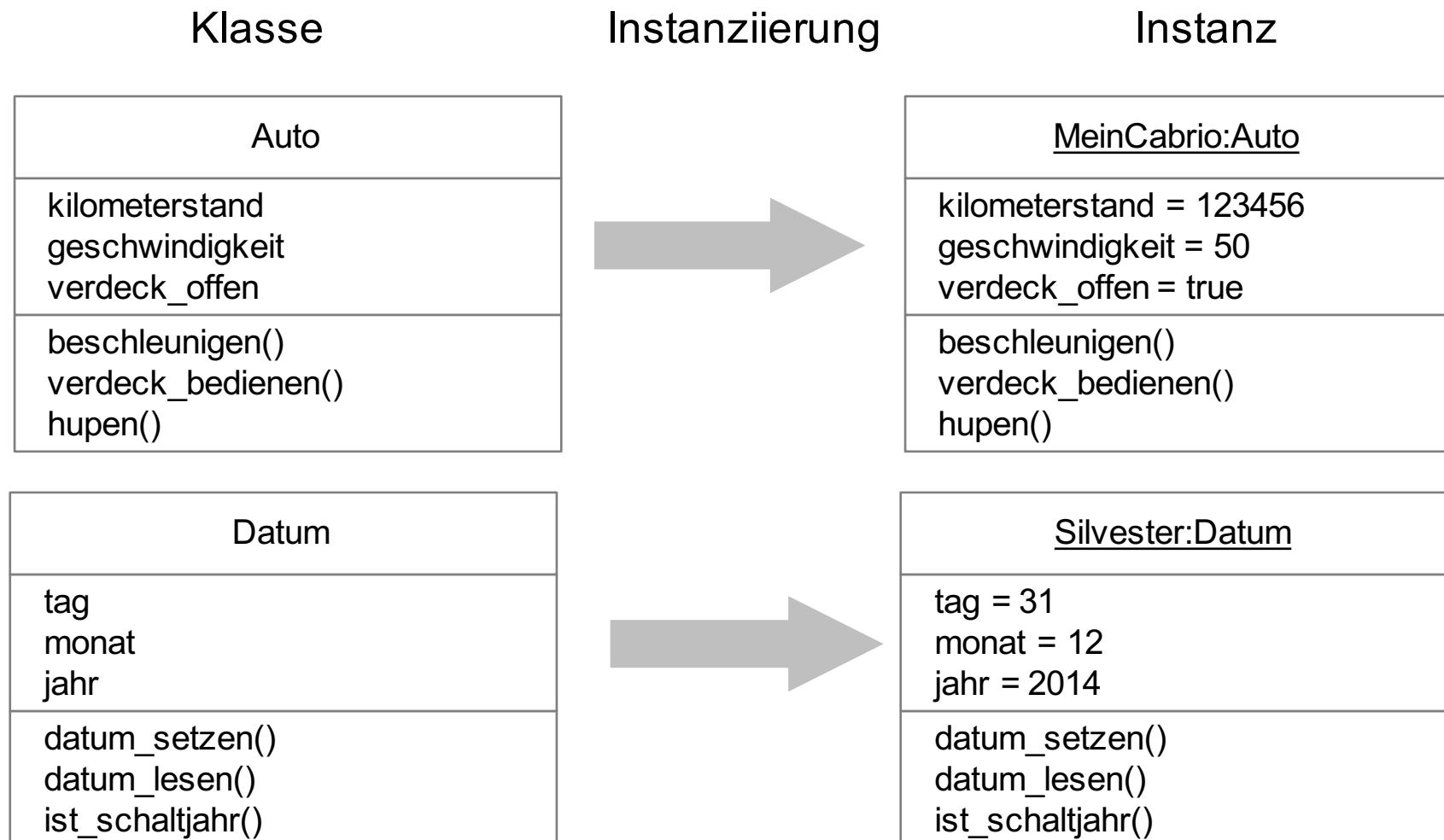
Wir haben bisher nicht unterschieden zwischen einem allgemeinen Objekt (ein Auto oder Datum mit seinen Attributen und Methoden) oder einem konkreten Objekt, wie das Datum eines ganz bestimmten Tages, wie beispielsweise Silvester im aktuellen Jahr.

Die Abstraktion dessen was ein Objekt für unsere Zwecke ausmacht, also seine Attribute und Methoden, bezeichnen wir als Klasse. Wenn wir aus einer Klasse heraus ein konkretes Objekt erstellen, dann bezeichnen wir dieses Objekt als eine Instanz der Klasse. Wir haben die Klasse **instanziert**. Wenn es im aktuellen Kontext nicht auf diese Unterscheidung ankommt, verwenden wir den Begriff Objekt weiter für Klassen und Instanzen.



## Instanziierung einer Klasse

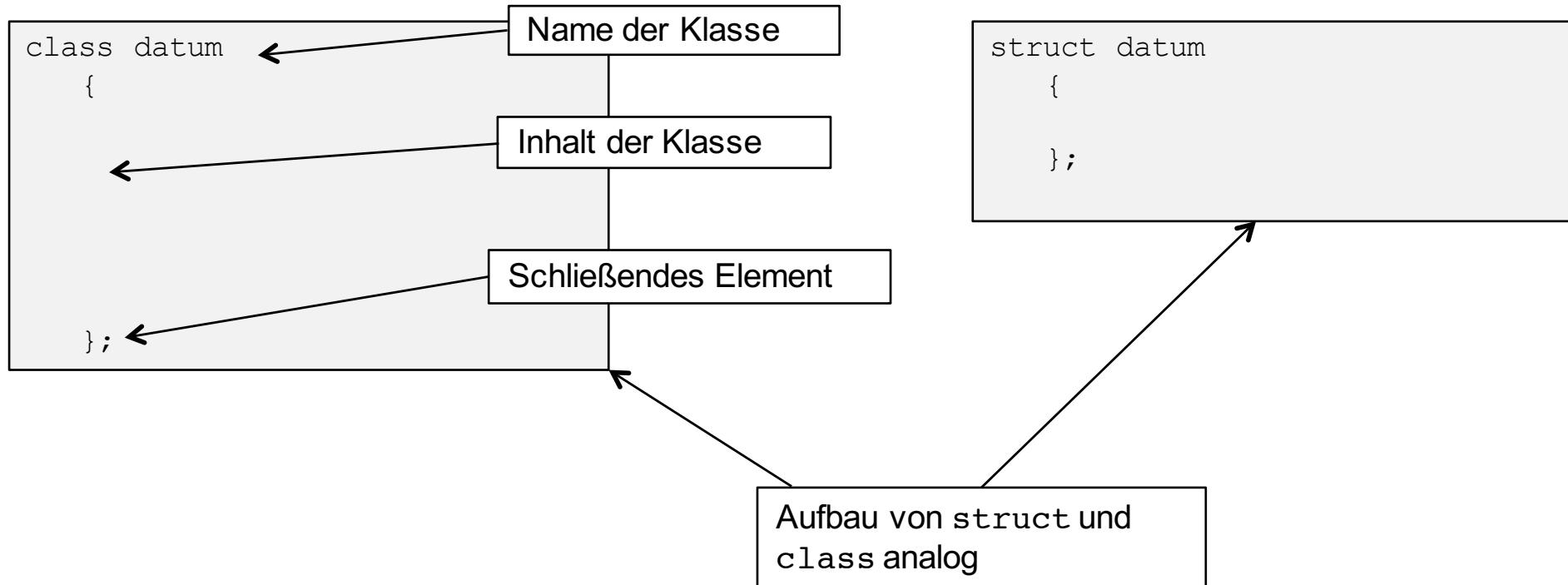
In der UML Notation wird die Instanziierung einer Klasse gekennzeichnet, indem wir den Namen der Instanz unterstrichen voranstellen und mit einem Doppelpunkt vom Klassennamen trennen. Konkrete Attributwerte einer Instanz notieren wir mit einem Gleichheitszeichen hinter dem Attributnamen



## Aufbau von Klassen in C++

Wir haben bisher allgemein über Klassen und Instanzen gesprochen und wollen uns nun damit befassen, wie Klassen in C++ umgesetzt werden.

Der Aufbau einer Klasse gleicht dem einer Datenstruktur `struct`.



## Zugriff auf Objekte

In unserem Alltag ist es normal, dass nicht jeder mit allen Objekten alles tun kann. So ist es beispielsweise jedem erlaubt, an mein Auto zu gehen und etwas unter den Scheibenwischer zu klemmen. Der Zugriff hierauf ist öffentlich (`public`). Bei den Steuergeräten eines Autos wird auch die Art des Zugriffs unterschieden. Komponenten können aus dem Steuergerät den Stand der gefahrenen Kilometer auslesen. Den Kilometerstand verändern kann nur das Steuergerät selbst. Hier ist offensichtlich der „lesende“ Zugriff (Kilometerstand auslesen) öffentlich, der „schreibende“ Zugriff (Kilometerstand ändern) ist privat (`private`).

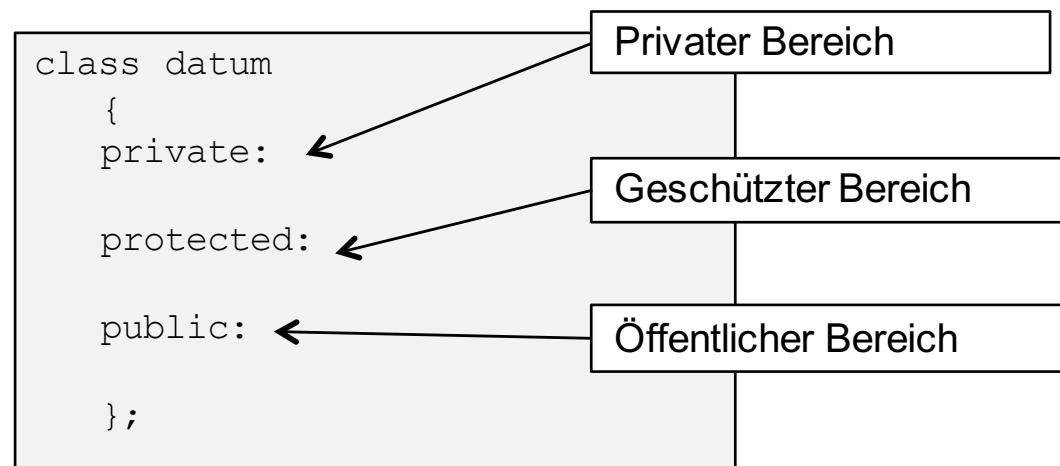
Innerhalb einer „Familie“ zusammengehöriger Steuergeräte kann auch ein Zugriff untereinander erlaubt sein, der anderen verwehrt wird. Diese Form eines geschützten (`protected`) Zugriffs finden wir auch in der objektorientierten Entwicklung bei Objekten die miteinander verwandt sind. Diese stehen in einer Beziehung, die wir als Vererbung noch kennenlernen werden.

