

Kapitel 20

Das Zusammenspiel von Objekten

Modellierung von Beziehungen

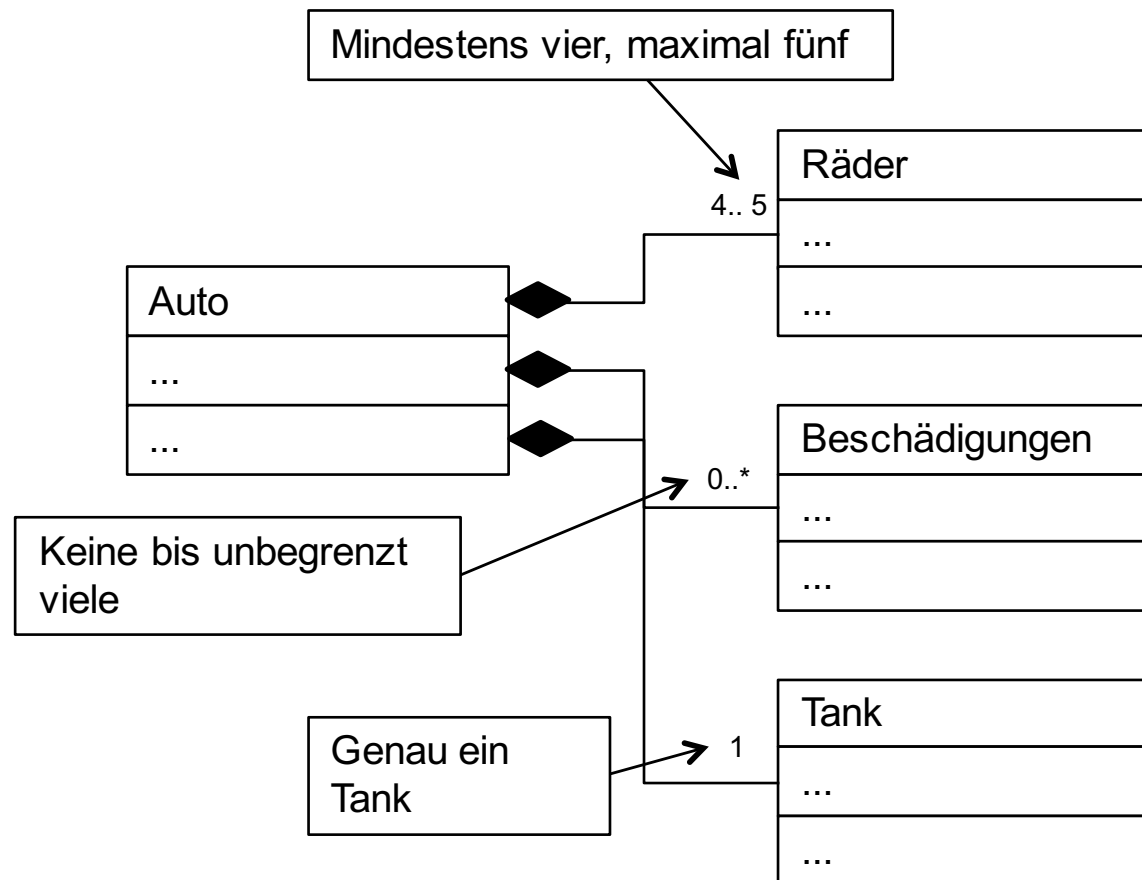
Wir haben mit Datenstrukturen bereits eine Hat-Ein beziehungsweise Ist-Teil-von Beziehung modelliert. Wir haben diese Modellierungsmöglichkeiten in der objektorientierten Modellierung natürlich weiterhin. Auch hier erweisen sich Objekte als Erweiterungen von Datenstrukturen und ersetzen sie streng genommen sogar.

Wir modellieren die Hat-Ein oder Ist-Teil-von Beziehung zwischen Objekten als Aggregation oder auch Komposition. Auf die feine Unterscheidung von Aggregation und Komposition wollen wir nicht eingehen und im weiteren von Komposition sprechen. In der Regel gehören die durch Komposition verbundenen Objekte der gleichen Begriffswelt an und haben die gleiche Lebensdauer.

Darstellung der Komposition

In UML wird für die Komposition die folgende Notation verwendet. Dabei kann zu den komponierten Objekten die Angabe einer Kardinalität hinzugefügt werden, die angibt, wie viele Teile einer bestimmten Klasse jeweils zur Gesamtheit gehören.

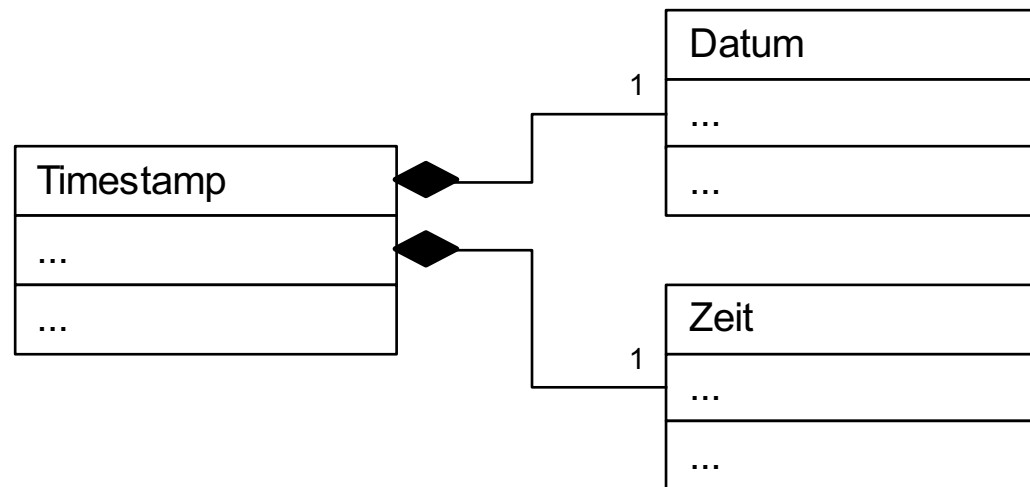
Als Beispiel wollen wir ein Auto modellieren, wie es beispielsweise für die Software eines Autovermieters verwendet werden könnte, um den Zustand des Fahrzeugs (Luftdruck, Beschädigungen, Tankinhalt) nach jeder Vermietung zu verwalten.



Komposition eigener Objekte

Wir wollen nun Objekte mit einer Hat-Ein Beziehung im Code zusammenführen. Dazu wollen wir einen Zeitstempel mit der Klasse `timestamp` modellieren. Ein Zeitstempel fasst in einem Objekt Datum und Zeit zusammen. Für das Datum haben wir bereits eine passende Klasse, die Klasse zur Speicherung der Zeit werden wir noch erstellen. Im weiteren Verlauf wird unsere Klasse `timestamp` dann Teil eines Logbucheintrages werden und dort den Zeitstempel entsprechender Einträge speichern.

Wir modellieren unsere Klasse `timestamp` so, dass sie genau ein Objekt `Datum` und genau ein Objekt `Zeit` hat:



Erstellen der Klasse zeit

Um die Klasse `timestamp` zu implementieren, fehlt uns also noch die Klasse `zeit`. Diese können wir nach dem Beispiel der bereits implementierten Klasse `datum` aber leicht erstellen:

```
class zeit
{
private:
    int stunde;
    int minute;
public:
    zeit( int st, int mi ) { set( st, mi ); }
    zeit() { set( 0, 0 ); }
    int getStunde() { return stunde; }
    int getMinute() { return minute; }
    void set( int st, int mi );
};
```

Konstruktor mit der set
Funktion implementiert

Parameterloser Konstruktor

Außerhalb implementierte
set Funktion

```
void zeit::set( int st, int mi )
{
    if(st < 0 || st > 23)
        st = 0;
    if(mi < 0 || mi > 59)
        mi = 0;

    minute = mi;
    stunde = st;
}
```

Ergänzung einer print-Methode

Wir erweitern unsere Klassen `datum` und `zeit` nun noch jeweils um eine `print`-Methode, die den Inhalt der Klasse formatiert ausgibt und die wir jeweils in der Klassendefinition umsetzen:

Vorher bereits
implementierte
Operatoren hier
nicht weiter
dargestellt

```
class datum
{
    private:
        int tag;
        int monat;
        int jahr;
    public:
        datum( int t, int m, int j ) { set( t, m, j ); }
        datum() { set( 1, 1, 1970 ); }
        void set( int t, int m, int j );
        void print() { printf( "%0.2d.%0.2d.%4d", tag, monat, jahr ); };

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

```
class zeit
{
    private:
        int stunde;
        int minute;
    public:
        zeit( int st, int mi ) { set( st, mi ); }
        zeit() { set( 0, 0 ); }
        int getStunde() { return stunde; }
        int getMinute() { return minute; }
        void set( int st, int mi );
        void print() { printf( "%0.2d:%0.2d", stunde, minute ); }
};
```

Einfache print
Methoden

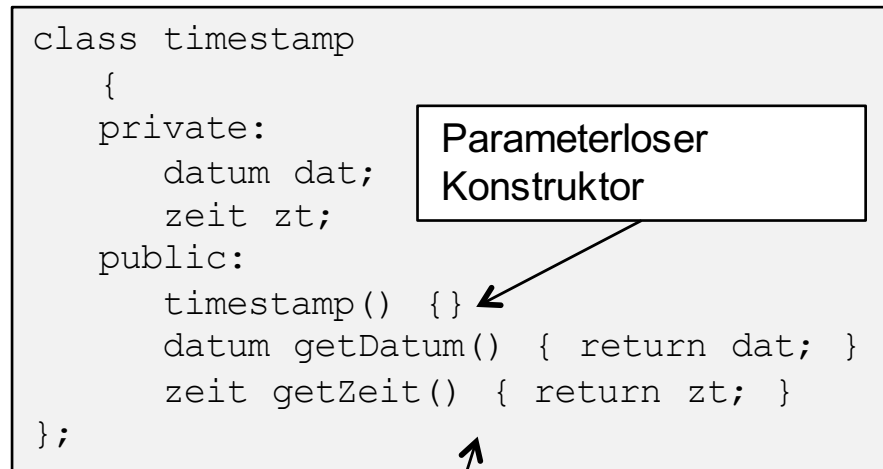
Definition der Klasse timestamp

Die Klassen die in `timestamp` zusammengeführt werden sollen, sind nun alle vorhanden. Wir können unsere neue Klasse zusammensetzen.

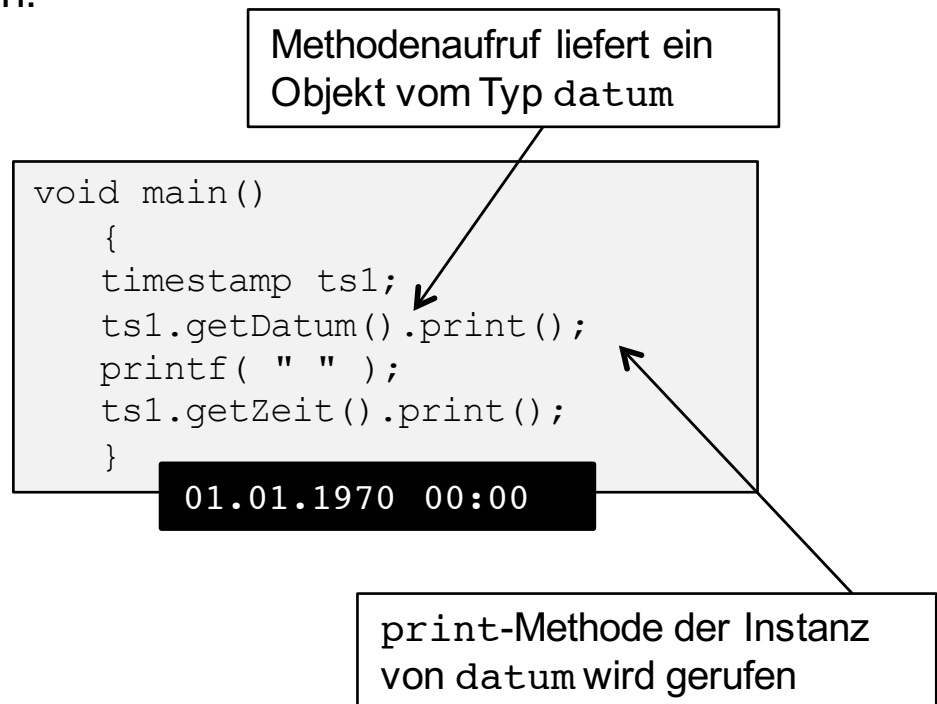
Wie bei `Datum` und `Zeit` sehen wir öffentliche Getter-Methoden vor, die Kopien der Attribute als Ergebnis an den Aufrufer zurückliefern.

Wir können die Klasse instanziiieren und verwenden:

```
class timestamp
{
private:
    datum dat;
    zeit zt;
public:
    timestamp() {}
    datum getDatum() { return dat; }
    zeit getZeit() { return zt; }
};
```



```
void main()
{
    timestamp ts1;
    ts1.getDatum().print();
    printf( " " );
    ts1.getZeit().print();
}
```

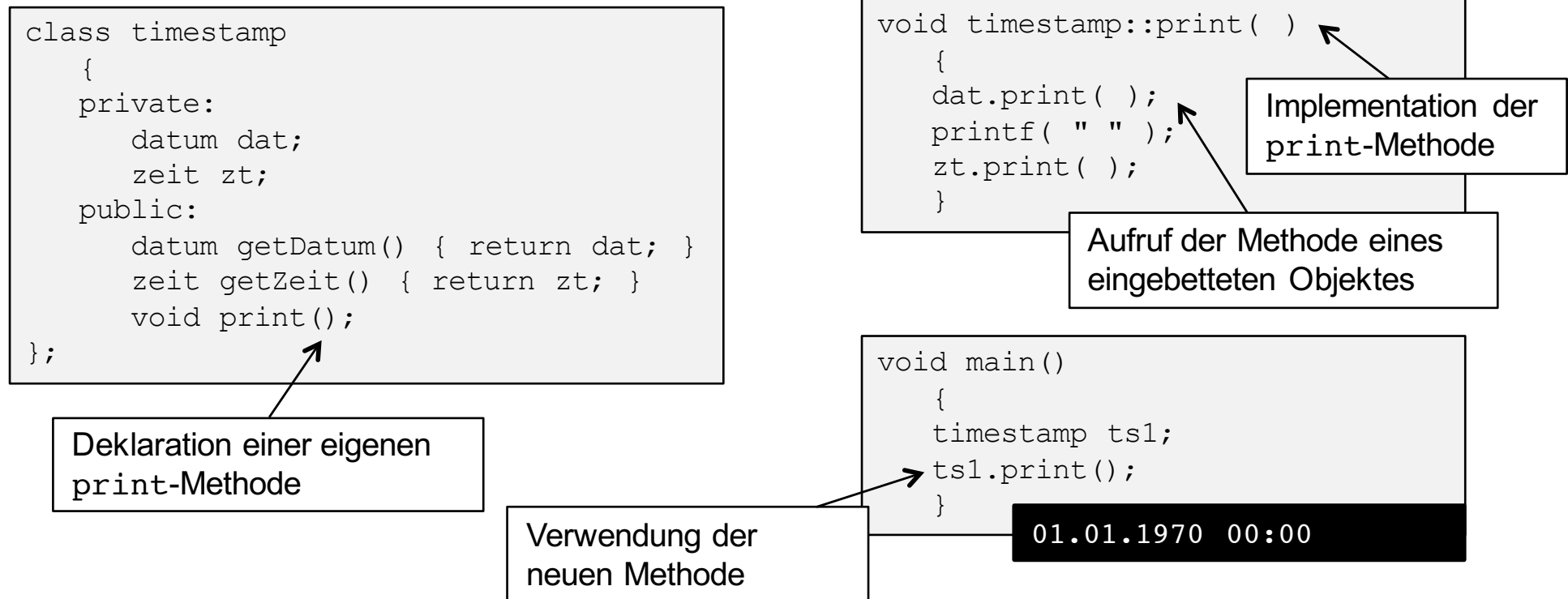


01.01.1970 00:00

Zur Ausgabe von `Datum` und `Zeit`, holen wir uns das eingebettete Objekt und rufen jeweils dessen `print`-Funktion. Der Zugriff erfolgt wie bei Datenstrukturen über den Punkt-Operator.

Implementierung der `print`-Methode

Wir erweitern unsere Klasse `timestamp` um eine eigene `print`-Methode. Hier profitieren wir bereits von der Funktionalität, die `datum` und `zeit` zur Verfügung stellen.



Wir rufen für die Objekte jeweils deren `print`-Methode auf und erhalten das gewünschte Ergebnis. Das Aufrufen von Methoden durch Objekte untereinander wird oft auch als „Senden von Nachrichten zwischen Objekten“ bezeichnet.

Die Konstruktion des eingebetteter Objekte

Objekte müssen immer in einem konsistenten Zustand sein. Ein Konstruktor eines Objektes instanziiert das Objekt in einem konsistenten Anfangszustand, der dann im weiteren Lebenszyklus der Instanz durch die Methoden konsistent verändert werden kann.

Durch die Konstruktoren werden dem Benutzer genau vorgegebene Optionen zur Verfügung gestellt, wie ein Objekt erstellt werden kann.

Wenn ein Objekt zur Instanziierung Zusatzinformationen von außen benötigt, dann stellt es entsprechende parametrisierte Konstruktoren zur Verfügung. Wenn keine solche Informationen notwendig sind, dann genügt ein Konstruktor mit der Vorgabe „keine Parameter“.

Wie wir bei der Klasse `timestamp` sehen, kann ein Objekt auch andere Objekte enthalten. Das Objekt wird damit Benutzer anderer Objekte. Unsere Klasse `timestamp` ist Benutzer der Klassen `zeit` und `datum`. Es muss sich damit auch an die vorhandenen Konstruktoren und deren Konstruktionsvorgaben halten und die Konstruktoren von `zeit` und `datum` mit den passenden Parametern rufen.

Der Default-Konstruktor von `timestamp`

Bisher haben wir den von System bereitgestellten parameterlosen Konstruktor für unsere Klasse genutzt. Wir wollen uns das Ergebnis noch einmal ansehen:

```
class timestamp
{
private:
    datum dat;
    zeit zt;
public:
    datum getDatum() { return dat; }
    zeit getZeit() { return zt; }
    void print();
};
```

datum und zeit sind mit ihren
eigenen parameterlosen
Konstruktoren initialisiert worden

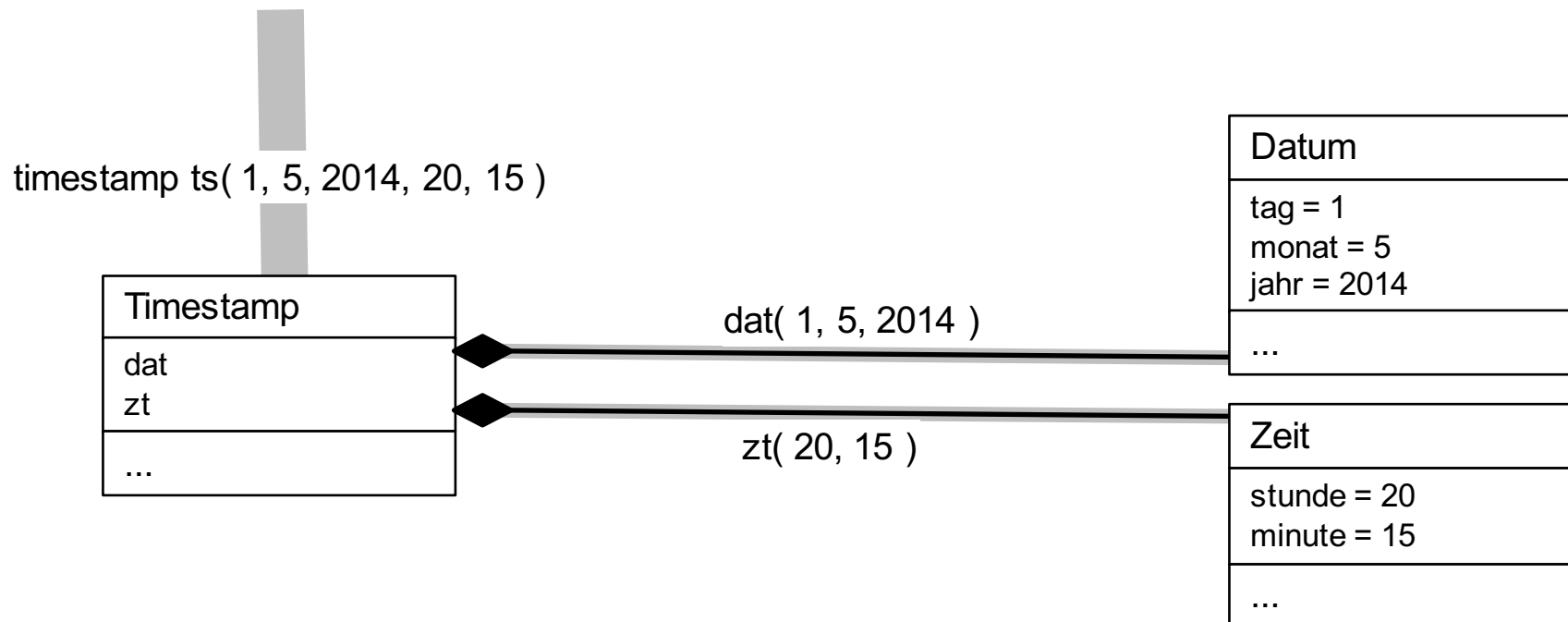
```
void main()
{
    timestamp ts1;
    ts1.print();
}
```

01.01.1970 00:00

Die eingebetteten Objekte `dat` und `zt` werden bei der Erstellung des `timestamp` Objektes jeweils mit ihrem parameterlosen Konstruktor instanziiert. Diese sind implizit für uns aufgerufen worden. Die eingebetteten Objekt sind mit ihnen korrekt initialisiert worden.

Ein parametrierter Konstruktor für die Klasse `timestamp`

Wir wollen dem Benutzer unserer Klasse `timestamp` auch einen parametrisierten Konstruktor anbieten, mit dem er ein Objekt der Klasse direkt mit festgelegtem Datum und Zeit initialisieren kann. Dazu müssen Konstruktionsparameter aus der Klasse `timestamp` an die eingelagerten Klassen `zeit` und `datum` weitergeleitet werden.



Weiterleitung von Parametern des Konstruktors

Wir haben im Konstruktor die Möglichkeit, Parameter des Konstruktors direkt an eingelagerte Objekte weiterzuleiten. Der entsprechende Aufruf ist von der Schnittstelle des Konstruktors durch einen Doppelpunkt getrennt. Die Identifikation des eingebetteten Objektes für das ein Konstruktor aufgerufen werden soll, erfolgt dabei durch seinen Member-Namen. Entsprechend der Parametersignatur wird dann ein geeigneter Konstruktor der Klasse aufgerufen. Der Aufruf erfolgt dabei vor dem Aufruf des Konstruktors der einlagernden Klasse. Innerhalb dieses Konstruktors steht das genutzte Objekt dann instanziiert zur Verfügung.

```
class timestamp
{
private:
    datum dat;
    zeit zt;
public:
    timestamp( int ta, int mo, int ja, int st, int mi );
    //...
};
```

Deklaration eines vollständig
parametrierten Konstruktors

Aufruf des Konstruktors für
das Objekt dat mit den
Parametern ta, mo und ja

Objekte dat und zt stehen initialisiert zur Verfügung

```
timestamp::timestamp( int ta, int mo, int ja, int st, int mi) : dat( ta, mo, ja),
{
    zt( st, mi)
}
```

Aufruf des Konstruktors für das Objekt
zt mit den Parametern st und mi

Initialisiererliste in der Klassendeklaration

Die Initialisiererliste kann auch in der Inline-Implementierung eines Konstruktors hinzugefügt werden:

```
class timestamp
{
private:
    datum dat;
    zeit zt;
public:
    timestamp( int ta, int mo, int ja) : dat( ta, mo, ja) {}
    // ...
};
```

Konstruktor mit Übergabe der
Datumsparameter und Aufruf des
Konstruktors für dat.

Leerer Rumpf des
Konstruktors

Im oben angegebenen Konstruktor werden die Parameter für das Datum direkt an das Objekt `dat` weitergeleitet, der Konstruktor für das Objekt `zt` wird nicht weiter spezifiziert, hier wird automatisch der parameterlose Konstruktor gerufen.

Einsatz der Klasse `timestamp`

Unter Verwendung der unterschiedlichen Konstruktoren können wir `timestamp` Objekte auf drei unterschiedliche Arten erstellen:

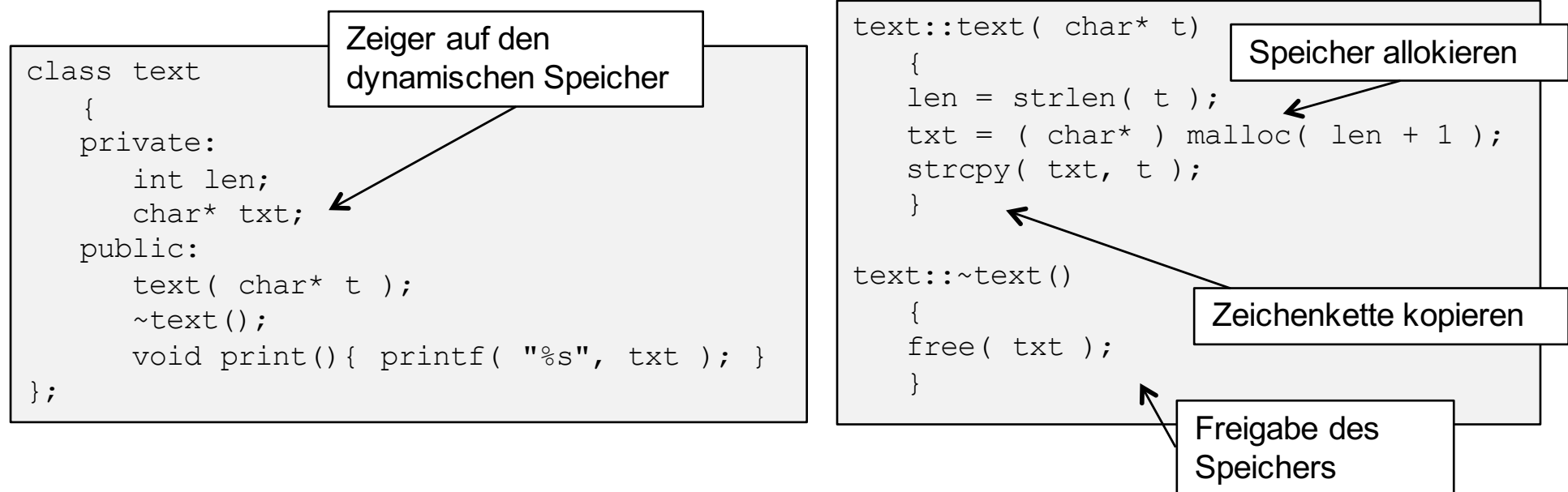
```
void main()
{
    timestamp ts1;
    timestamp ts2( 1, 5, 2014 );
    timestamp ts3( 1, 5, 2014, 20, 15 );
    ts1.print(); printf( "\n" );
    ts2.print(); printf( "\n" );
    ts3.print(); printf( "\n" );
}
```