Um bestimmte Funktionalitäten zu ermöglichen, kann auch der Zugriff durch „befreundeter“ Komponenten zugelassen sein. Diese Freunde (`friend`) können dann auf entsprechende Daten zugreifen, als wären es ihre eigenen. Die sonst geltenden Einschränkungen sind für sie aufgehoben. Beispielsweise zum Zurücksetzen des Tageskilometerzählers durch ein anderes Gerät.

Diese unterschiedlichen Formen des Zugriffes und deren Aufteilung in verschiedene Bereiche gibt es auch in der objektorientierten Programmierung. Die Benennung mit öffentlich (`public`), privat (`private`) und geschützt (`protected`) sowie die Erklärung von Freundschaften (`friend`) finden wir auch dort wieder.

## Zugriffsschutz von Klassen

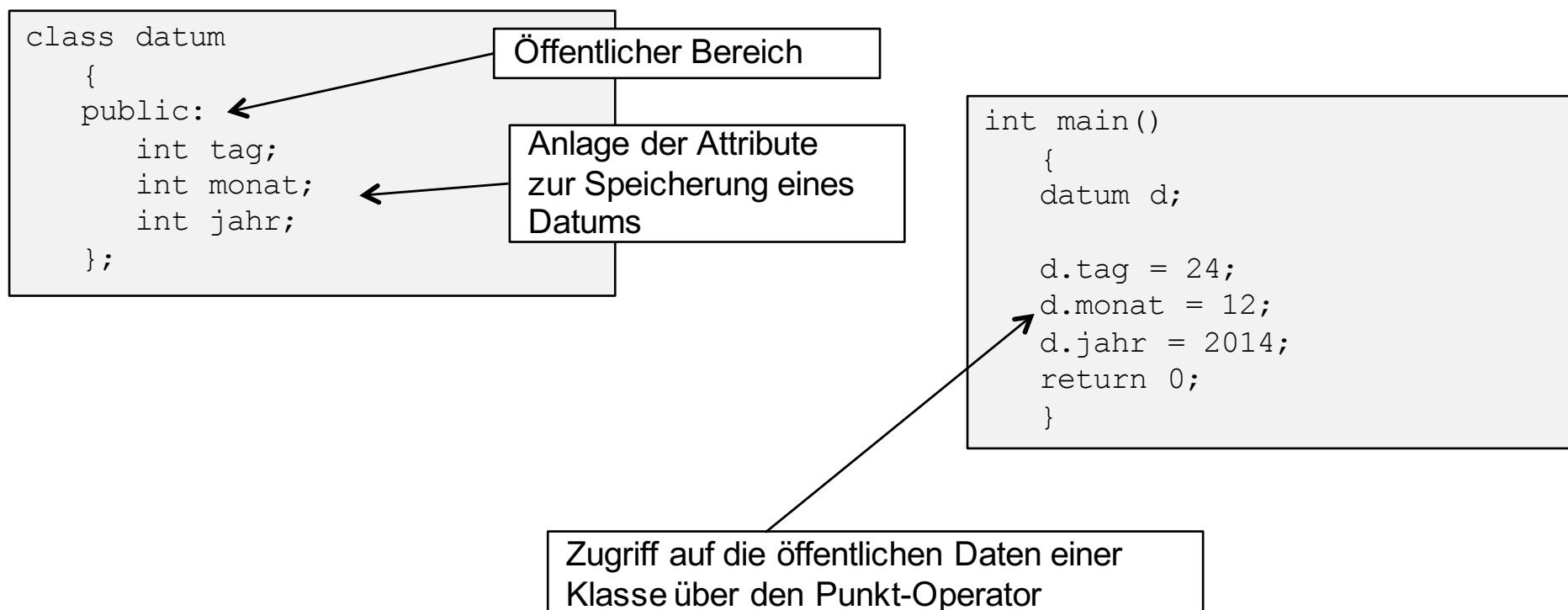
Anders als bei Datenstrukturen gibt es also bei Klassen einen Schutz gegen die missbräuchliche Verwendung der in den Klassen enthaltenen Elemente. Um diesen Schutz wirksam werden zu lassen, werden die Elemente einer Klasse in entsprechenden Bereichen abgelegt. Auf Elemente im privaten Bereich „private“ besteht besonderer Zugriffsschutz. Auf Elemente im öffentlichen Bereich „public“ kann jeder zugreifen. Die entsprechenden Bereiche werden innerhalb der Klassendefinition durch die jeweiligen Schlüsselwörter gekennzeichnet. Elemente, die keinem Bereich zugeordnet sind, sind privat. Alle Bereiche können auch mehrfach und in beliebiger Reihenfolge vorkommen. Die hier angegebene Reihenfolge ist allerdings üblich und bietet sich aus Gründen der Übersichtlichkeit an.



Den geschützten Bereich „protected“ werden wir im Rahmen der Vererbung behandeln.

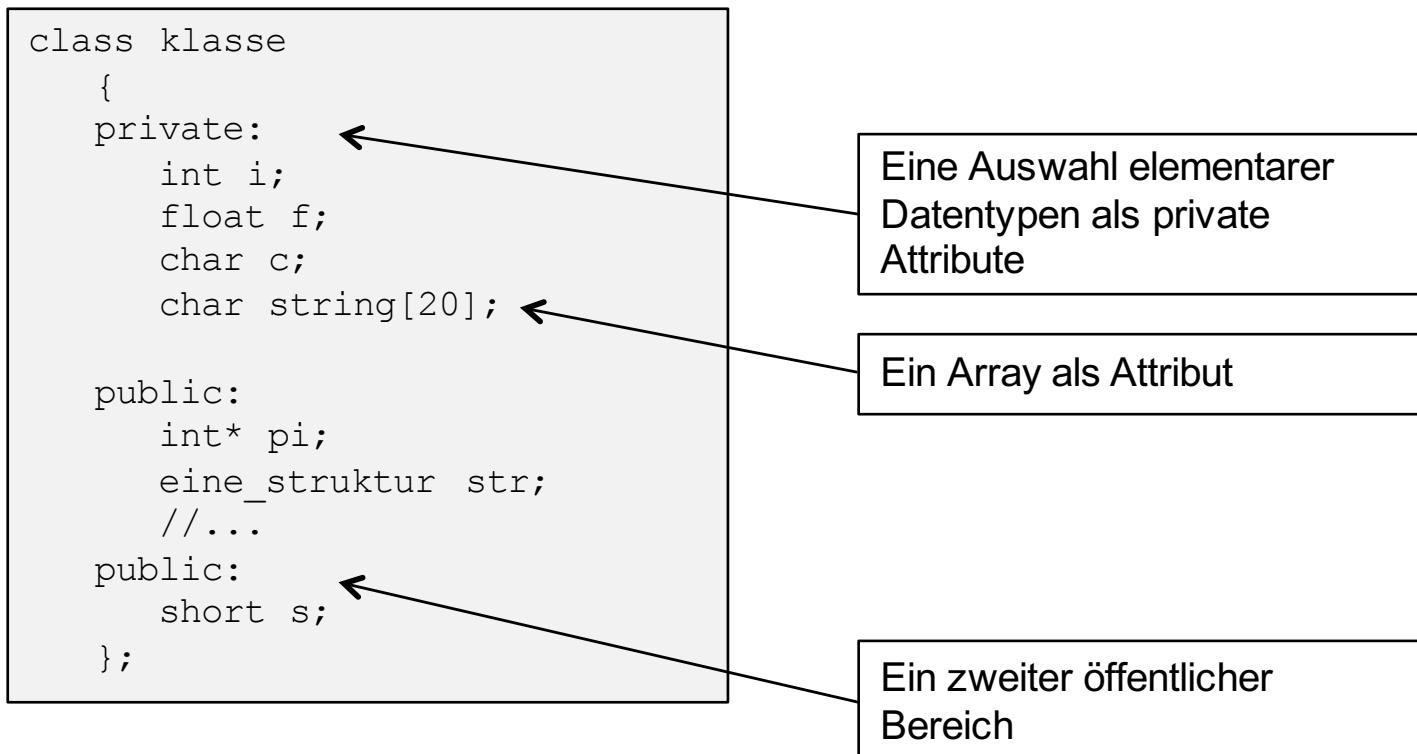
## Attribute in Klassen

Die Daten, die wir in Klassen speichern, werden Attribute oder Datenmember genannt. Zur Umsetzung unserer Datumsklasse könnten wir die notwendigen Daten im öffentlichen Bereich der Klasse speichern. Damit ist ein lesender und schreibender Zugriff von überall her möglich. Der Zugriff auf die Attribute findet wie bei Strukturen mit dem Punkt-Operator statt. Bis zu dieser Stelle ist unsere Klasse mit einer Datenstruktur identisch. Faktisch entspricht eine Klasse mit ausschließlich öffentlichen Datenmembern einer Datenstruktur.



## Mögliche Datentypen der Attribute

Generell können in einer Klasse alle elementaren Datentypen und deren Arrays als Attribute verwendet werden, dies umfasst auch Datenstrukturen. Es können auch Klassen als Datenelemente von Klassen vorkommen, aber das wird in einem eigenen Kapitel behandelt.



## Erweiterung von Klassen um Verhalten

Wir haben in der Vergangenheit bereits ausgiebig mit Datenstrukturen und Funktionen gearbeitet. Dabei waren die Datenstrukturen und Funktionen strikt getrennt. Formal waren sie damit zwar unabhängig, wenn man sich deren Zusammenwirken ansieht, waren Daten und Funktionen aber durchaus stark miteinander verflochten.

Betrachtet man beispielsweise das behandelte Listen-Modul aus Kapitel 16, ist offensichtlich, dass die Datenstruktur erst mit den Listenoperationen (`create`, `insert`, `remove`, `find`) sinnvoll verwendbar wird. Andererseits können die Operationen nur mit der passenden Datenstruktur arbeiten und sind ohne diese Struktur nutzlos.

Bei einer Anpassung an der Datenstruktur folgen typischerweise auch notwendige Anpassungen der Operationen. Andersherum erfordern Änderungen an der Funktionalität der Operationen vielfach auch eine Änderung der Datenstrukturen. Faktisch besteht also ein starker Zusammenhang.