```
01.01.1970 00:00
01.05.2014 00:00
01.05.2014 20:15
```

1. Parameterlos, hier werden für `datum` und `zeit` die parameterlosen Konstruktoren aufgerufen, die das Datum auf den 1.1.1970 und die Zeit auf 0:00 Uhr setzen
2. Unter Angabe von Tag, Monat und Jahr, hier werden die Konstruktorparameter an `datum` übergeben, `zeit` wird parameterlos auf 0:00 Uhr instanziiert
3. Unter Angabe von Tag, Monat, Jahr, Stunde und Minute, hier werden `datum` und `zeit` mit den übergebenen Parametern konstruiert

Eine Klasse `text` zur einfacheren Verwaltung von Texten

Wir wollen für unser Logbuch nun eine Klasse zur Verwaltung von Texten erstellen. Vorerst halten wir unsere Klasse so einfach wie möglich. In der Deklaration der Klasse sehen wir den Zeiger `txt` vor, über den wir den dynamisch allokierten Speicher verwalten. Zusätzlich wollen wir die Länge unseres Strings in `len` speichern.



Zur Umsetzung definieren wir einen Konstruktor, der eine Zeichenkette übergeben bekommt sowie einen Destruktor, der den allokierten Speicher wieder freigibt.

Der Konstruktor ermittelt die Größe der übergebenen Zeichenkette, allokiert Speicher und kopiert die Zeichenkette in den neu allokierten Bereich. Der Destruktor gibt den allokierten Speicher wieder frei, wenn ein Objekt der Klasse zerstört wird.

Ein erster Test unserer Klasse `text`

Bevor wir unsere Klasse an anderer Stelle verwenden, wollen wir einen kleinen Testrahmen aufbauen. Dazu erstellen wir in einem Programm neben unserer Funktion `main` eine weitere Funktion `text_test`, die bisher noch leer ist. Der Aufruf der leeren Funktion führt allerdings zum Absturz des Programms.

```
class text
{
private:
    int len;
    char* txt;
public:
    text( char* t );
    ~text();
    void print(){ printf( "%s", txt );}
};
```

```
text::text( char* t)
{
    len = strlen( t );
    txt = ( char* ) malloc( len + 1 );
    strcpy( txt, t );
}

text::~~text()
{
    free( txt );
}
```

```
void text_test( text x )
{
}

void main( )
{
    text t( "test1" );
    text_test ( t );
}
```

Der Programmcode führt zum Absturz, wo liegt das Problem?

Ergänzung einer Ausgabe zur Fehlersuche

Unsere Klasse besteht bisher praktisch nur aus Konstruktor und Destruktor sowie einem Hauptprogramm mit einer zusätzlichen Funktion. Wir erweitern Konstruktor und Destruktor um eine Ausgabe und sehen, dass der Destruktor zweimal aufgerufen wird.

```
class text
{
private:
    int len;
    char* txt;
public:
    text( char* t );
    ~text();
    void print(){ printf( "%s", txt );}
};
```

```
text::text( char* t)
{
    len = strlen( t );
    txt = ( char* ) malloc( len + 1 );
    strcpy( txt, t );
    cout << "Konstruktor fuer: " << txt << "\n";
}

text::~~text()
{
    cout << "Destruktor fuer: " << txt << "\n";
    free( txt );
}
```

Ausgabe in Konstruktor und Destruktor

```
void text_test( text x )
{
}
```

```
void main( )
{
    text t( "test1" );
    text_test ( t );
}
```

Konstruktor fuer: test1
Destruktor fuer: test1
Destruktor fuer: test1

Destruktor wird zweimal aufgerufen

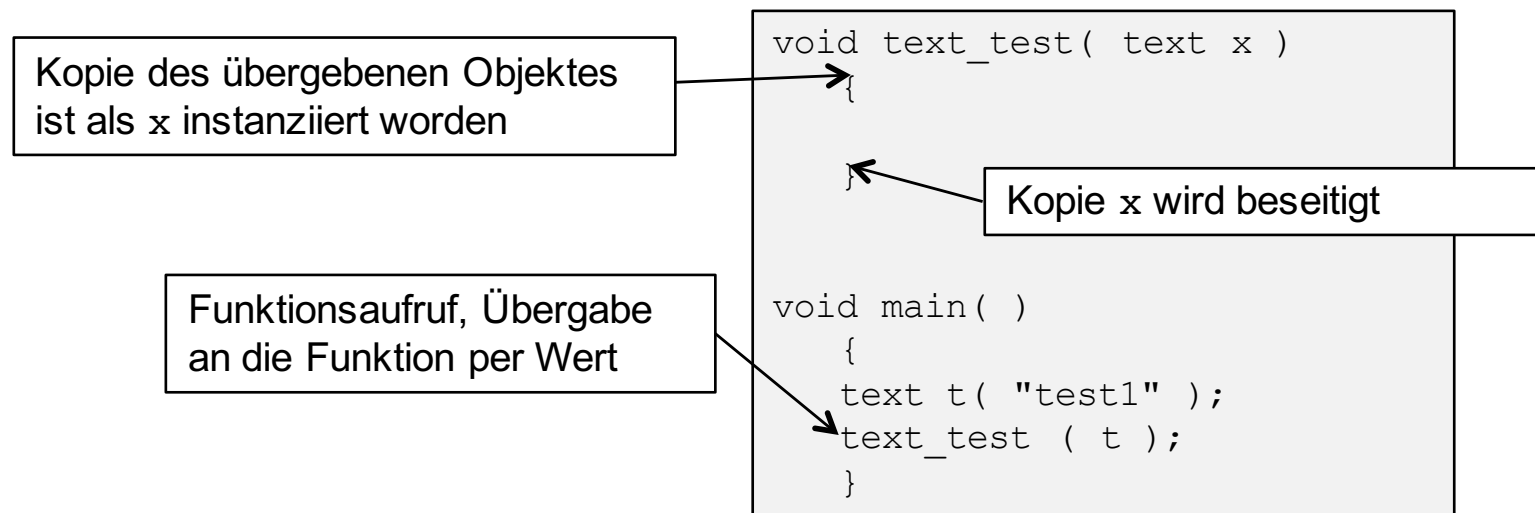
Analyse des Ergebnisses

Der Destruktor unserer Klasse wird zweimal durchlaufen, es werden also zwei Instanzen abgebaut. Hierfür ist der Aufruf der Funktion `text_test` mit dem Objekt als Parameter verantwortlich. Wir wissen bereits aus C, dass bei Aufruf einer Funktion die Parameter als Kopie an die Funktion übergeben werden:

Bei Aufruf einer Funktion werden die Parameter als Kopie an die Funktion übergeben

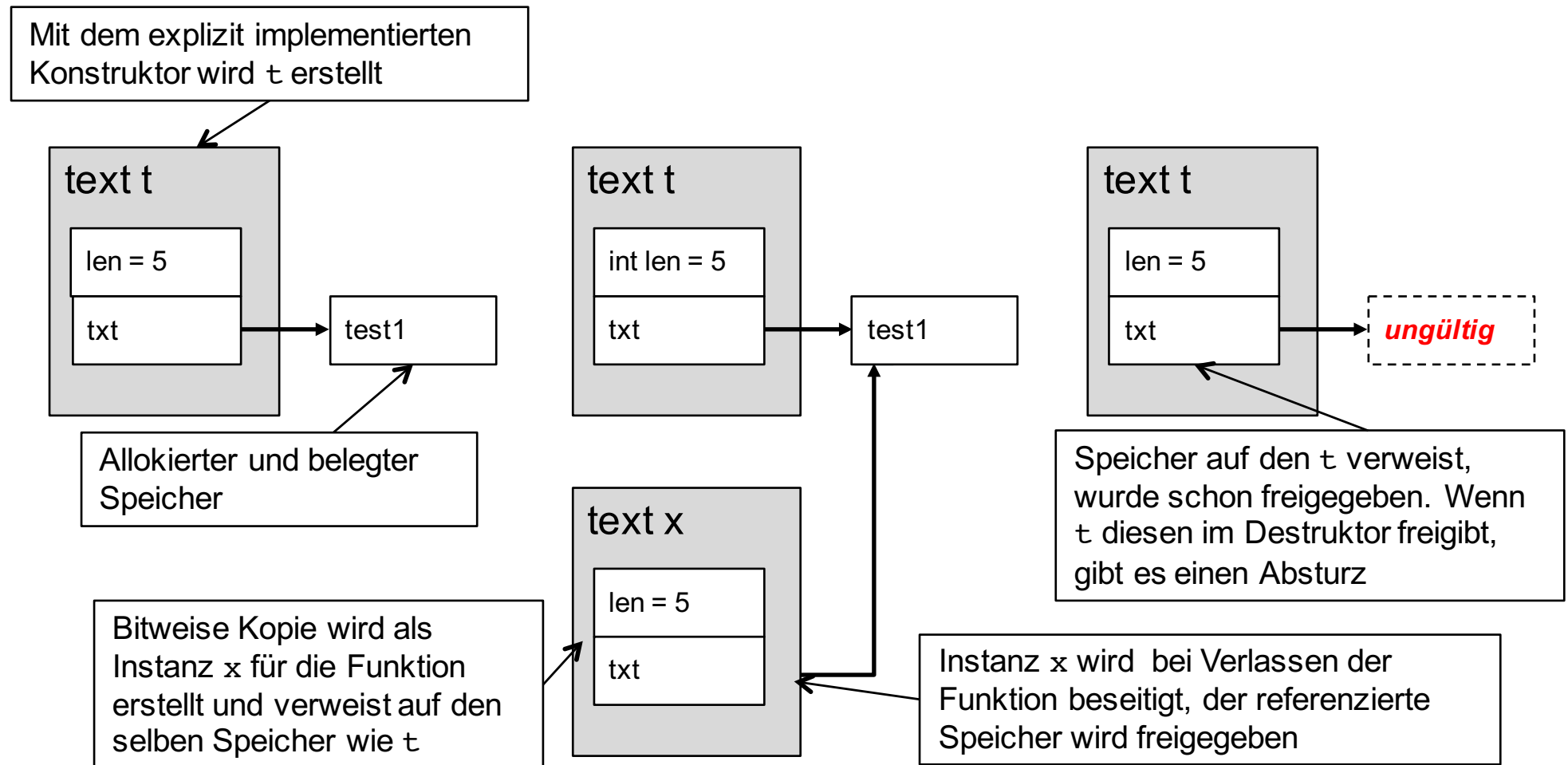
Damit entsteht implizit eine neue Instanz unserer Klasse als Kopie. Diese wird am Ende der Funktion beseitigt, wenn der Gültigkeitsbereich der Kopie endet.

Wird eine Kopie benötigt, erzeugt das Laufzeitsystem an der Schnittstelle das benötigte Duplikat. Es erzeugt dabei eine identische, bitweise Kopie aller Attribute.



Ablaufbeispiel mit automatisch erzeugter Kopie

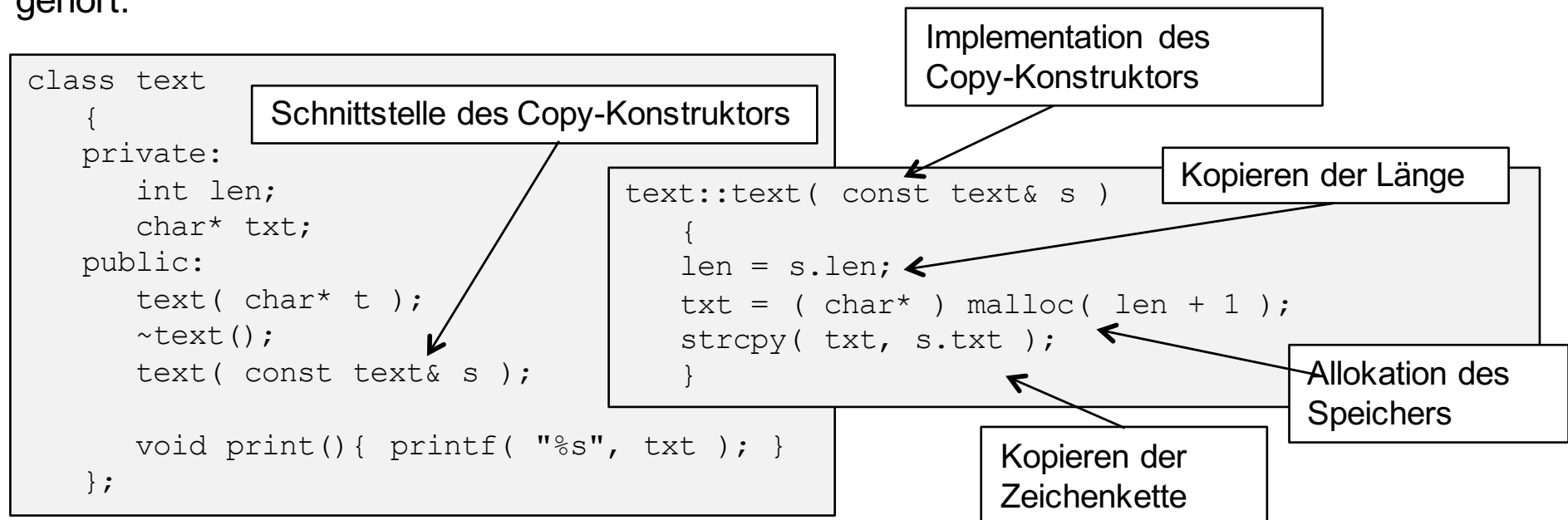
Die automatisch erzeugte bitweise Kopie aller Attribute erscheint als die Lösung, die wir benötigen. Bei genauerer Betrachtung sehen wir aber, warum dieses Verhalten hier ein Problem darstellt.



Erweiterung der Klasse um einen Copy-Konstruktor

Um das beschriebene Problem zu vermeiden, müssen wir der Klasse einen Copy-Konstruktor implementieren. Dieser muss dafür sorgen, dass eine korrekte Kopie unseres Objektes erzeugt wird, die auf „eigenen“ Speicher verweist.

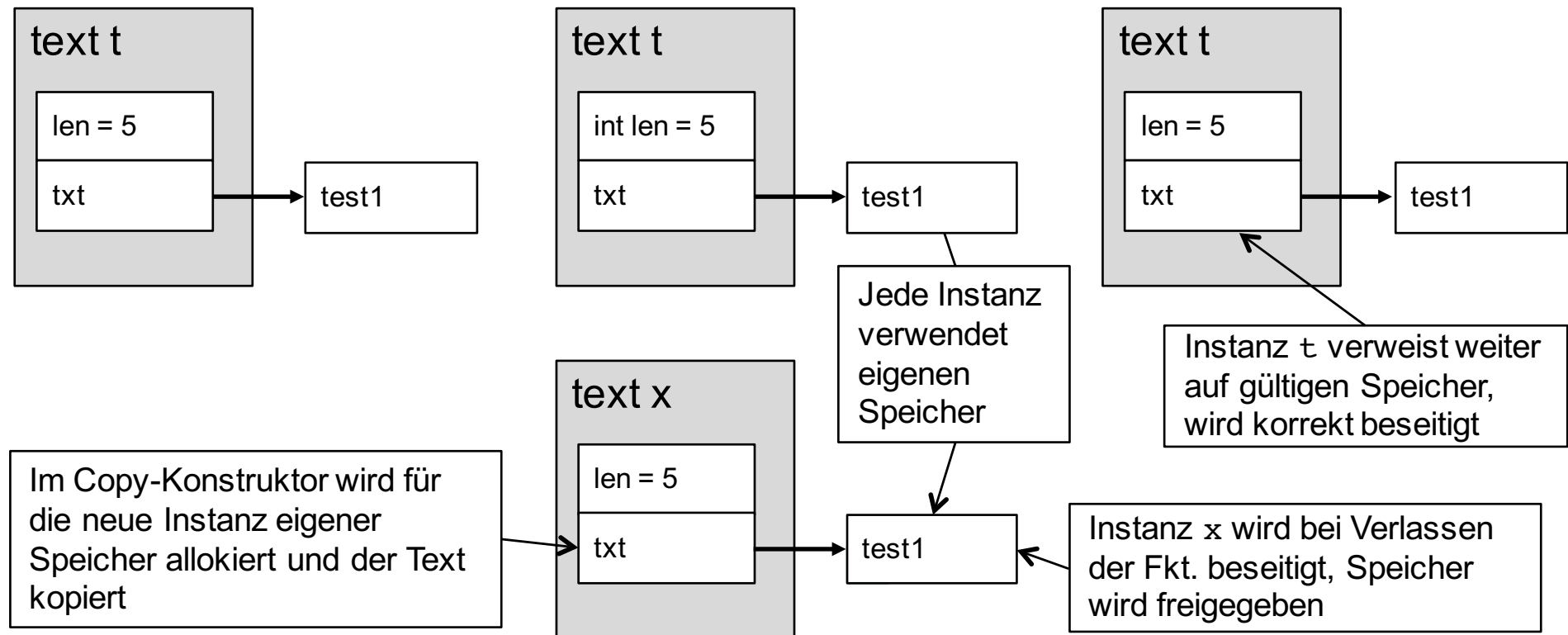
Der Copy-Konstruktor ist ein Konstruktor, trägt damit den Namen der Klasse und hat keinen Rückgabebetyp. Als Parameter erhält er eine konstante Referenz auf den Typ der Klasse, zu der er gehört.



Wir erweitern unsere Klasse um den Copy-Konstruktor, der für die neue Instanz eigenen Speicher allokiert und den Inhalt aus der kopierten Instanz überträgt. So können die Objekte unabhängig voneinander wieder freigegeben werden.


Ablaufbeispiel mit passend implementiertem Copy-Konstruktor

Mit dem implementierten Copy-Konstruktor wird eine unabhängige Kopie erzeugt, die eigenen Speicher verwaltet und freigibt. Damit ergibt sich nun der folgende korrekte Ablauf:



Zuweisung von Objekten

Es gibt eine weitere Situation, bei der durch automatisch bereitgestelltes Verhalten ein ähnliches Problem entstehen kann. Dies ist bei der Zuweisung an ein Objekt der Fall.

```
void main ()  
{  
    text u ( "test2" );  
    text v ( "test3" );  
  
    v = u;   
}
```

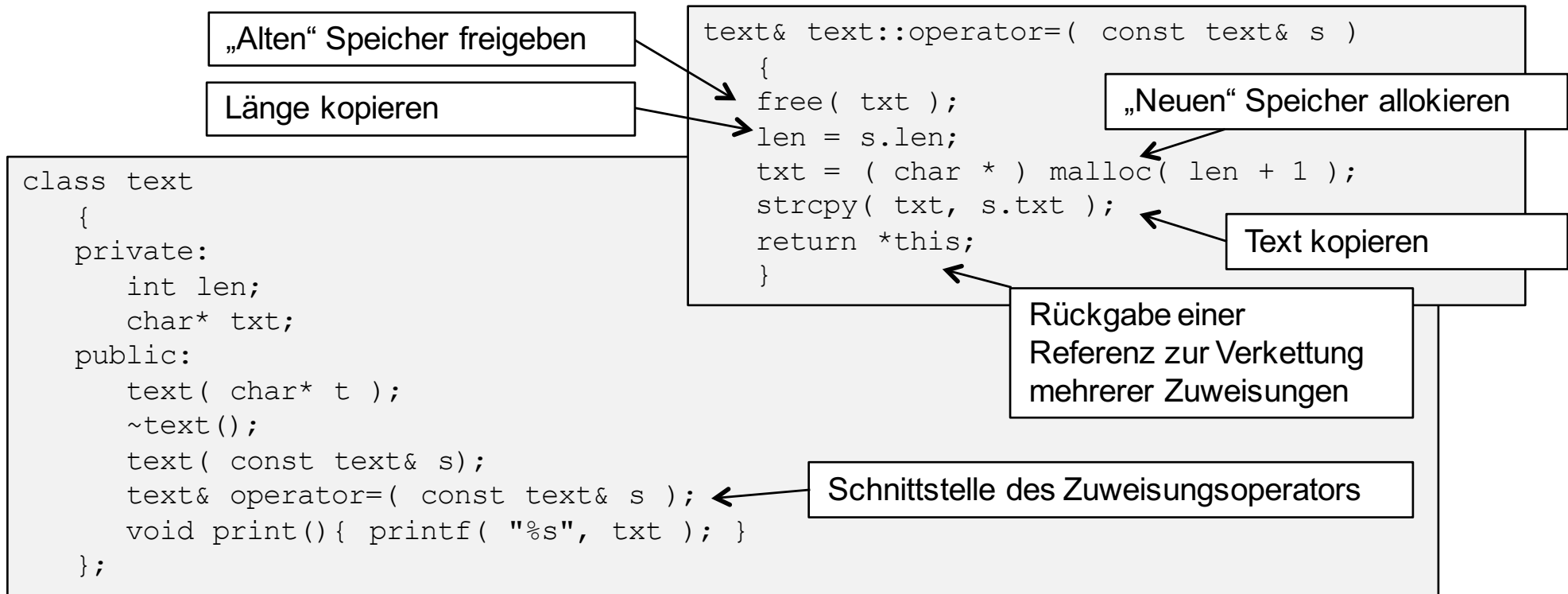
Zuweisung an
bestehendes Objekt

Die automatisch ausgeführte Zuweisung erzeugt ebenfalls eine exakte bitweise Kopie des zugewiesenen Objektes. Neben dem bereits bekannten Problem der doppelten Freigabe bei der Beseitigung der beiden Instanzen entsteht hier ein weiterer Fehler.

Für die Instanz `v` auf der linken Seite der Zuweisung ist bereits Speicher allokiert worden. Bei der Zuweisung wird der Zeiger auf diesen allokierten Speicher überschrieben. Dieser Speicher kann danach nicht mehr freigegeben werden, da seine Adresse jetzt nicht mehr bekannt ist.

Implementierung des Zuweisungsoperators

Um Abhilfe zu schaffen, werden wir auch den Zuweisungsoperator passend überladen. Dies geht wie bei anderen Operatoren auch. Er bekommt eine ähnliche Funktionalität wie der Copy-Konstruktor, arbeitet allerdings auf einer bereits existierende Instanz.



```
void main ()
{
    text u ( "test2" );
    text v ( "test3" );
    v = u;
}
```

Zuweisung an bestehende Objekte über den erstellten Zuweisungsoperator

Vorgehen für eigene Objekte

Wenn wir Klassen implementieren, für die durch eine einfache bitweisen Kopie kein in sich konsistentes Objekt erzeugt wird, sollte von Anfang an ein passender Copy-Konstruktor und ein entsprechender Zuweisungsoperator implementiert werden.

Dies gilt auch, wenn bei der aktuellen Verwendung der Klasse (noch) nicht kopiert oder zugewiesen wird und die gezeigten Probleme nicht auftreten. Nur wenn eine solche Klasse vollständig implementiert ist, kann sie später in einer anderen Umgebung ohne Probleme wiederverwendet werden.

Erweiterung unserer Klasse `text`

Unsere Klasse `text` ist nun voll funktionsfähig. Sie hat Konstruktor und Destruktor, die die Speicherverwaltung übernehmen. Mit dem explizit erstellten Copy-Konstruktor und dem passenden Zuweisungsoperator haben wir die Klasse auch für die übliche Verwendung in C++ gerüstet.

Bisher hat unsere Klasse aber noch keine Funktionalität, außer die, ihren Text auszugeben. Wir wollen zum Abschluss daher noch eine Methode hinzufügen, die mit dem Text in der Klasse arbeitet. Dazu wollen wir innerhalb unseres gespeicherten Textes suchen. Die Methode `find` soll eine Zeichenkette als Parameter übergeben bekommen und die Position des ersten Auftretens dieser Zeichenkette in unserem Text zurückgeben.

Ergänzen der Methode `find`

Wir fügen die Deklaration der Methode unserer Klasse hinzu und implementieren die gewünschte Funktionalität.