Eine unkoordinierte Änderung oder Erweiterung an Datenstrukturen oder Operationen kann dabei schnell zu einem nicht mehr überschaubaren und damit nicht mehr wartbaren System führen.

Es liegt also nahe, Datenstruktur und Operationen zu kombinieren. Dies passiert in einem Objekt, in dem zusammengehörige Daten und Funktionen als Member einer Klasse zusammengeführt werden.

## Funktions-Member einer Klasse

Wir wollen nun unserer Datums-Klasse eine erste Methode und damit Verhalten hinzufügen. Dabei wollen wir mit einer Methode beginnen, die übergebene Parameter für `tag`, `monat` und `jahr` in den entsprechenden Datenmembern speichert.

Wir werden die Methode `set` nennen. Solche Methoden, die die Werte von Datenmembern setzen werden oft auch als Setter-Methoden bezeichnet. Unsere Methode wird die folgende Schnittstelle besitzen:

```
void set( int t, int m, int j )
```

Parameter für die zu setzenden Werte  
von `tag`, `monat` und `jahr`

Dieses Verhalten erscheint im Moment überflüssig. Wie wir ja bereits gesehen haben, können wir die Werte ja einfach direkt schreiben. Der Nutzen dieser Methode wird sich aber schnell zeigen.

## Setzen der Datenmember über eine Methode

Wir platzieren unsere Setter-Methode mit ihren drei Parametern ebenfalls im öffentlichen Bereich der Klasse. So kann jeder auf sie zugreifen.

Methoden können in der Klassendeklaration als sogenannte **Inline-Methoden** implementiert werden. Sie werden dazu direkt in die Klassendeklaration geschrieben. Dies sieht auf den ersten Blick etwas ungewöhnlich aus, wenn man die Formatierung anpasst, erkennt man aber schnell die gewohnte Form.

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

Anlage einer Funktion **in** der Klasse  
mit einer **Inline-Implementierung**

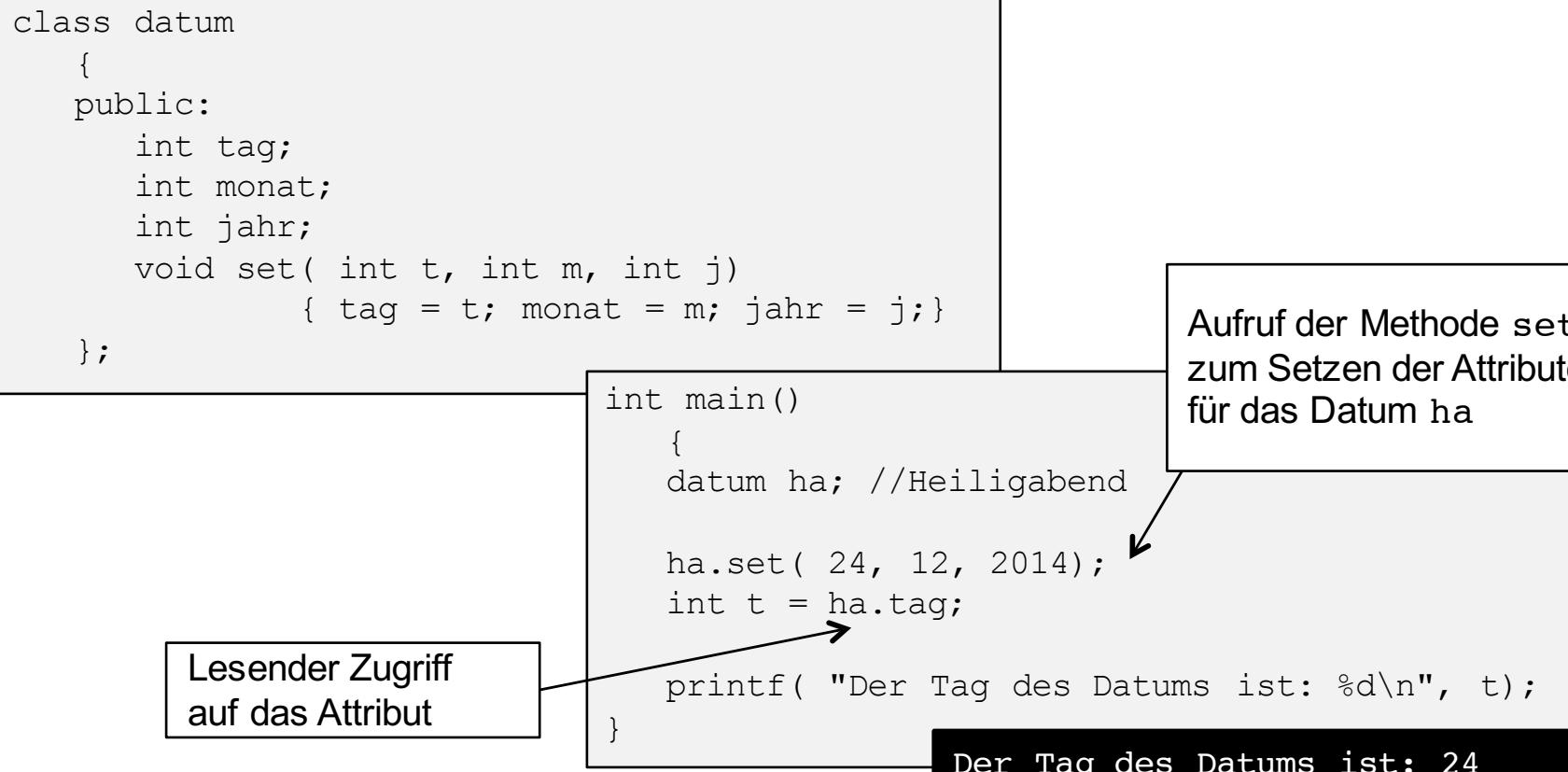
Die Attribute sind für die Klasse definiert, zu der auch die Methode gehört. Der Zugriff auf Attribute „innerhalb“ der Klasse erfolgt direkt, ohne Punkt-Operator

Code umgestellt in die bekannte Form

```
void set( int t, int m, int j)
{
    tag = t;
    monat = m;
    jahr = j;
}
```

## Verwendung der Funktions-Member

Wir haben eine öffentliche Methode zum Schreiben der Attribute implementiert. Der Aufruf einer solchen Methode erfolgt wie der Zugriff auf die Attribute einer Klasse über den Punkt-Operator.



## Member im privaten Bereich einer Klasse

Wir hatten zuerst alle Attribute in den öffentlichen Bereich gelegt. Wir wollen mit unserer Klasse aber die Integrität unseres Objekts sicherstellen. So wie nur berechtigte Komponenten den Kilometerstand unseres Autos konsistent ändern können, so wollen wir auch vermeiden, dass jeder auf die Attribute unseres Objekts zugreifen kann. Wir erklären daher die Datenmember für private und definieren sie dazu in einem privaten Bereich. Wir belassen nur die set-Methode im öffentlichen Bereich. Die set-Methode bleibt damit auch weiter aus unserem Hauptprogramm aufrufbar. Die Methode selbst gehört mit zur Klasse, sie befindet sich „innen“, daher darf sie ohne Einschränkungen auf private Elemente der Klasse zugreifen. Der Zugriff auf das Datenmember tag von außen über den Punkt-Operator schlägt jetzt allerdings fehl. Der Compiler quittiert mit einer Fehlermeldung, da dies nun als ein unerlaubter Zugriff auf den privaten Bereich gilt.

```
class datum
{
    private:
        int tag;
        int monat;
        int jahr;

    public:
        void set( int t, int m, int j)
        { tag = t; monat = m; jahr = j; }
};
```

Anlage der Attribute zur Speicherung eines Datums jetzt im privaten Bereich der Klasse

Implementierung der set-Methode weiter im öffentlichen Bereich

```
int main()
{
    datum ha; //Heiligabend

    ha.set( 24, 12, 2014);
    int t = ha.tag; // FEHLER
}
```

Zugriff auf das private Attribut der Klasse von außen schlägt fehl

## Lesender Zugriff auf die Datenmember

Wir wollen unsere Attribute hier aber nicht nur setzen können, sondern wollen sie auch auslesen. Die Lösung für dieses Problem liegt nahe. Wir erstellen zu jedem zu lesenden Attribut eine Methode, die das Attribut liest und den entsprechenden Wert als Resultat zurückgibt. Solche lesenden Methoden werden auch Getter-Methoden genannt. Auch unsere Getter legen wir als Inline-Methoden an und platzieren sie im öffentlichen Bereich.

Nun können wir den direkten lesenden Zugriff auf das Attribut durch unsere Getter-Methode ersetzen und erhalten den gewünschten Wert. Damit können wir über unsere Methoden jetzt mittelbar wieder auf die privaten Attribute zugreifen.

```
class datum
{
private:
    int tag;
    int monat;
    int jahr;

public:
    int getTag() { return tag; }
    int getMonat() { return monat; }
    int getJahr() { return jahr; }

    void set( int t, int m, int j)
        { tag = t; monat = m; jahr = j; }
};
```

Inline-Implementierung  
der Getter-Methoden

```
int main()
{
    datum ha; //Heiligabend

    ha.set( 24, 12, 2014);

    int t = ha.getTag();
}
```

Zugriff auf den Wert des  
privaten Attributes mit Hilfe  
einer Getter-Methode

## Verwendung der Getter- und Setter-Methoden

Unsere Klasse hat ihre Daten nun „gekapselt“. Die Zugriffe auf die internen Daten, die den Zustand beschreiben, ist nur noch über die bereitgestellten Methoden möglich.

Wir haben jetzt die Option, schreibenden Zugriff so zu gestalten, dass fehlerhafte Eingaben korrigiert oder abgelehnt werden, ohne die Instanz in einen inkonsistenten Zustand zu bringen.