```
class text
{
private:
    int len;
    char* txt;
public:
    text( char* t );
    ~text();
    text( const text& s );
    text& operator=( const text& s );
    void print(){ printf( "%s", txt ); }
    int find( char* f );
};
```

```
int text::find( char* f )
{
    char* pos = strstr( txt, f );

    if(!pos)
        return -1;
    else
        return (pos - txt);
}
```

Der Rückgabewert der Funktion entspricht der Position der ersten gefundenen Übereinstimmung mit der gesuchten Zeichenkette. Wird die Zeichenkette nicht gefunden, dann liefert die `find` Methode `-1` als Ergebnis.

Die verwendete Funktion `strstr` der C-Laufzeitbibliothek gibt einen Zeiger auf den Anfang des gefundenen Textes zurück. Wenn der gesuchte Text nicht gefunden wurde, ist das Ergebnis ein Nullzeiger.

Verwendung der Klasse `text`

Wir können unsere Klasse nun komplett verwenden:

```
void main ()
{
    text t1( "Ein Auto" );
    text t2( "Ein Cabrio" );
    text t3( t2 );
    t3 = t1;

    t3.print( );
    printf( "\n" );

    int pos = t3.find( "Auto" );

    printf( "'Auto' an Pos: %d\n", pos );
}
```

Zuweisung an bestehendes
Objekt über den erstellten
Zuweisungsoperator

```
Ein Auto
'Auto' an Pos: 4
```

Beispiele

Wir haben nun die Grundlagen der objektorientierten Entwicklung kennengelernt. Die besonderen Vorteile der Objektorientierung kommen zum Tragen, wenn größere Programme entstehen und sich Objekte wiederverwendet werden können. Wir bearbeiten nun zwei Beispiele, in denen die gelernten Vorgehensweisen angewendet werden und die von der Wiederverwendung profitieren.

Der Datentyp „Menge“

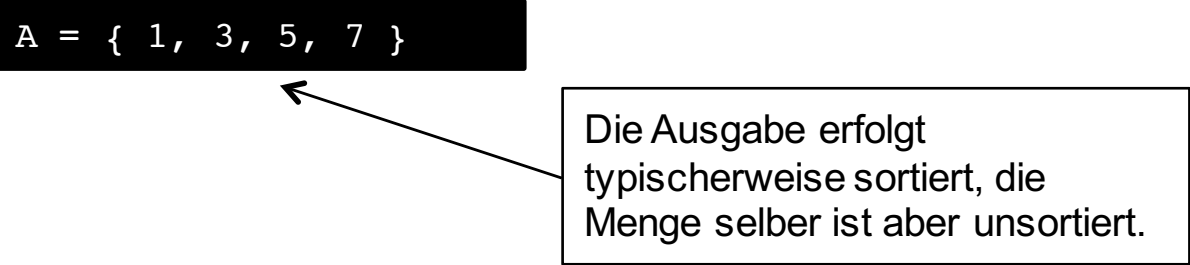
Mit dem Datentyp Menge wollen wir einen Datentyp implementieren, den es in vielen Programmiersprachen bereits als Grunddatentyp gibt. Um eine sinnvolle Implementierung zu ermöglichen, wollen wir zuerst die Anforderungen an unseren Datentyp zusammentragen.

Allgemein ist eine Menge eine Sammlung von Elementen eines bestimmten Datentyps, bei der es nicht auf die Reihenfolge ankommt. Jedes Element der Menge kann maximal einmal vorkommen, ist also entweder nicht oder einmal enthalten.

Für unser Beispiel wollen wir eine Menge implementieren, die Zahlen von 0 bis 255 aufnehmen kann. Unsere Menge soll dabei die wesentlichen aus der Mengenlehre bekannten Operationen ermöglichen:

Eine Menge A die die Zahlen 1, 3, 5 und 7 enthält wollen wir folgendermaßen darstellen:

$A = \{ 1, 3, 5, 7 \}$



Die Ausgabe erfolgt typischerweise sortiert, die Menge selber ist aber unsortiert.

Einige Mengenoperationen in der Wiederholung

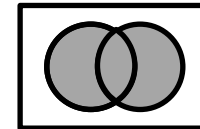
Zur kurzen Wiederholung der Basisoperationen gehen wir davon aus, dass unsere Menge nur die Zahlen von 0 bis 7 enthalten kann und wir die beiden Mengen A und B gegeben haben:

$$A = \{ 1, 2, 4 \}$$

$$B = \{ 1, 4, 6, 7 \}$$

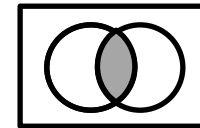
Vereinigen wir die Mengen A und B und wählen den Operator '+' als Operator für die Vereinigung, also die Zusammenfassung der beiden Mengen dann erhalten wir:

$$C = A + B = \{ 1, 2, 4, 6, 7 \}$$



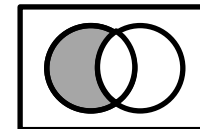
Zum Durchschnitt zweier Mengen auch deren Schnittmenge genannt, gehören alle Elemente, die in beiden Mengen vorhanden sind. Mit dem Operator '*' für den Durchschnitt erhalten wir:

$$C = A * B = \{ 1, 4 \}$$



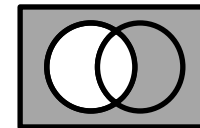
Die Differenzmenge $A - B$ ist die Menge aller Elemente, die zu A, nicht aber zu B gehören. Mit dem Operator '-' ergibt das als Ergebnis:

$$C = A - B = \{ 2 \}$$



Das Komplement $\sim B$ ist die Menge aller Elemente, die in der Grundmenge, aber nicht in B sind:

$$C = \sim B = \{ 0, 2, 3, 5 \}$$



Operatoren die eine neue Menge bilden

Wir wollen in unserer Klasse die folgenden Operatoren implementieren, die jeweils als Ergebnis eine neue Menge erzeugen:

Operation	Beschreibung
$A + B$	Erzeugt die Vereinigung der Mengen A und B
$A * B$	Erzeugt den Durchschnitt der Mengen A und B
$A - B$	Erzeugt die mengentheoretische Differenz „ A ohne B “
$\sim A$	Erzeugt das Komplement der Menge A
$A + e$	Erzeugt die Menge, die alle Elemente aus A und zusätzlich das Element e enthält
$A - e$	Erzeugt die Menge, die alle Elemente aus A , aber nicht das Element e enthält

Verändernde Operatoren

Es wird auch Operatoren geben, die eine bestehende Menge verändern.

Operation	Beschreibung
$A \ += \ B$	Fügt die Elemente aus B zur Menge A hinzu
$A \ *= \ B$	Entfernt aus A alle Elemente, die nicht zu B gehören
$A \ -= \ B$	Entfernt aus A alle Elemente, die zu B gehören
$A \ += \ e$	Fügt das Element e der Menge A hinzu
$A \ -= \ e$	Entfernt das Element e aus der Menge A

Prüfende Operatoren

Zusätzlich gibt es Operatoren, die eine bestehende Menge prüfen und ein entsprechendes Ergebnis zurückliefern:

Operation	Beschreibung
$A \leq B$	Prüft, ob A Teilmenge von B ist. Das Ergebnis ist 1, wenn die Teilmengenbeziehung besteht, ansonsten 0
$!A$	Prüft, ob die Menge A leer ist. Bei einer leeren Menge ist das Ergebnis 1, ansonsten 0
$e < A$	Prüft, ob die Menge A das Element e enthält. Kommt e in A vor, so ist das Ergebnis 1, andernfalls 0

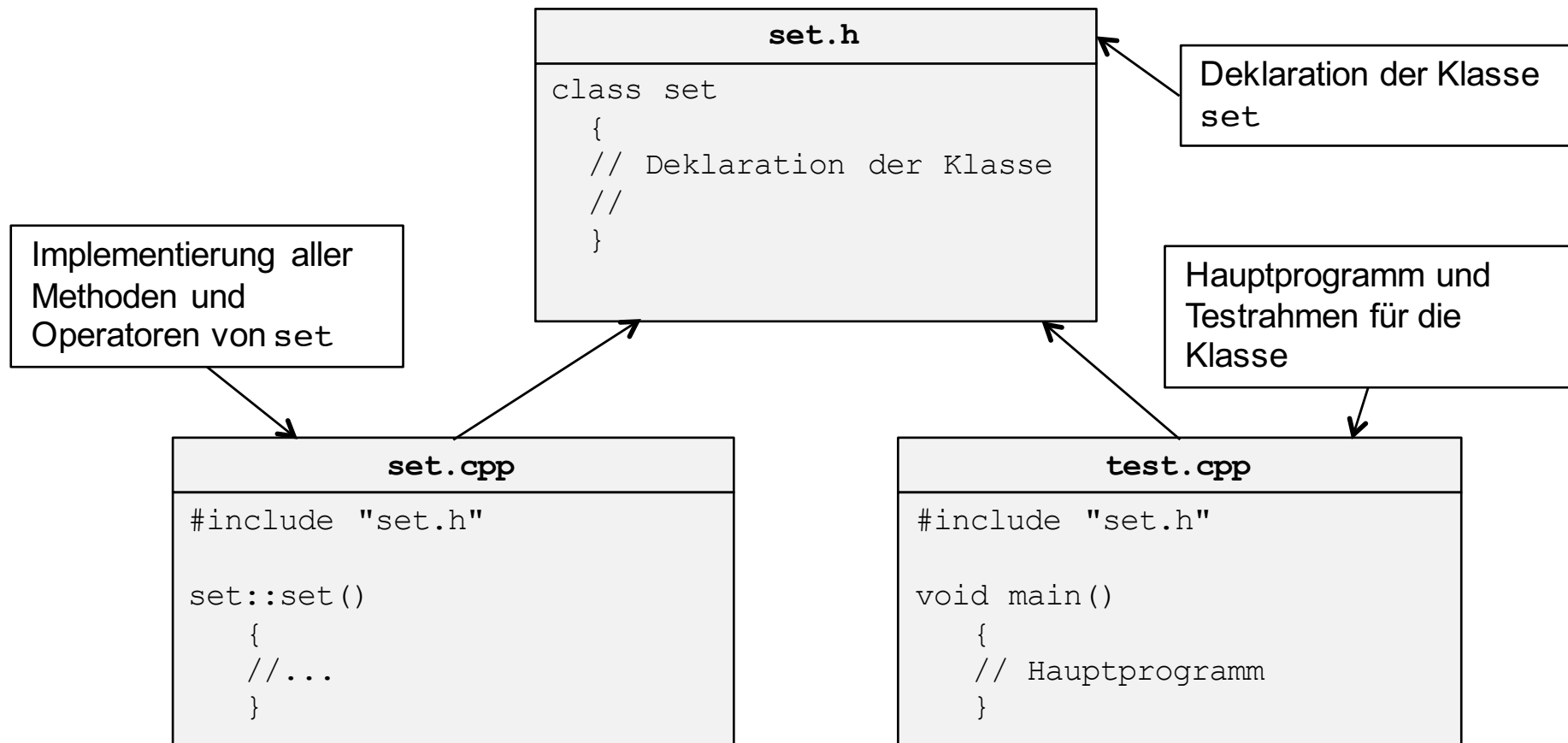
Ausgabeoperator

Abschließend gibt es einen Operator, mit dessen Hilfe wir eine Menge in einen Output-Stream wie `cout` ausgeben können:

Operation	Beschreibung
<code>os << A</code>	Gibt die Menge <code>A</code> auf dem ostream <code>os</code> aus.

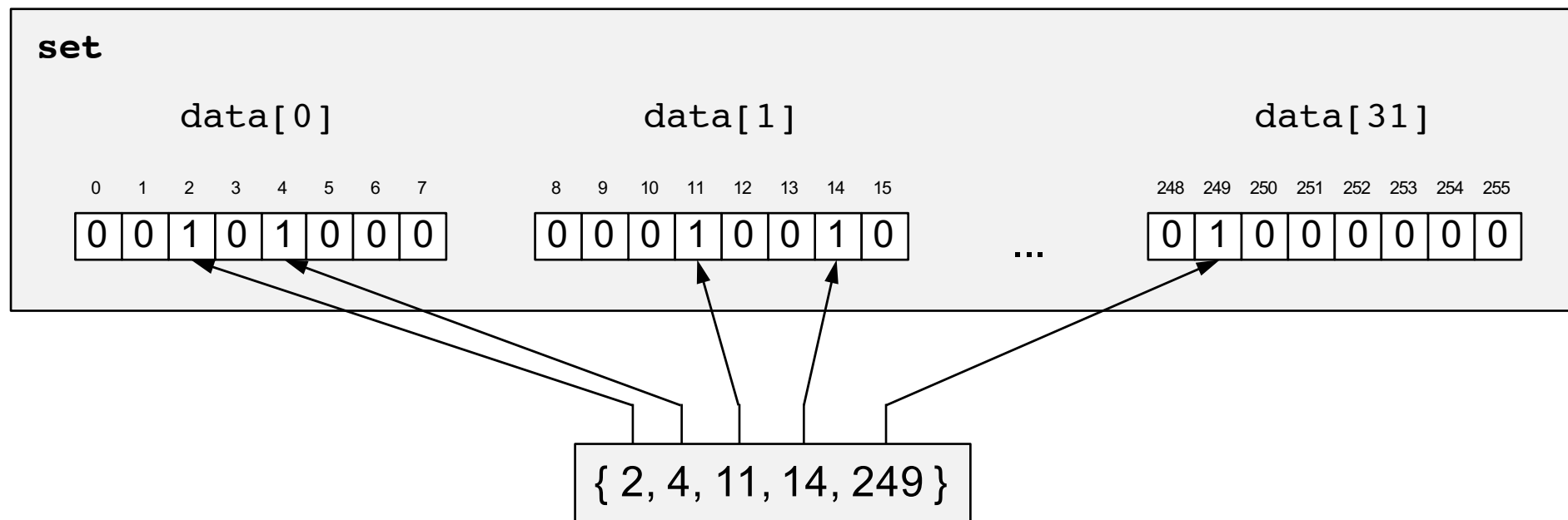
Aufteilung des Codes

Wir werden unsere Klasse für die Menge **set** nennen. Da wir die Klasse später wiederverwenden wollen, achten wir auf eine saubere Aufteilung unseres Codes und erstellen drei Dateien:



Interne Darstellung der Menge

Wir wollen unsere Menge intern als ein Array vorzeichenloser Zeichen (`unsigned char`) speichern. Jeweils nur ein einzelnes Bit an der entsprechenden Bitposition soll anzeigen, ob das Element in der Menge vorhanden ist oder nicht. Um die Zahlen von 0 bis 255 als Elemente unserer Menge zu verwalten, genügt damit ein Array mit 32 Zeichen:



```
class set
{
private:
    unsigned char data[32];
};
```

Die Deklaration unserer Klasse set

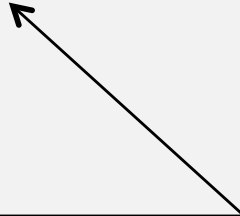
Wir werden alle Operatoren als `friend` Funktionen implementieren. Die entsprechenden Deklarationen nehmen wir in die Klasse mit auf.

Die Operatoren die eine neue Menge erzeugen, haben als Rückgabetyp jeweils wieder ein Objekt des Datentyps `set`.

Wir übergeben die Parameter dieser einzelnen Operatoren, jeweils als konstante Referenzen. So kann der Nutzer der Klasse mit einem Blick in die Klassendeklaration erkennen, dass die Operatoren die per Referenz übergebenen Operanden nicht manipulieren.

```
class set
{
    friend set operator+( const set& s1, const set& s2 );
    friend set operator-( const set& s1, const set& s2 );
    friend set operator*( const set& s1, const set& s2 );
    friend set operator~( const set& s );
    friend set operator+( const set& s, const int e );
    friend set operator-( const set& s, const int e );

    ///...
};
```



Deklaration der Operatoren
die ein neues `set` erzeugen

Fortsetzung der Klassendeklaration

Die Operatoren die den linken Operanden verändern, wie beispielsweise der Operator '+=', geben als Ergebnis eine Referenz auf ein Objekt der Klasse `set` zurück. Hier wird nur der zweite Operand als konstante Referenz übergeben, da wir den ersten Operanden ja ausdrücklich verändern wollen.

Die prüfenden Operatoren geben jeweils einen Wert vom Typ `int` als Prüfergebnis zurück. Auch hier sollen die Operanden nicht verändert werden und werden entweder als Kopie oder als konstante Referenz übergeben.

```
class set
{
    // .. Erster Teil der Klassendeklaration
    friend set &operator+=( set& s1, const set& s2 );
    friend set &operator-=( set& s1, const set& s2 );
    friend set &operator*=( set& s1, const set& s2 );
    friend set &operator+=( set& s, const int e );
    friend set &operator-=( set& s, const int e );

    friend int operator<= ( const set& s1, const set& s2 );
    friend int operator< ( int e, const set& s2 );
    friend int operator! ( const set& s );
    friend ostream& operator<< ( ostream& os, const set& s );

private:
    unsigned char data[32];
public:
    set();
};
```

Operatoren die den linken
Operanden verändern

Prüfende Operatoren

Ausgabeoperator

Array zur Speicherung der Daten


Konstruktor

Der Konstruktor der Klasse set

Im Konstruktor unserer Klasse müssen wir dafür sorgen, dass eine neu instanziierte Menge der leeren Menge entspricht, also alle Bits des zur Datenspeicherung verwendeten Arrays auf 0 gesetzt sind.

```
set::set()  
{  
    int i;  
    for( i = 0; i < 32; i++ )  
        data[i] = 0;  
}
```

Alle Elemente des Arrays
werden auf 0 gesetzt



Implementierung des Operators +=

Wir implementieren nun unseren ersten Operator und wählen dazu den Operator '+' der die Elemente aus S2 zur Menge S1 hinzufügt:

0	1	2	3	4	5	6	7
0	1	1	0	1	0	0	0

S1 = { 1, 2, 4 }

0	1	0	0	1	0	1	1
---	---	---	---	---	---	---	---

S2 = { 1, 4, 6, 7 }

0	1	1	0	1	0	1	1
---	---	---	---	---	---	---	---

S1 += S2 // { 1, 2, 4, 6, 7 }

Dazu muss unsere Operator in dem Array von s1 zusätzlich zu den dort bereits gesetzten Bits die setzen, die im Array von s2 gesetzt sind. Dies erreichen wir mit einem bitweisen Oder-Operator:

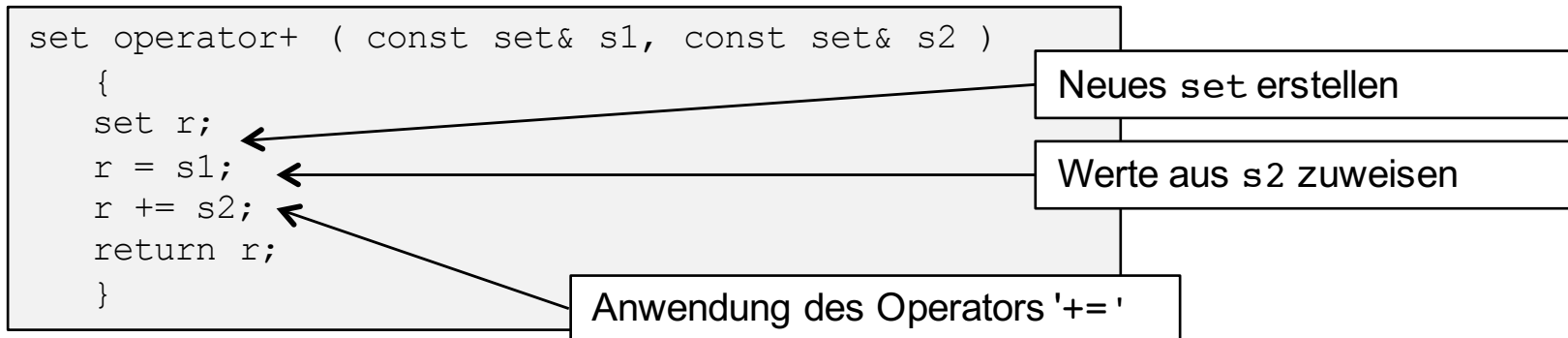
```
set& operator+= ( set& s1, const set& s2 )  
{  
    int i;  
    for( i = 0; i < 32; i++ )  
        s1.data[i] |= s2.data[i];  
  
    return s1;  
}
```

Bitweise Oder-Verknüpfung

Vereinigung zweier Mengen

Wir haben als ersten Operator den Operator '+' implementiert. Oft kann man weitere Operatoren mit bereits implementierten leicht umsetzen. So auch hier, wir werden den Operator '+' jetzt verwenden, um die Vereinigung von Mengen einfach zu implementieren.

Die Vereinigung zweier Mengen erzeugt als Ergebnis eine neue Menge. Innerhalb unseres Operators erzeugen wir daher zuerst die neue Menge, die wir nachher als Ergebnis zurückgeben werden und weisen ihr die Werte des linken Operanden zu. Danach besteht der Rest der Implementierung nur noch aus der Anwendung des bereits umgesetzten Operators '+' und der Rückgabe des Ergebnisses.



Die Operatoren -= und -

Die Operation $s1 - s2$ entfernt aus $s1$ alle Elemente, die zu $s2$ gehören. Die Implementierung der Operationen ' $-=$ ' und ' $-$ ' läuft damit prinzipiell genauso ab, wie bei den beiden vorherigen Operatoren, mit dem Unterschied, dass bei der Differenzbildung die Bits die im Operanden $s2$ gesetzt sind in dem linken Operanden $s1$ gelöscht werden müssen:

0	1	2	3	4	5	6	7	
0	1	1	0	1	0	0	0	$s1 = \{ 1, 2, 4 \}$
0	1	0	0	1	0	1	1	$s2 = \{ 1, 4, 6, 7 \}$
0	1	1	0	0	0	0	0	$s1 -= s2 \quad // \quad \{ 1, 2 \}$

Bitweise Und-Verknüpfung von $s1$ mit dem Komplement von $s2$

```
set& operator-= ( set& s1, const set& s2 )  
{  
    int i;  
    for( i = 0; i < 32; i++ )  
        → s1.data[i] &= ~s2.data[i];  
    return s1;  
}
```

```
set operator- ( const set& s1, const set& s2 )  
{  
    set r;  
    r = s1;  
    r -= s2; ←  
    return r;  
}
```

Verwendung des
Operators ' $-=$ '
zur Implementierung des
Operators ' $-$ '

Durchschnitt zweier Mengen

Der Durchschnitt zweier Mengen wird gebildet, indem eine bitweise Und-Verknüpfung der beiden Datenarrays durchgeführt wird, so dass nur die Bits gesetzt bleiben, die in beiden Mengen gesetzt sind:

```
set& operator*= ( set& s1, const set& s2 )
{
    int i;
    for( i = 0; i < 32; i++ )
        s1.data[i] &= s2.data[i];
    return s1;
}
```

Bitweise Und-Verknüpfung



```
set operator* ( const set& s1, const set& s2 )
{
    set r;
    r = s1;
    r *= s2;
    return r;
}
```

Das Komplement einer Menge

Um das Komplement einer Menge zu bilden, müssen wir alle Bits in dem Datenarray invertieren. Auch dies ist mit den uns bereits aus C bekannten Bit-Operationen kein Problem:

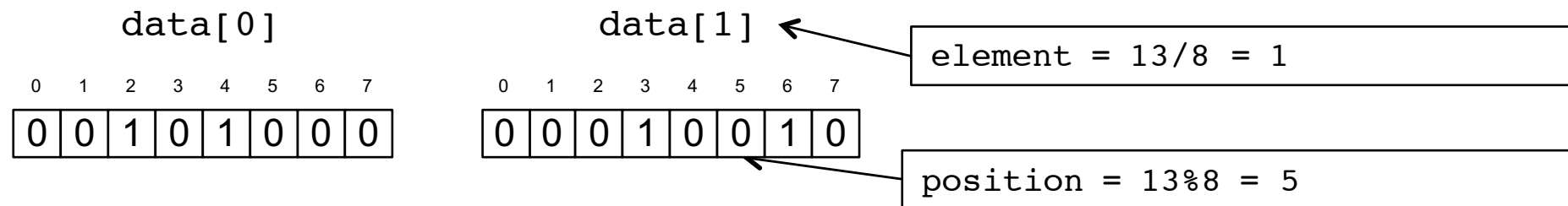
```
set operator~ ( const set& s )  
{  
    int i;  
    set r;  
    for( i = 0; i < 32; i++ )  
        r.data[i] = ~s.data[i];  
  
    return r;  
}
```

Erstellung eines neuen set

Bitweises Komplement alle Array-
Elemente in das neue set

Hinzufügen eines Elementes zu einer Menge

Um ein Element zu einer Menge hinzuzufügen, müssen wir das entsprechende Bit im Array lokalisieren und dieses Bit gezielt setzen. Das Bit für die Position e steht im Array Element $e/8$ und dort an der Position $e\%8$. Mit diesen Informationen können wir das Bit gezielt manipulieren. Am Beispiel von $e = 13$ ergibt sich:



```
set& operator+= ( set& s1, const int e )  
{  
    if( ( e >= 0 ) && ( e < 256 ) )  
        s1.data[e/8] |= ( 1 << ( e%8 ) );  
    return s1;  
}
```

Einzelnes Bit setzen

Mit bitweisem Oder verknüpfen

Array-Element ermitteln

```
set operator+ ( const set& s1, const int e )  
{  
    set r;  
    r = s1;  
  
    r += e;  
    return r;  
}
```

Umsetzung des Operators '+' mit
neuem set

Entfernen von Elementen aus einer Menge

Nach den Ausführungen zum Hinzufügen von Elementen zu einer Menge, ist das Entfernen nun ebenfalls leicht umgesetzt:

```
set& operator-= ( set& s1, const int e )  
{  
    if( ( e >= 0 ) && ( e < 256 ) )  
        s1.data[e/8] &= ~( 1 << ( e%8 ) );  
    return s1;  
}
```

Mit bitweisem Und verknüpfen

```
set operator- ( const set& s1, const int e )  
{  
    set r;  
    r = s1;  
  
    r -= e;  
    return r;  
}
```

Array-Element ermitteln

Operator für die Teilmengenprüfung

Wir können nun die vergleichenden Operatoren umsetzen. Die Operation $A \leq B$ prüft, ob A Teilmenge von B ist.

Zur Überprüfung der Teilmengenbeziehung muss getestet werden, ob mindestens die Bits aus der einen Menge auch in der anderen gesetzt sind:

```
int operator<= ( const set& s1, const set& s2 )
{
    int i;
    for( i = 0; i < 32; i++ )
    {
        if( ( s1.data[i] & s2.data[i] ) != s1.data[i] )
            return 0;
    }
    return 1;
}
```

Prüfung aller Array-Elemente

Werden Bits im Array von s1 gefunden, die im Array von s2 nicht gesetzt sind, besteht keine Teilmengenbeziehung, Prüfung wird abgebrochen

Wenn kein vorzeitiger Abbruch erfolgt ist, besteht eine Teilmengenbeziehung

Operator für die Prüfung eines enthaltenen Elements

Um zu prüfen, ob ein bestimmtes Element einer Menge vorhanden ist, müssen wir prüfen, ob das entsprechende Bit gesetzt ist. Die Prüfung hat starke Ähnlichkeit mit dem Hinzufügen oder Entfernen eines bestimmten Elementes.