## Funktionen außerhalb der Klasse

Wir wollen unsere Setter-Methode entsprechend erweitern, dass sie vor dem Setzen der Attribute eine einfache Prüfung vornimmt und ungültige Werte korrigiert. Die Prüfungen und die Korrektur halten wir hier bewusst einfach. Wir wollen für unser Datum annehmen, das es gültige Daten zwischen dem 01.01.1970 und dem 30.12.2099 aufnehmen soll. Wir gehen davon aus, dass alle Monate 30 Tage lang sind. Selbst mit dieser Einschränkung ist bereits jetzt abzusehen, dass die Implementierung dieser Methode mehrere Zeilen in Anspruch nehmen wird. Eine Inline-Implementierung wird hier schnell unübersichtlich. Wir wollen daher dieses Funktions-Member außerhalb der Klasse implementieren. Dazu ersetzen wir zuerst unsere bisherige Implementierung durch eine Deklaration der Methode:

Deklarationen der Methode ohne  
Implementierung

```
class datum
{
    private:
        int tag;
        int monat;
        int jahr;

    public:
        int getTag() { return tag; }
        int getMonat() { return monat; }
        int getJahr() { return jahr; }

        void set( int t, int m, int j );
};
```

## Implementierung von Funktionen außerhalb der Klasse

Die Methode die innerhalb der Klasse deklariert worden ist, wird nun außerhalb der Klasse implementiert. Den Code zur Korrektur der Daten selbst, werden wir hier nicht weiter vertiefen. Ungültige Werte werden immer auf einen vorgegebenen Wert korrigiert.

```
class datum
{
private:
    int tag;
    int monat;
    int jahr;
public:
    int getTag() { return tag; }
    int getMonat() { return monat; }
    int getJahr() { return jahr; }
    void set(int t, int m, int j);
};
```

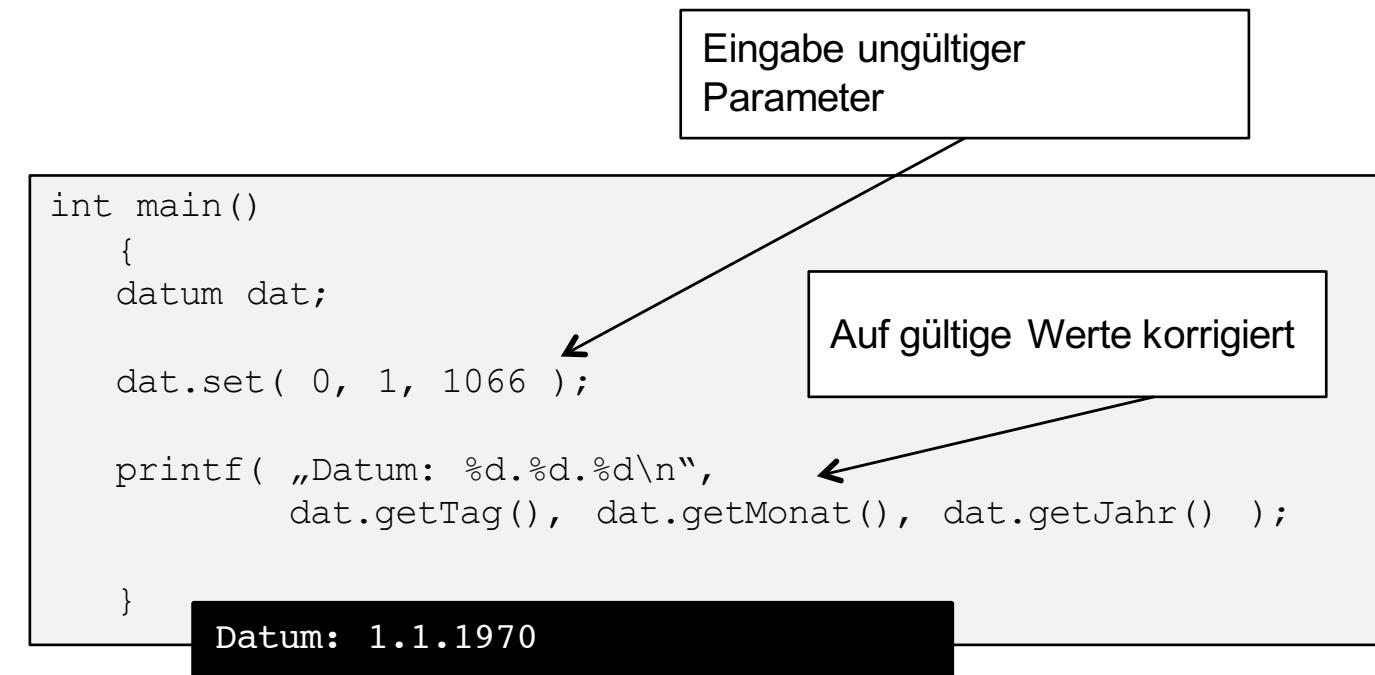
Voll qualifizierter  
Name der Klasse und  
Methode

```
void datum::set< int t, int m, int j >
{
    if( j<1970 || j>2099 )
        j = 1970;
    if( m<1 || m>12 )
        m = 1;
    if( t<1 || t>30 )
        t = 1;
    tag = t; monat = m; jahr = j;
}
```

Implementation der  
deklarierten Methode

## Verwendung der neuen Setter-Methode

Mit unserer neuen Setter-Methode werden falsche Datumswerte nun bei der Eingabe entsprechend korrigiert, so dass unser Objekt nach Aufruf der set-Methode immer ein gültiges Datum gemäß der vorgegebenen Regeln enthält:



## Erstellung und Beseitigung von Objekten

Wir können jetzt Instanzen unserer Klasse `datum` anlegen und haben mit der `set`-Methode dafür gesorgt, dass nur noch im Rahmen unserer Anforderungen gültige Datumswerte in die Attribute eingetragen werden können.

Damit ist aber immer noch nicht garantiert, dass die Attribute des Objektes immer mit konsistenten Werten gefüllt sind. Dies liegt daran, dass wir bisher nicht kontrollieren können, in welchem Zustand das Objekt erstellt wird. Direkt nach seiner Instanziierung ist der Zustand des Objektes noch undefiniert.

Wir benötigen eine Möglichkeit, bereits die Erstellung der Instanz zu beeinflussen. Diese Möglichkeit gibt es mit dem sogenannten **Konstruktor**. Der Konstruktor steuert den Instanziierungsprozess eines Objektes und ist dafür verantwortlich, die Instanz bei der Erstellung direkt in einen konsistenten Zustand zu bringen.

Das Gegenstück zum Konstruktor ist der **Destruktor**, der den Abbau einer Instanz steuert. Diesen werden wir abschließend behandeln.

## Der Konstruktor eines Objektes

Der Konstruktor einer Klasse ist eine spezielle Methode. Sie ist so eng mit der Klasse verknüpft, dass sie den gleichen Namen trägt wie die Klasse. Ein Konstruktor hat **keinen** Rückgabetyp, nicht einmal `void`.

Eine Klasse kann keinen, einen oder mehrere Konstruktoren haben. Wie andere Methoden auch, kann der Konstruktor parameterlos sein. Wenn eine Klasse mehrere Konstruktoren hat, wird wie bei überladenen Funktionen der passende Konstruktor ausgewählt. Dazu werden die Parameter verwendet, die bei der Instanziierung angegeben worden sind. Entsprechend müssen sich auch die Parametersignaturen der Konstruktoren unterscheiden.

## Konstruktor für die Klasse datum

Wir wollen nun unserer Klasse einen Konstruktor hinzufügen. Der Konstruktor soll von überall aufrufbar sein, daher platzieren wir ihn im öffentlichen Bereich. Wir implementieren ihn wie unseren Setter außerhalb der Klassendeklaration. Seine Implementation greift auf den bestehenden Setter zurück.

```
class datum
{
private:
    int tag;
    int monat;
    int jahr;
public:
    int getTag() { return tag; }
    int getMonat() { return monat; }
    int getJahr() { return jahr; }
    void set( int t, int m, int j );
    datum( int t, int m, int j );
};
```

Deklaration und  
Definition eines  
Konstruktors

```
datum::datum( int t, int m, int j )
{
    set(t, m, j);
}
```

Eine bestehende Methode (inklusive der  
existierenden Prüfungen) wird im Konstruktor  
verwendet

## Einsatz des Konstruktors

Wir wollen nun unseren neu erstellten Konstruktor direkt zum Einsatz bringen und erhalten das erwartete Ergebnis:

```
int main()
{
    datum em( 1, 5, 2014 ); //1. Mai
    printf( "1. Mai: %d.%d.%d\n",
            em.getTag(), em.getMonat(), em.getJahr() );
}
```

1. Mai: 1.5.2014

Wir bedienen hier die Schnittstelle unseres Konstruktors. Wenn es mehrere Konstruktoren gibt, wird der passende anhand der Parametersignatur ausgewählt

Sobald wir einen Konstruktor mit Parametern erstellt haben, ist der bisher verwendet parameterlose Konstruktor, nicht mehr verfügbar. Solange wir keinen Konstruktor explizit bereitgestellt hatten, war er vom System automatisch erstellt worden. Dies ist jetzt nicht mehr der Fall. Die folgende Erstellung eines Objektes ist damit jetzt nicht mehr möglich

```
int main()
{
    datum ha; // FEHLER
}
```

Compilerfehler mit dem Hinweis der Konstruktor sei nicht verfügbar

## Erstellung eines parameterlosen-Konstruktors

C++ erstellt für eine Klasse **automatisch** einen Konstruktor ohne Parameter, wenn für eine Klasse kein Konstruktor definiert worden ist. Bisher haben wir, ohne es zu wissen, diesen automatisch erstellten Konstruktor verwendet. Nachdem wir aber unseren ersten eigenen Konstruktor erstellt haben, wird der Default-Konstruktor nicht mehr vom System erzeugt. Wenn wir weiter einen Konstruktor ohne Parameter für unsere Klasse verwenden wollen, müssen wir ihn selbst bereitstellen.

```
class datum
{
private:
    int tag;
    int monat;
    int jahr;
public:
    int getTag() { return tag; }
    int getMonat() { return monat; }
    int getJahr() { return jahr; }
    void set( int t, int m, int j );
    datum( int t, int m, int j );
    datum() { set( 1, 1, 1970 ); }
};
```

```
int main()
{
    datum em( 1, 5, 2014 ); // 1. Mai

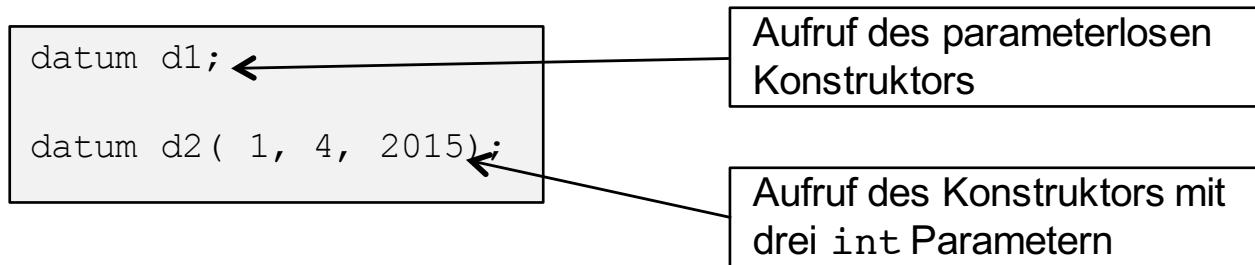
    printf( "1. Mai: %d.%d.%d\n",
            em.getTag(), em.getMonat(), em.getJahr() );

    datum ha; //Heiligabend
    ha.set( 24, 12, 2014 );
}
```

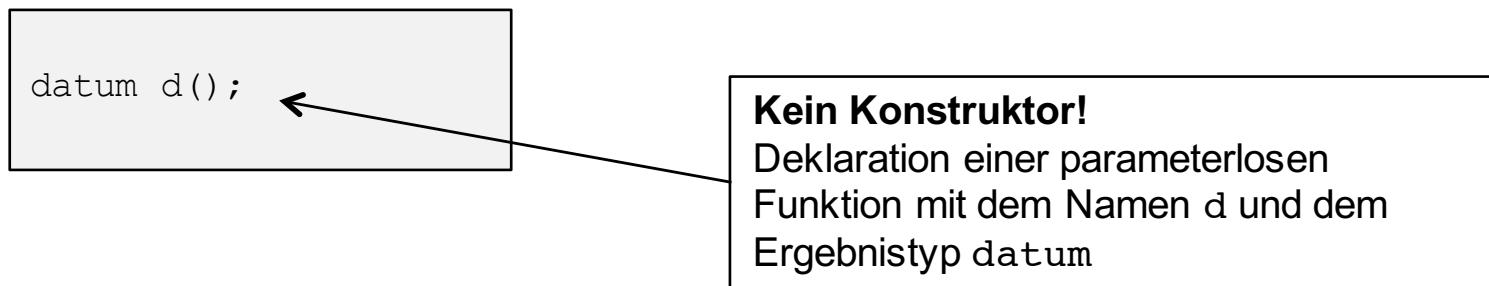
Es steht ein parameterloser Konstruktor zur Verfügung, die Zeile kann übersetzt und ausgeführt werden

## Aufruf von Konstruktoren

Für den Aufruf von Konstruktoren wollen wir noch einen kurzen Blick auf die Notation werfen.  
Bei den nachstehenden Notationen handelt es sich um gültige Aufrufe unserer Konstruktoren:



Die folgende Konstruktion sieht nur auf den ersten Blick wie der Aufruf eines parameterlosen Konstruktors aus, ist aber kein solcher. Die Zeile instanziert keine Klasse sondern deklariert lediglich eine Funktion.



## Destruktor für die Klasse datum

Der Destruktor ist das Gegenstück zum Konstruktor. Er wird aufgerufen, wenn ein Objekt zerstört wird. Im Destruktor können Aufräumarbeiten wie die Freigabe der vom Objekt belegten Ressourcen vorgenommen werden. Dies betrifft insbesondere die Freigabe von allokiertem Speicher. Jede Klasse kann maximal einen Destruktor haben. Der Destruktor hat wie der Konstruktor keinen Rückgabewert und ist immer parameterlos. Der Destruktor hat den Namen der Klasse, angeführt von einer vorangesetzten Tilde. Der Destruktor kann nicht explizit aufgerufen werden. Er wird vom System gerufen, wenn eine Instanz abgebaut wird.

Die Klasse datum benötigt keine besonderen Aufräumarbeiten. Zu Demonstrationszwecken haben wir einen funktionslosen Destruktor erstellt. Hier könnte der Destruktor komplett entfallen, wir werden aber noch Klassen kennenlernen, bei denen der Destruktor wichtig ist.

```
class datum
{
private:
    int tag;
    int monat;
    int jahr;
public:
    int getTag() { return tag; }
    int getMonat() { return monat; }
    int getJahr() { return jahr; }
    void set( int t, int m, int j );
    datum( int t, int m, int j );
    datum() { set( 1, 1, 1970); }
    ~datum() {}
```

Der Destruktor hat in dieser Klasse keine Aufräumarbeiten zu erledigen, er bleibt leer

## Instanziierung von Klassen

Im Umfeld des Konstruktors ist von Instanziierung und Abbau oder Zerstörung von Objekten gesprochen worden. Wir werden daher die Instanziierung in C++ noch einmal genauer betrachten. Als Einstieg werden wir die verschiedenen Formen der Variablenanlage in C noch betrachten und parallel dazu die Situation bei der Instanziierung von Objekten in C++ vergleichen.

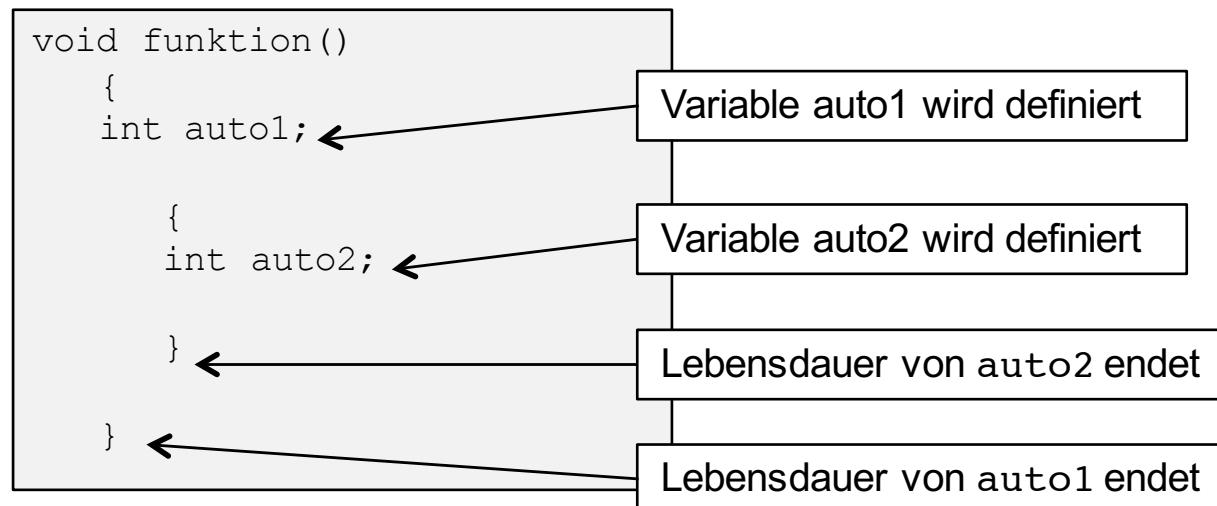
In C können Variablen auf drei Arten angelegt werden

- automatisch
- statisch
- dynamisch

In C++ existieren die genannten Arten weiter. Wir haben aber schon gesehen, dass durch Konstruktoren und Destruktoren bei Erzeugung und Abbau von Instanzen zusätzlicher Code ausgeführt wird. Die einzelnen Abläufe im Lebenszyklus sind damit komplexer als in C.

## Automatische Variablen in C

Automatische Variablen werden innerhalb von Blöcken angelegt. Sie haben die Lebensdauer des umschließenden Blockes.

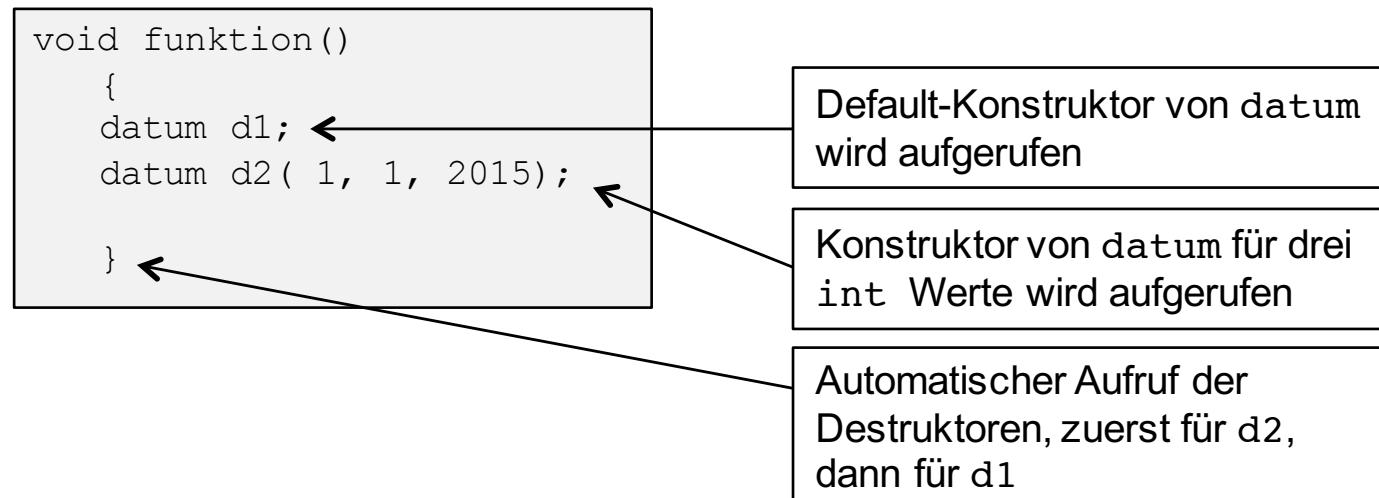


## Automatische Instanziierung in C++

Objekte einer Klasse werden automatisch instanziert, wenn eine Variable dieser Klasse in einem Block angelegt wird. Dabei wird der Konstruktor aufgerufen, dessen Signatur zu den mitgegebenen Parametern passt. Jedes mal, wenn der Programmablauf die Definition passiert, wird das Objekt neu instanziert.