```
int operator<( int e, const set& s )
{
    if( ( e >= 0 ) && ( e < 256 ) )
        return s.data[e/8] & ( 1 << ( e%8 ) );
    return 0;
}
```

Einzelnes Bit setzen

Mit bitweisem Und verknüpfen

Das Ergebnis des bitweisen Vergleiches
ist das Ergebnis der Prüfung

Operator für die Prüfung der leeren Menge

Die leere Menge erkennen wir leicht daran, dass alle Felder des Arrays den Wert 0 haben.

```
int operator! ( const set& s )
{
    int i;
    for( i = 0; i < 32; i++ )
    {
        if( s.data[i] )
            return 0;
    }

    return 1;
}
```

Prüfung aller Array-Elemente

Wenn ein Element ungleich 0 ist, ist die Menge nicht leer

Wenn kein vorzeitiger Rücksprung erfolgt ist, ist die Menge leer

Ausgabeoperator

Jetzt fehlt nur noch der Ausgabeoperator um, unsere Menge in einen ostream über cout auszugeben. Den Ausgabeoperator implementieren wir mit einem unserer überladenen Prüfoperatoren:

```
{ 2, 4, 6, 8, 10, 12}
```

Beispiel einer Ausgabe

```
ostream& operator<< ( ostream& os, const set& s )  
{  
    int i;  
    int append;  
    os << '{';  
  
    for( i = 0, append = 0; i < 256; i++ )  
    {  
        if( i < s )  
        {  
            if( append )  
                os << ',';  
            append = 1;  
            os << ' ' << i;  
        }  
    }  
    os << '}' << '\n';  
    return os;  
}
```

Ausgabe einer offenen geschweiften Klammer am Anfang

Flag um das trennende Komma vor dem ersten Element nicht auszugeben

Überladener Operator, prüft ob i im set s enthalten ist. Wenn ja erfolgt eine Ausgabe

Eine geschlossene geschweifte Klammer am Ende der Ausgabe

Test unsere Mengenklasse

Wir erstellen nun ein kleines Testprogramm, für unsere neue Klasse:

```
int main()
{
    set A;
    A += 2;
    A += 4;
    A += 6;
    A += 8;
    A += 10;
    A += 12;

    set B;
    B += 2;
    B += 4;
    B += 6;
    B += 7;
    B += 9;
    B += 11;

    cout << " A = " << A;
    cout << " B = " << B << '\n';

    cout << " A + B = " << A + B;
    cout << " A * B = " << A * B;
    cout << " A - B = " << A - B;
}
```

```
A = { 2, 4, 6, 8, 10, 12}
B = { 2, 4, 6, 7, 9, 11}
```

```
A + B = { 2, 4, 6, 7, 8, 9, 10, 11, 12}
A * B = { 2, 4, 6}
A - B = { 8, 10, 12}
```

Fortsetzung der Ausgabe

Wir erweitern unser Testprogramm nun noch um einige Prüfungen und erhalten das folgende Ergebnis:

```
int main()
{
    // Erster Teil von main
    if( !( A*B ) )
        cout << "Der Durchschnitt von A und B ist leer\n";
    else
        cout << "Der Durchschnitt von A und B ist nicht leer\n";

    if( !( A <= B ) )
        cout << "A ist keine Teilmenge von B\n";
    cout << "Berechnung einiger Formeln\n";
    cout << "(A + 1) * ~(B + 8) = \n" << ( A + 1 ) * ~( B + 8 );
    cout << "((A + 1) * ~(B + 8)) - 10 = \n" << ( ( A + 1 ) * ~( B + 8 ) ) - 10;
    cout << "((A + 1) * ~(B + 8) - 10) + B = \n" << ( ( A + 1 ) * ~( B + 8 ) - 10 ) + B;

    if( ( A*B + 15 ) <= ( B + 15 ) )
        cout << " A*B+15 ist Teilmenge von B+15\n";

    A += ( B - 11 );
    cout << "A = " << A;
    A *= ( B -= 2 );
    cout << "A = " << A;
    cout << "B = " << B;

    return 0;
}
```

```
Der Durchschnitt von A und B ist nicht leer
A ist keine Teilmenge von B
Berechnung einiger Formeln
(A + 1) * ~(B + 8) =
{ 1, 10, 12}
((A + 1) * ~(B + 8)) - 10 =
{ 1, 12}
((A + 1) * ~(B + 8) - 10) + B =
{ 1, 2, 4, 6, 7, 9, 11, 12}
A*B+15 ist Teilmenge von B+15
A = { 2, 4, 6, 7, 8, 9, 10, 12}
A = { 4, 6, 7, 9}
B = { 4, 6, 7, 9, 11}
```

Bingo

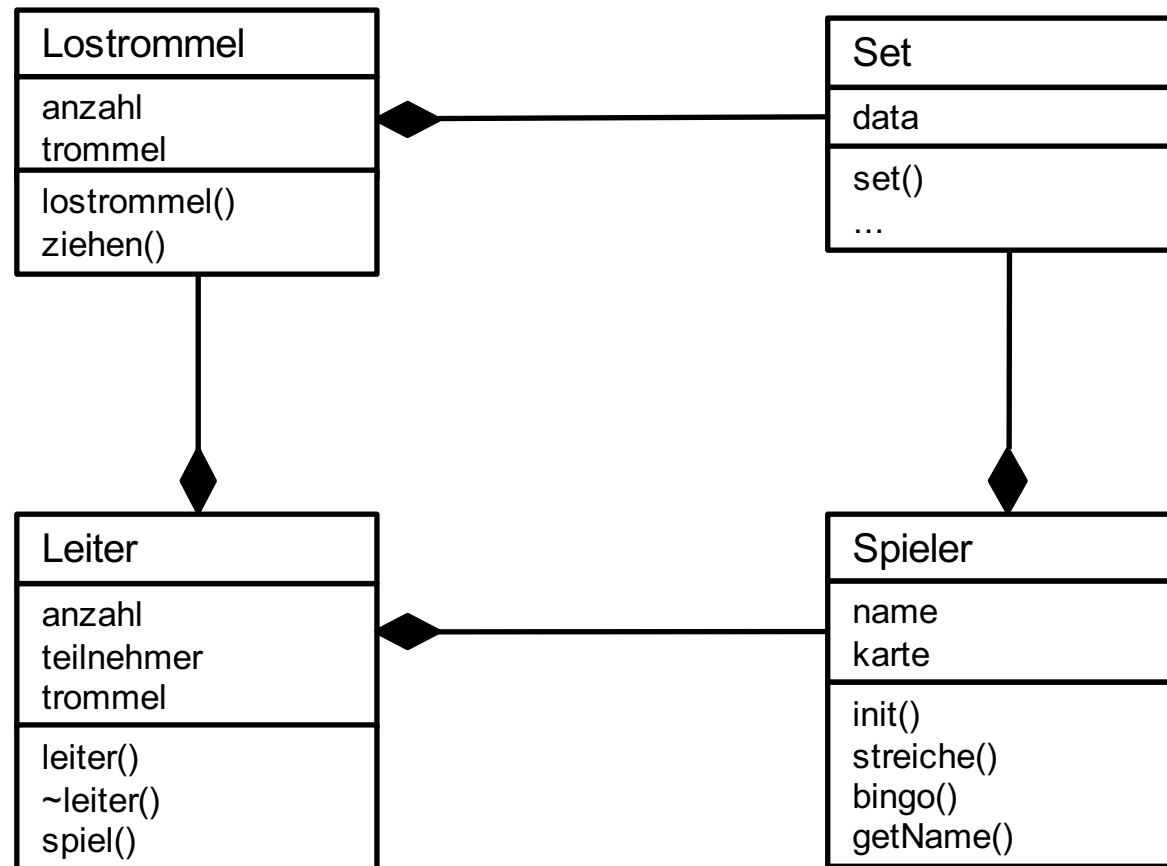
Im vorigen Beispiel haben wir eine Klasse zur Verwaltung von Mengen implementiert. Wir wollen diese Klasse nun sogleich verwenden und ein Bingo Spiel programmieren. Bingo ist ein Glücksspiel, an dem eine beliebige Anzahl von Spielern teilnehmen kann.

Jeder Spieler hat vor sich eine Karte, auf der zehn Zahlen von 1 bis 50 notiert sind. Der Spielleiter zieht aus einer Lostrommel Zahlen (0 – 50) und ruft diese öffentlich aus. Immer wenn ein Spieler die gezogene Zahl auf seiner Karte findet, streicht er die Zahl durch.

Wer als erster alle Zahlen durchgestrichen hat, hat gewonnen – Bingo!

Die verwendeten Klassen in der Übersicht

Die Grundelemente Lostrommel, Spieler und Spielleiter sind bereits genannt worden. Diese drei Klassen verwenden die bereits implementierte Set-Klasse und bilden damit das Bingo-Spiel:



Lostrommel

Wir werden das Spiel Schritt für Schritt implementieren und beginnen mit der Lostrommel. Die zugehörige Klasse deklariert nur wenige Elemente.

```
class lostrommel
```

```
{
```

```
  private:
```

```
    int anzahl;
```

```
    set trommel;
```

```
  public:
```

```
    lostrommel( int max );
```

```
    int ziehen();
```

```
};
```

Anzahl der Kugeln in der Lostrommel

Menge von Kugeln

Methoden der Klasse

Konstruktor der Lostrommel

Der Konstruktor der Lostrommel erwartet als Parameter die Anzahl der Kugeln in der Lostrommel und füllt die Trommel dann entsprechend:

```
lostrommel::lostrommel( int max )  
{  
    int i;  
    anzahl = max + 1;  
    for( i = 0; i < anzahl; i++ )  
        trommel += i;  
}
```

Höchste vorkommende Nummer

Anzahl der Kugeln (0 bis max)

Auffüllen der Lostrommel

Operator '+=' fügt das Element
e der Menge A hinzu

Die Methode `ziehen` der `Lostrommel`

Um eine Kugel zu ziehen, ermitteln wir zunächst eine Zufallszahl `z` zwischen 0 und der Anzahl der noch vorhandenen Kugeln.

```
int lostrommel::ziehen()
{
    int z, i, x;
    if( !anzahl )
        return -1;

    z = rand() % anzahl;

    for( x = 0, i = 0; i <= z; x++ )
    {
        if( x < trommel )
            i++;
    }

    x--;
    trommel -= x;
    anzahl--;
    return x;
}
```

Nur wenn noch Kugeln in der Trommel sind

Ermittlung der Zufallszahl `z`

Wir gehen durch `z` Kugeln

Prüfen ob Kugel `x` noch in Trommel enthalten

Wir entnehmen die `z`-te Kugel mit dem Wert `x` und dekrementieren die Anzahl der Kugeln in der Trommel

Rückgabe des Wertes der entnommenen Kugel

Die Klasse `spieler`

In die Klasse `spieler` nehmen wir den Namen des Spielers und seine Spielkarte auf. Auch die Spielkarte ist wie die Lostrommel als Menge implementiert.

Drei der Methoden werden inline in der Klasse realisiert:

```
class spieler
{
    friend ostream& operator <<( ostream& os, spieler& sp );

private:
    char name[20];
    set karte;

public:
    void init( int anz, int max );
    void streiche( int z ) { karte -= z; }
    int bingo() { return !karte; }
    char* getName() { return name; }

};
```

Die Spielkarte des Spielers wird auch als set implementiert

Die Zahl `z` wird aus der Karte gestrichen

Wenn die Karte leer ist, hat der Spieler ein Bingo

Die Klasse kann den Namen des Spielers zurückgeben

Die methode `spieler::init`

Die Initialisierung des Spielers erfordert die Eingabe des Spielernamens. Danach wird über eine eigens für den Spieler temporär erstellte Lostrommel eine Karte für den Spieler erstellt. Auch hier kommt wieder unsere Menge zum Einsatz.