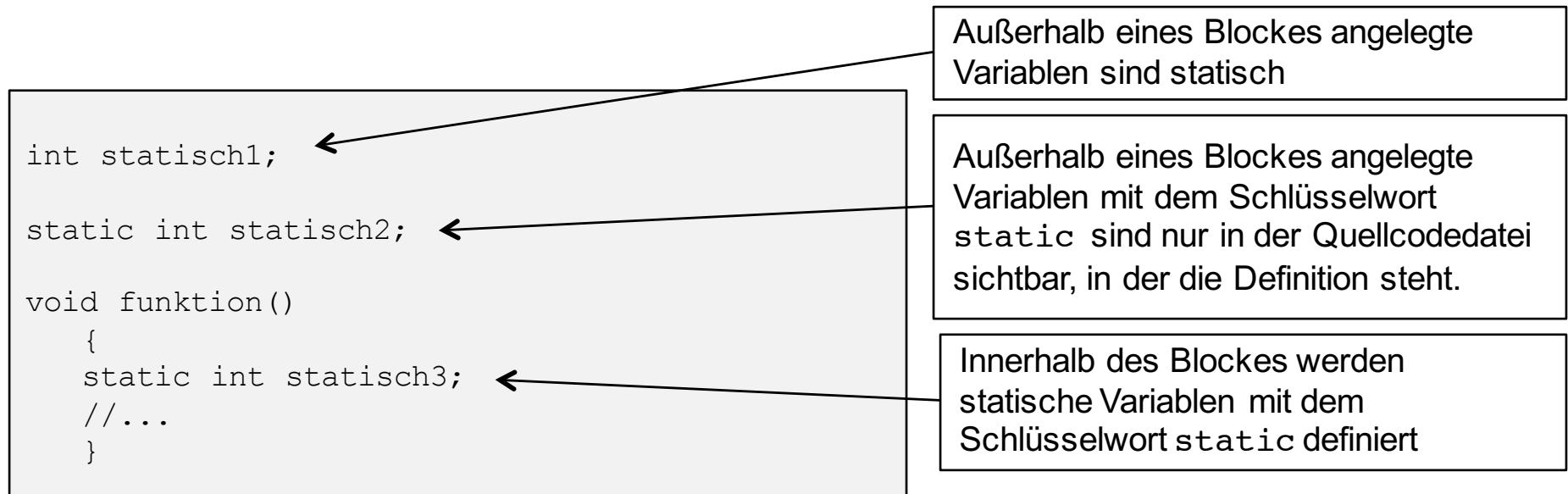
Automatische Objekte werden durch einen gegebenenfalls vorhandenen Destruktor beseitigt, sobald der Block verlassen wird, in dem sie definiert wurden.

Die Destruktoren laufen in der umgekehrten Reihenfolge der Konstruktoren ab. Was zuletzt konstruiert wurde, wird zuerst beseitigt.



## Statische Variablen in C

Statische Variablen werden außerhalb von Blöcken angelegt oder mit Hilfe des Schlüsselwortes `static` gekennzeichnet.



## Statische Instanziierung in C++

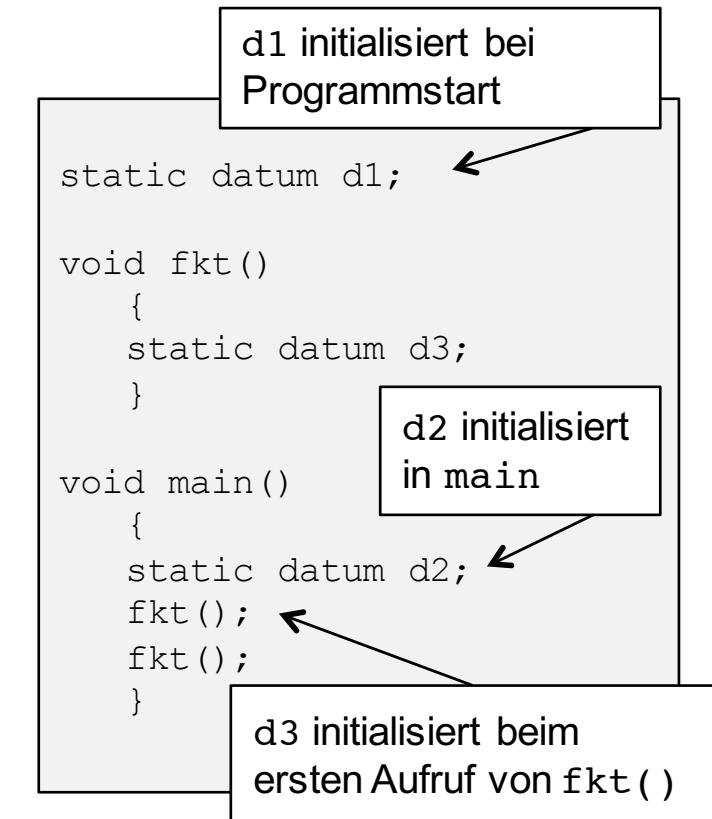
Da in C++ mit den Konstruktoren und Destruktoren Code implizit aufgerufen wird, ist es hier wichtig zu unterscheiden, ob statische Objekte innerhalb oder außerhalb von Funktionen angelegt werden:

- Anlage eines statischen Objektes außerhalb von Funktionen**

Die Objekte werden vor dem eigentlichen Programmstart, also noch vor der Ausführung des in `main` enthaltenen Codes instanziert, entsprechende Konstruktoren werden ausgeführt.

- Anlage eines statischen Objektes innerhalb von Funktionen**

Das Objekt wird einmalig instanziert, wenn der Programmablauf erstmalig dessen Deklaration passiert. Wenn ein statisches Objekt innerhalb einer Funktion einmal angelegt ist, bleibt es bestehen und wird auch bei erneutem Passieren des Codes nicht neu initialisiert. Es wird nicht bei Verlassen der Funktion, sondern erst bei Programmende beseitigt.



Statische Objekte werden nach der Beendigung des Programms in der umgekehrten Reihenfolge der Instanziierung beseitigt. Für die Angabe zur Reihenfolge der Destruktion gelten die gleichen Regeln wie bei automatischen Objekten. Wenn die Objekte ohne Seiteneffekte arbeiten, sollte die Reihenfolge allerdings keine Rolle spielen.

## Dynamische Variablen in C

Dynamische Variablen werden zur Laufzeit angelegt und existieren bis zur expliziten Beseitigung. Wird der dynamisch reservierte Speicher nicht freigegeben, ist er bis zum Programmende belegt.

```
void funktion()
{
    int *pi;

    pi = (int*)malloc(sizeof(int));
    //...

    free ( pi );
}
```

Dynamisches Allokieren des Speichers

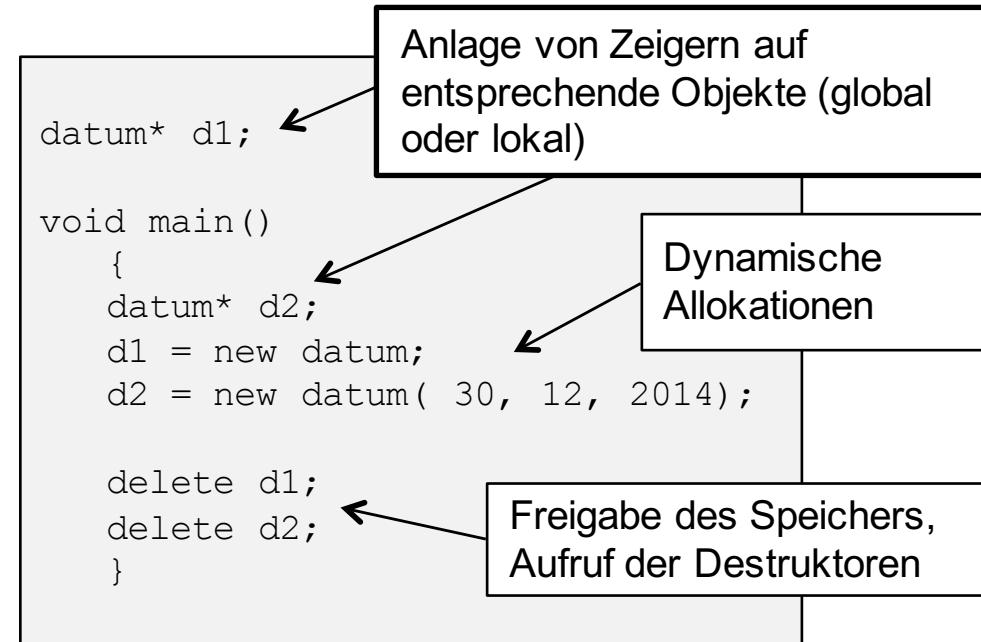
Freigabe des allokierten Speichers

## Dynamische Instanziierung in C++

In C haben wir benötigten Speicher mit den Funktionen `malloc` und `calloc` der Laufzeitbibliothek allokiert. Zum Allokieren von Klassen können diese Funktionen **nicht** verwendet werden.

Um ein Objekt dynamisch zu allokiieren, reicht es nicht aus, nur den entsprechenden Speicher bereitzustellen. Es ist unbedingt notwendig, dass auch ein geeigneter Konstruktor der Klasse ausgeführt wird. Um dies sicherzustellen, wird in C++ der **new-Operator** verwendet, um Klassen dynamisch zu instanziieren. Die Parameter für einen geeigneten Konstruktor werden dem new-Operator mitgegeben.

Der new-Operator gibt als Ergebnis einen Zeiger auf den erzeugten Objekttyp zurück.

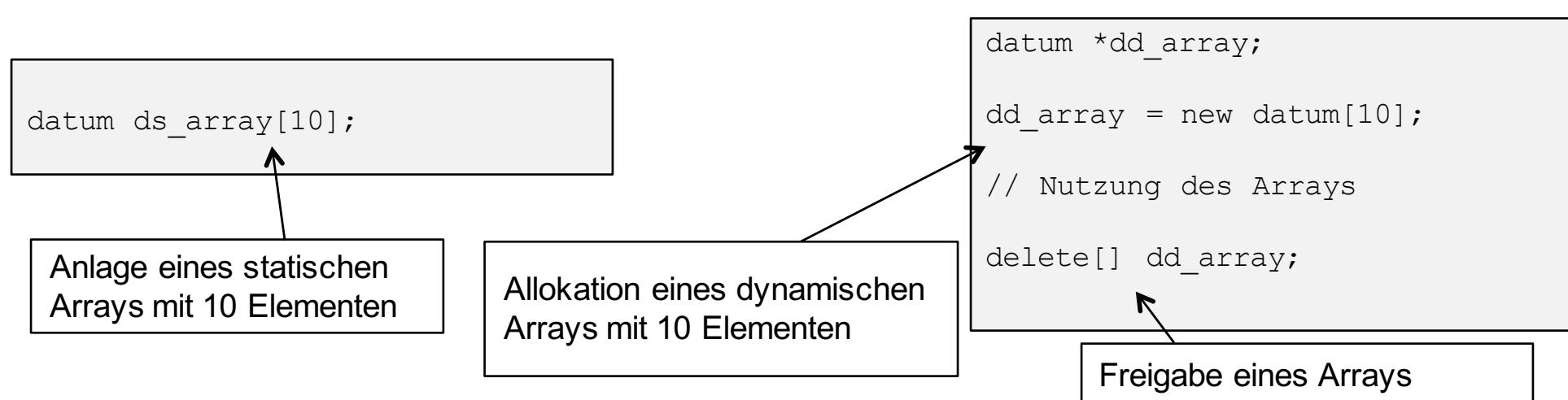


Mit dem **delete-Operator** wird der Destruktor (soweit vorhanden) für dynamisch angelegte Klassen ausgeführt und der allokierte Speicher freigegeben.

## Instanziierung von Arrays in C++

Arrays können auch in C++ auf die bereits bekannte Art und Weise angelegt werden. Um ein Array von Objekten einer Klasse zu erzeugen, muss die entsprechende Klasse einen parameterlosen Konstruktor besitzen. Ob es sich dabei um einen explizit erstellten oder eine automatisch vom System bereitgestellten Konstruktor handelt, ist egal. Eine Übergabe von Parametern an den Konstruktor und eine individuelle Instanziierung ist nicht möglich. Der Aufruf der Konstruktoren erfolgt in der Reihenfolge des wachsendem Index.

Ein Array von Objekten kann auch dynamisch allokiert werden. Die Anzahl der gewünschten Objekte wird auch hier in den eckigen Klammern angegeben. Ein Array von Objekten wird mit dem **delete[]**-Operator freigegeben. Es handelt sich hierbei um einen eigenen Operator, der für die korrekte Freigabe eines dynamischen Arrays verwendet werden **muss**.



## Operatoren für Klassen überladen

Wir können auch für Klassen überladene Operatoren definieren. Das eröffnet uns die Möglichkeit, viele Operationen elegant und übersichtlich zu notieren. Mit einem überladenen Operator haben wir beispielsweise die Möglichkeit, die Anzahl von Tagen zwischen zwei Daten einfach als deren Differenz zu betrachten und einen entsprechenden Operator zu erstellen. Die Verwendung sieht folgendermaßen aus:

```
datum ha( 24, 12, 2014 );
datum zf( 26, 12, 2014 );

int diff = zf - ha;
printf( "Heiligabend und zweiter Feiertag liegen %d Tage auseinander\n", diff );
```

Heiligabend und zweiter Feiertag liegen 2 Tage auseinander

Den entsprechenden Operator können wir bereits mit den uns bekannten Mitteln implementieren. Für Datumsoperationen bietet es sich oft an, einen Fixpunkt zu wählen. Dazu ermitteln wir für jedes der beiden Daten die Anzahl der vergangenen Tage, die von einem imaginären „nullten Januar im Jahr Null“ bereits vergangen sind. Aus diesen Werten ermitteln wir dann die Differenz.

```
int operator-( datum& l, datum& r )
{
    int tage_l = 360*l.getJahr() + 30*l.getMonat() + l.getTag();
    int tage_r = 360*r.getJahr() + 30*r.getMonat() + r.getTag();
    return tage_l - tage_r;
}
```

Anzahl der Tage die am 0. eines Monats im Jahr bereits vergangen sind

Berechnung der Differenz

## Verwendung des überladenen Operators

Wir können unseren überladenen Operator nun verwenden wie die Operatoren der integrierten Datentypen:

```
datum ha( 24, 12, 2014 );
datum zf( 26, 12, 2014 );

int diff = zf - ha;
printf( "Heiligabend und zweiter Feiertag liegen %d Tage auseinander\n", diff );
```

Heiligabend und zweiter Feiertag liegen 2 Tage auseinander

## Friends

In dem vorangegangenen Beispiel haben wir den Operator unter Verwendung der Getter-Methoden implementiert. Das war notwendig, da die Funktion `operator-` nicht zur Klasse `datum` gehört und daher keinen Zugriff auf die privaten Datenmember hat. Nicht nur hier kann sich der umfassende Zugriffsschutz auch als lästig erweisen. In Bibliotheken zusammengehöriger Klassen wollen wir thematisch verbundene Klassen untereinander oft weitergehende Zugriffsrechte einräumen. Um das zu erreichen, können wir hier bisher nur nach dem Motto „alles oder nichts“ Member im öffentlichen Bereich der Klasse platzieren.

Um differenzierter vorzugehen, kann eine Klasse anderen Funktionen oder Klassen einen besonderen Status zuerkennen und sie zu einem „Freund“ erklären. Diese Freunde erhalten dann den gleichen Status wie Member-Funktionen der entsprechenden Klassen und können damit auf deren private Member zugreifen. Friend-Funktionen werden oft verwendet, wenn Funktionen erstellt werden, die auf unterschiedliche Klassen zugreifen müssen, aber keiner der Klassen zugeordnet werden sollen.

Die Freundschaftserklärung gilt nur in eine Richtung. Das bedeutet ich kann jemand anderen zu meinem Freund erklären und ihm damit besondere Zugriffsrechte an meinen privaten Daten einräumen. Ich kann mich aber nicht selbst zu einem Freund von jemand anderem erklären und mir dadurch besondere Zugriffsrechte an dessen Daten einräumen.

Mit der Erklärung einer Klasse oder Funktion zum Freund sollte sparsam umgegangen werden. Es gibt Situationen, in denen diese Möglichkeit sinnvoll eingesetzt wird, ein freigiebiger Umgang mit Freundschaften deutet aber oft auch auf eine ungünstige Modellierung hin.

## Operatoren für Klassen als friend überladen

Wenn unsere Klasse `datum` die Funktion `operator-` zu einem Freund erklärt, erweitert sich die Deklaration der Klasse folgendermaßen:

```
class datum
{
    friend int operator-( datum& l, datum& r );
private:
    // ...
public:
    // ...
};
```

Erklärung der  
`friend`-Funktion

Bereich nicht spezifiziert,  
daher `private`

Damit lässt sich unser Operator etwas knapper und übersichtlicher implementieren, auch wenn die eigentliche Funktionalität identisch bleibt:

```
int operator-( datum& l, datum& r )
{
    int tage_l = 360*l.jahr + 30*l.monat + l.tag;
    int tage_r = 360*r.jahr + 30*r.monat + r.tag;
    return tage_l - tage_r;
}
```

Direkter Zugriff auf die  
privaten Attribute der Klasse

## Operatoren als Methode der Klasse

Es gibt noch eine weitere Möglichkeit, einer Klasse einen Operator hinzuzufügen. In diesem Fall wird der Operator als Methode der Klasse deklariert und implementiert:

```
class datum
{
//...
public:
//...
int operator-( datum& r);
};
```

Deklaration als öffentliche  
Methode

In diesem Fall benötigt ein zweistelliger Operator nur ein Argument. Der erste Operand ist implizit durch das Objekt gegeben, auf das der Operator als Member-Funktion ausgeführt wird. Das zweite, rechte Argument wird als Parameter der Methode übergeben. Der Aufruf erfolgt wie gewohnt:

```
datum ha( 24, 12, 2014 );
datum zf( 26, 12, 2014 );
int diff = zf - ha;
```

Die Implementation des Operators sieht damit folgendermaßen aus:

```
int datum::operator-( datum& r)
{
int tage_l = jahr*360 + 30*monat + tag;
int tage_r = r.jahr*360 + 30*r.monat + r.tag;
return tage_l - tage_r;
}
```

Der linke Operand `zf` ist das  
Objekt selbst, hier erfolgt  
direkter Zugriff

Der Parameter ist aus der aktuellen Klasse,  
wir haben Zugriff auf die privaten Attribute

## Ausgabe in C++

Zum Einstieg in C haben wir direkt zu Beginn einfache Ein- und Ausgabeoperationen mit `printf` und `scanf` eingeführt und verwendet.

Wie wir am Beispiel der `printf`-Funktion schon gesehen haben, arbeiten diese Funktionen auch unter C++ weiter.

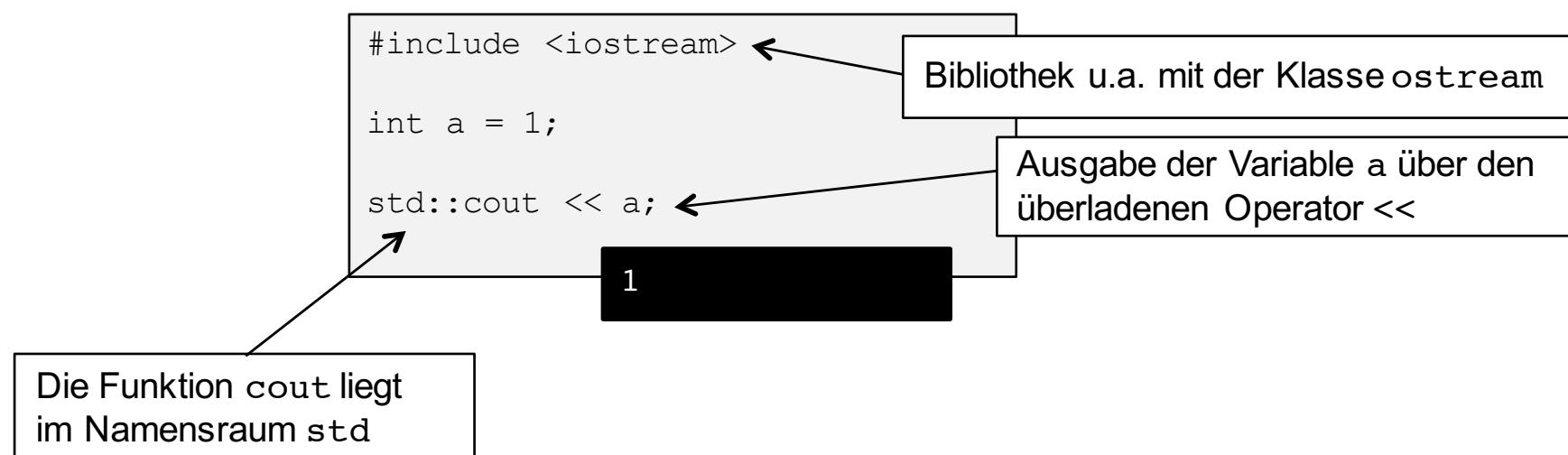
Allerdings gibt es für C++ auch ein System zur Ein- und Ausgabe, das das Klassenkonzept und die Möglichkeit zur Überladung von Operatoren ausnutzt. Dieses System werden wir im folgenden kurz kennenlernen, ohne in die Details einzutauchen.

## Bildschirmausgabe mit dem Operator <<

Um Text auf dem Bildschirm auszugeben, stellt eine Bibliothek in C++ das Objekt cout zur Verfügung. Das Objekt ist eine Instanz der Klasse ostream (für Output-Stream), die vom System zum Programmstart instanziert wird. Um cout nutzen zu können, muss zuvor die Bibliothek iostream inkludiert worden sein, so wie für printf die Bibliothek stdio inkludiert werden muss.