```
void spieler::init( int k, int max )
{
    cout << "Name: ";
    cin >> name;

    lostrommel ltr( max );
    for( ; k; k-- )
        karte += ltr.ziehen();

    cout << *this;
}
```

Der Name des Spielers wird
eingegeben

Eine temporäre Lostrommel
wird erstellt

Es wird k mal gezogen um die
Karte zufällig zu füllen

Name und Karte des Spielers werden
über den noch zu Implementierenden
Ausgeboperator ausgegeben

Der Ausgabeoperator für den Spieler


Der Ausgabeoperator für die Klasse `spieler` ist unter Verwendung der Ausgabe der Menge unkompliziert erstellt:

```
ostream& operator<< ( ostream& os, spieler& sp )  
{  
    return os << sp.name << ": " << sp.karte;  
}
```

Ausgabe des Spielernamens

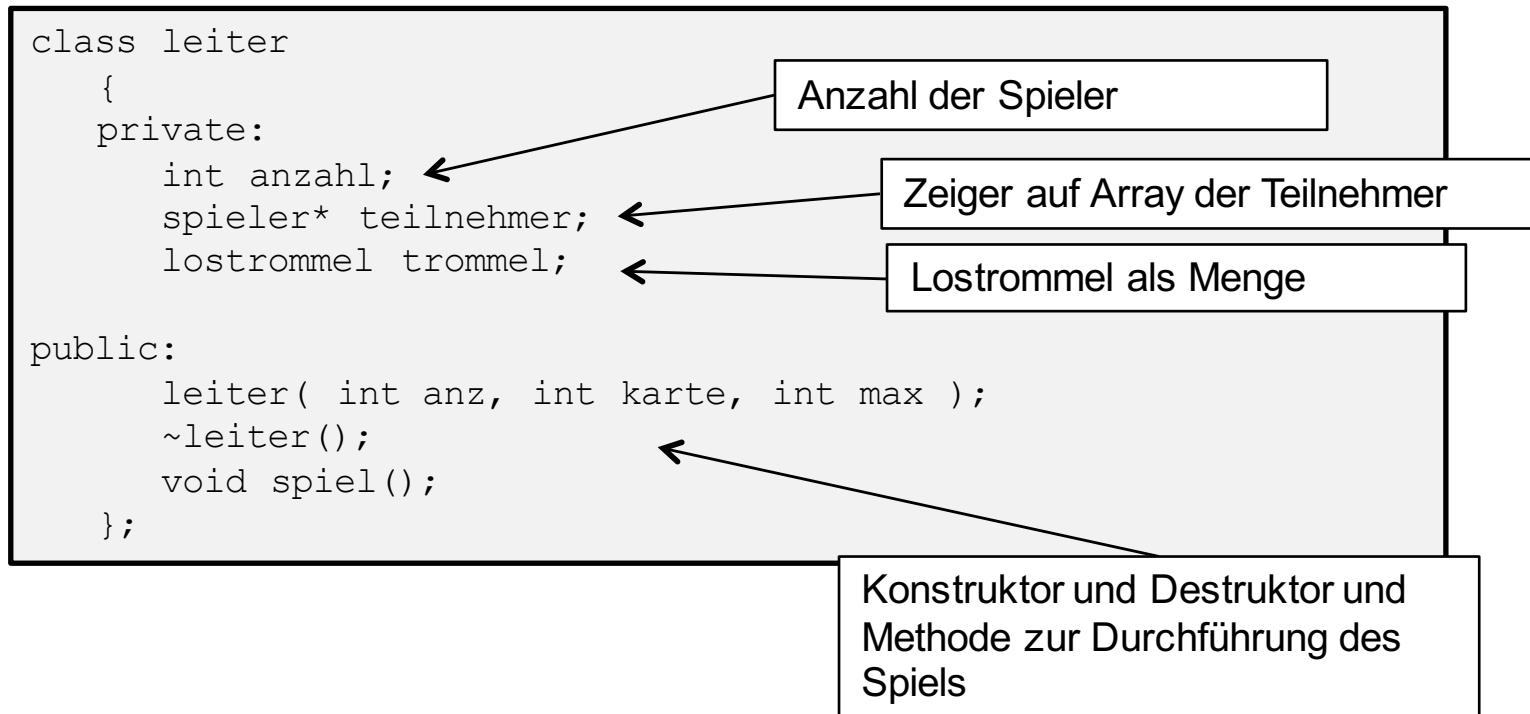


Ausgabe der verbliebenen Zahlen
auf der Karte



Die Klasse `leiter` für den Spielleiter

Der Spielleiter verfügt über eine Lostrommel und verwaltet die Mitspieler;



Konstruktor und Destruktor der Klasse `leiter`

In seinem Konstruktor erhält der Spielleiter drei Parameter, die die Rahmendaten des zu leitenden Spieles bestimmen. Den Parameter `max` mit der größtmöglichen Zahl auf den Karten verwendet er zur Konstruktion seiner Lostrommel.

Mit `anz` für Anzahl der Spieler erstellt er das Array der Spieler, die er alle über ihre `init` Methode initialisiert und ihnen die Anzahl der Zahlen auf der Karte über `karte` mitgibt.

```
leiter::leiter( int anz, int karte, int max ) : trommel( max )
{
    teilnehmer = new spieler[anz];
    for( anzahl = 0; anzahl < anz; anzahl++ )
        teilnehmer[anzahl].init( karte, max );
}
```

Initialisiererliste zur Konstruktion der Lostrommel

Dynamisches Array der Teilnehmer

Initialisierung der Teilnehmer

Der Destruktor muss nur die dynamisch allokierten Spieler beseitigen:

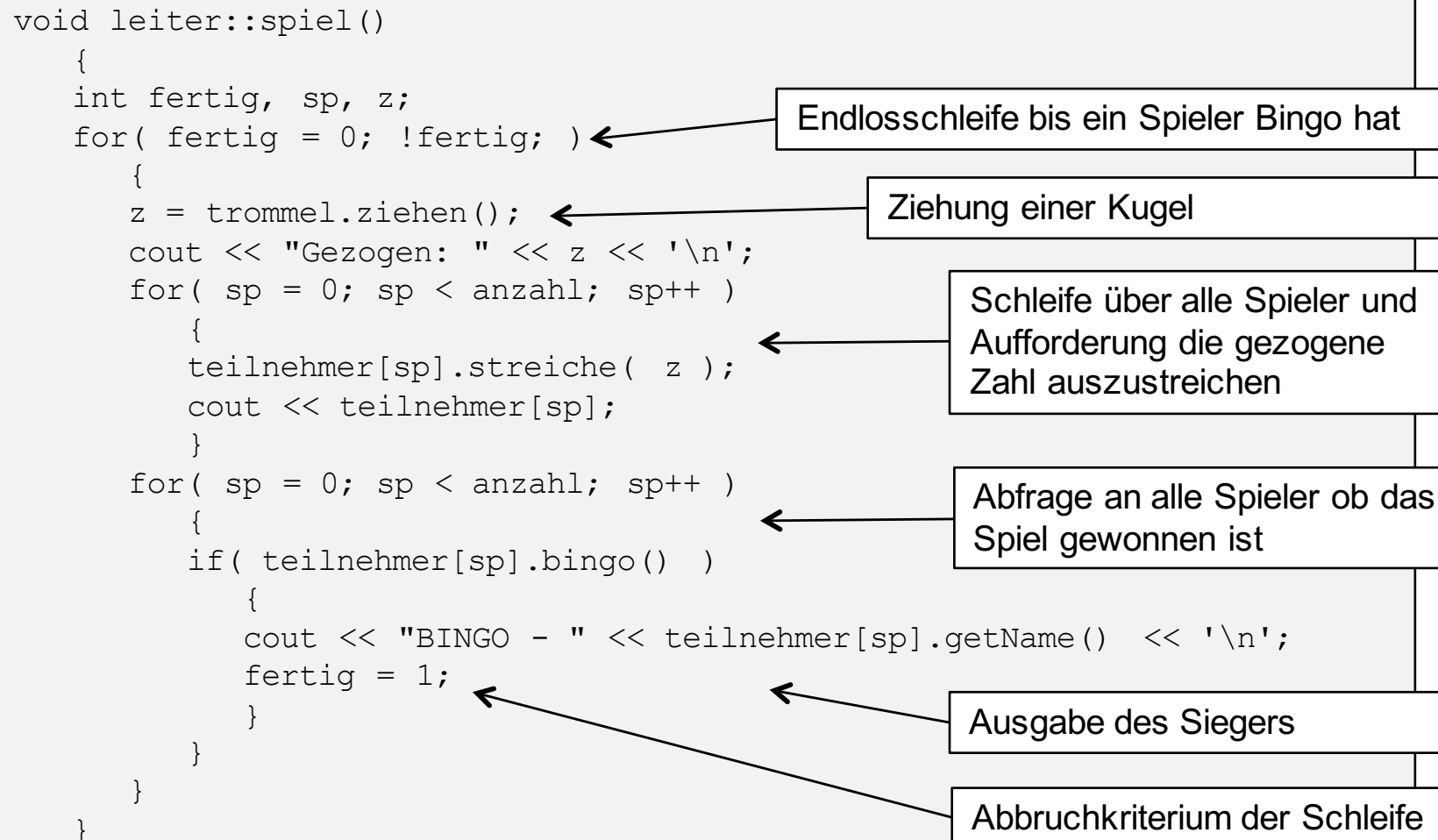
```
leiter::~~leiter()
{
    delete[] teilnehmer;
}
```

Freigabe des Arrays der Spieler

Der Spielleiter verwaltet dynamisch allokierte Ressourcen. Im Sinne der Vollständigkeit sollte daher auch ein Copy-Konstruktor und ein Zuweisungsoperator definiert werden. Diese werden wir in diesem Beispiel aber nicht aufführen.

Die Methode `spiel`

Mit der Methode `spiel` startet der Spielleiter das Spiel. In einer Schleife zieht er jeweils eine Zahl aus der Lostrommel und fordert alle Spieler auf, die gezogene Zahl von ihrer Karte zu streichen. Anschließend fragt er alle Spieler, ob einer von ihnen ein Bingo hat. Ist das der Fall, wird der Name des entsprechenden Spielers ausgegeben. Der Spielleiter beendet das Spiel, wenn in einer Spielrunde mindestens ein Spieler Bingo hatte.



Das Hauptprogramm main

Durch die Verteilung der Aufgaben auf die verschiedenen Objekte, hat das Hauptprogramm nur noch verhältnismäßig wenig zu erledigen. Es initialisiert nur noch den Zufallsgenerator und erfragt vom Nutzer die erforderlichen Spielparameter. Danach wird der Spielleiter instanziiert, der dann das Spiel durchzuführen hat..

```
void main()
{
    int seed, anzahl, karte, maximum;
    cout << "Startwert fuer Z-Generator: ";
    cin >> seed;
    srand( seed );
    cout << "Anzahl Teilnehmer: ";
    cin >> anzahl;
    cout << "Kartengroesse: ";
    cin >> karte;
    cout << "Maximum: ";
    cin >> maximum;
    if( maximum > 63 )
        maximum = 63;
    if( karte > maximum + 1 )
        karte = maximum + 1;
    leiter ltr( anzahl, karte, maximum );
    ltr.spiel();
}
```

Abgefragten Startwert für den
Zufallsgenerator setzen

Spielparameter abfragen

Spielleiter konstruieren

Spielleiter mit Durchführung beauftragen

Ein Beispieldurchlauf

Ein Beispieldurchlauf könnte dann folgendermaßen aussehen:

Spielparameter werden eingegeben

Spieler Anton, Berta und Claus werden initialisiert

Erste Kugel wird gezogen (7)

7 bei Claus von der Karte gestrichen, das Spiel geht weiter

Claus hat nur noch eine Zahl auf der Karte

Claus hat gewonnen – Bingo!

```
Startwert fuer Z-Generator: 1
Anzahl Teilnehmer: 3
Kartengroesse: 4
Maximum: 8
Name: Anton
Anton: { 3, 5, 6, 8}
Name: Berta
Berta: { 0, 4, 6, 8}
Name: Claus
Claus: { 0, 1, 7, 8}
Gezogen: 7
Anton: { 3, 5, 6, 8}
Berta: { 0, 4, 6, 8}
Claus: { 0, 1, 8}
Gezogen: 3
Anton: { 5, 6, 8}
Berta: { 0, 4, 6, 8}
Claus: { 0, 1, 8}
Gezogen: 0
Anton: { 5, 6, 8}
Berta: { 4, 6, 8}
Claus: { 1, 8}
Gezogen: 8
Anton: { 5, 6}
Berta: { 4, 6}
Claus: { 1}
Gezogen: 1
Anton: { 5, 6}
Berta: { 4, 6}
Claus: {}
BINGO – Claus
```