Die Ausgabe wird dann über die Methoden und überladenen Operatoren des cout Objektes angesprochen. Der wesentliche Operator für die Ausgabe ist der Operator '<<'. Mit diesem Operator können elementare Datentypen wie int oder float an den Ausgabestrom geleitet werden.

Die Ausgabe erfolgt dann folgendermaßen:



## Verkettete Ausgaben

Die Ausgaben nach cout können auch „verkettet“ werden:

```
#include <iostream>
using std::cout; // Alternativ: using namespace std;
int main()
{
    int a = 1;
    char c = 'X';
    char* s = "Text";
    cout << "Der Wert von a ist: " << a << '\n';
    cout << "Der Wert von c ist: " << c << '\n';
    cout << "Der Wert von s ist: " << s << '\n';
}
```

Verkettung von Ausgaben

Der Wert von a ist: 1  
Der Wert von c ist: X  
Der Wert von s ist: Text

Oft wird die Ausgabe in einen Stream auch mit dem endl-Objekt beendet. Dies bewirkt nicht nur einen Zeilenumbruch, sondern auch die sofortige Ausgabe des Streams. In großer Zahl verwendet kann sich dies nachteilig auf die Performance der Ausgabe auswirken.

```
cout << "Ausgabe mit Bufferleerung" << endl;
```

Zeilenumbruch und Bufferleerung

## Überladen des '<<' Operators

Neben den vom System bereitgestellten '<<' -Operatoren können wir auch entsprechende Operatoren für unsere Klasse überladen. Damit können wir eine elegante Ausgabe unserer Objekte erreichen. Ausgabeoperatoren werden häufig außerhalb der Klasse und als friend Funktionen definiert.

```
class datum
{
    friend ostream& operator<<( ostream& os, const datum& d);
private:
    //...
};
```

Rückgabewert ist wieder eine Referenz auf einen ostream zur Verkettung

ostream an den ausgegeben wird

Auszugebender Parameter. In der Ausgabe soll der Parameter nicht verändert werden und wird als konstante Referenz übergeben

```
ostream& operator<<( ostream& os, const datum& d)
{
    os << d.tag << '.' << d.monat << '.' << d.jahr;

    return os;
}
```

Rückgabe des ostream zur Verkettung

Die Ausgabe eines Objektes wird damit sehr einfach:

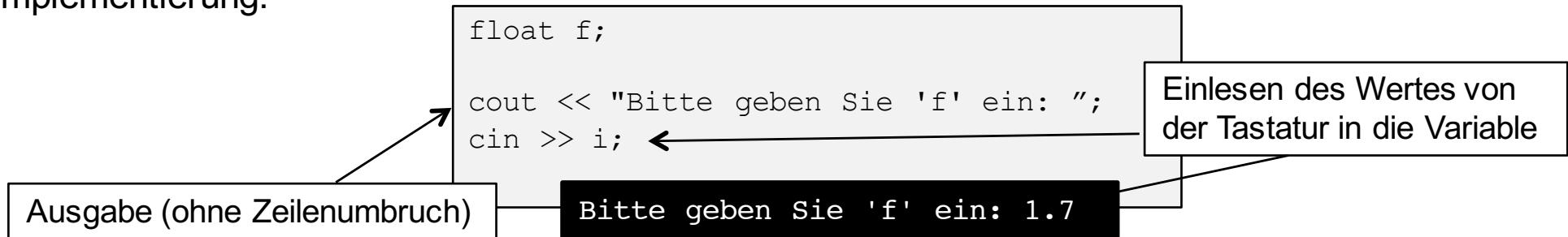
```
datum einheit( 3, 10, 1990);
cout << "Tag der Einheit: " << einheit << '\n';
```

Tag der Einheit: 3.10.1990

## Tastatureingabe

Wie die Ausgabe nutzt auch die Eingabe in C++ überladene Operatoren. Hier steht das Objekt `cin` im Zentrum, eine Instanz der Klasse `istream`. Bei der Eingabe wird der Operator '`>>`' verwendet. Wie sich bei dem Operator schon andeutet, ist der Ablauf der Eingabe wie bei der Ausgabe, lediglich die Flussrichtung dreht sich um.

Wollen wir beispielsweise einen Wert in eine `int` Variable einlesen, verwenden wir die folgende Implementierung:



Der '`>>`-Operator verwendet Referenzen, um den Wert in die angegebene Variable einzutragen. Daher ist der in C verwendete Adressoperator für die Eingabe hier nicht notwendig. Auch für den '`>>`-Operator gilt, dass die Überladungen für die elementaren Datentypen bereits existieren, beispielsweise ist damit folgende Eingabe möglich:

```
char name[100];
int alter;
cout << "Bitte geben Sie Ihren Namen ein: ";
cin >> name;
cout << "\nBitte geben Sie Ihr Alter ein: ";
cin >> alter;
```

## Überladen des '>>'-Operators

Wir können auch den '>>'-Operator für unsere eigenen Klasse überladen. Als Beispiel wollen wir einen einfachen Eingabeoperator für unsere Datums-Klasse erstellen. Für den Zugriff auf die privaten Attribute unserer Klasse, erklären wir auch diesen Operator zum friend:

```
friend istream& operator>>( istream& is, datum& d);
```

Übergabe als Referenz

```
istream& operator>>( istream& is, datum& d)
{
    int tag, monat, jahr;
    is >> tag;
    is >> monat;
    is >> jahr;
    d.set( tag, monat, jahr );
    return is;
}
```

```
datum d;
cout << "Geben Sie Tag, Monat und Jahr ein: " << '\n';

cin >> d;

cout << "Das Datum ist: " << d << '\n';
```

Geben Sie Tag, Monat und Jahr ein:

15

8

2015

Das Datum ist: 15.8.2015

## Dateioperationen

Auch für Dateioperationen verlässt sich C++ auf Objekte, Methoden und Operatoren. Die vorgestellten Operatoren zur Ein- und Ausgabe werden nicht nur für Ein- und Ausgaben auf der Konsole verwendet, sondern auch für Dateioperationen.

Für die Ausgabe auf dem Bildschirm stellt das System das Objekt cout bereit, an das wir mit dem überladenen Operator << ausgegeben haben. Eine Datei wird repräsentiert durch ein Objekt vom Datentyp ostream. Wir haben unseren Operator << für die Ausgabe an einen ostream definiert, wir können ihn aber auch für einen ofstream verwenden. Die Gründe dafür werden wir aber erst später bei der Vererbung kennenlernen.

Nach Einbinden des zugehörigen Headers fstream, können wir ein ofstream Objekt instanziieren und nutzen:

```
#include <fstream>
using namespace std;

int main()
{
    ofstream datei( "datum.txt" );
    datum d1( 1, 1, 2015 );
    datei << d1 << endl;
    datei.close();
}
```

Headerdatei für File-Streams (fstream)

Objekt datei vom Typ ostream erzeugen. Name der Datei wird im Konstruktor übergeben

Ausgabe von d1 an datei mit dem überladenen Operator <<

Schließen der Datei

1.1.2015

Inhalt der Datei datum.txt nach Programmablauf

## Einlesen von Daten

Zum Einlesen von Daten aus einer Datei wird ein `ifstream` verwendet. Um eine Datei zu öffnen und deren Inhalt auf dem Bildschirm auszugeben, wird so vorgegangen:

```
#include <fstream> ← Headerdatei für File-Streams (fstream)
using namespace std;

int main()
{
    ifstream datei( "datei.txt" ); ← Öffnen der Datei datei.txt zum Lesen

    if( !datei ) ← Prüfen ob die Datei erfolgreich geöffnet wurde. Bei dem
    {           Ausdruck handelt es sich nicht um die Überprüfung eines Zeigers,
        cout << "Fehler!\n";
        exit( 1 );
    }

    while( true )
    {
        char c;
        datei.get( c );
        if( datei.eof() ) ← Mit der Methode eof prüfen ob das Dateiende (end
                            of file) erreicht worden ist)

            break;
        cout.put( c ); ← Ausgabe des gelesenen Zeichens an cout
    }

    datei.close(); ← Schließen der Datei
}
```

## Der **this**-Pointer

Bei unserer bisherigen Arbeit mit Objekten hat es ausgereicht, dass innerhalb eines Objektes auf Attribute zugegriffen werden kann. Adressen von Objekten haben wir nur verwendet, wenn wir Objekte dynamisch erstellt oder mit Arrays gearbeitet haben. Wir können auch weiter wie in C die Adresse eines Objektes mit dem Adressoperator ermitteln:

```
datum d;  
datum *zeiger;  
  
zeiger = &d;
```

Für diese Operation müssen wir allerdings einen expliziten Zugriff auf das Objekt haben, im oben angegebenen Beispiel die Instanz in der Variable d.

Innerhalb der Member-Funktion eines Objektes befinden wir uns aber praktisch im „Inneren“ des Objektes. Hier haben wir keinen Namen, über den wir auf das Objekt zugreifen können.

Um aus einer Memberfunktion Zugriff auf die Instanz zu bekommen, auf der die Funktion ausgeführt wird, bietet C++ den sogenannten **this**-Pointer..

Eine einfache Verwendung liegt in der Ausgabe eines Objektes über den Ausgabeoperator mithilfe von **this**:

```
void datum::ausgabe ()  
{  
    cout << *this;  
}
```

Aufruf der Ausgabe mit einem dereferenzierten  
**this**-Pointer als Referenz

## Einrichten eines Querverweises mit dem this-Pointer

Eine weitere typische Verwendung des this-Pointers ist das Einrichten von Querverweisen wie unten gezeigt. Die Klasse Verwalter speichert einen Verweis auf die Instanz eines Objektes der Klasse Haus. Das Haus registriert sich dabei mit seinem this-Pointer.

```
class Verwalter
{
private:
    Haus* haus;

public:
    Verwalter() {}
    void registrieren( Haus* h ) { haus = h; }
};
```

Speichern der übergebenen Adresse der Instanz in dem privaten Datenfeld

```
class Haus
{
public:
    Haus( Verwalter* v) { v->registrieren( this ); }
};
```

Das Haus bekommt bei der Erzeugung einen Zeiger auf den Verwalter übergeben und registriert sich dort mit seiner Adresse

Erzeugung des Verwalters und des Hauses mit automatischer Registrierung beim Verwalter im Konstruktor der Hauses

```
int main()
{
    Verwalter ver;
    Haus villa( &ver );
}
```