Programmieren in C++

Veranstaltungsbegleitendes Skript

Lehrbeauftragter an der Beuth Hochschule für Technik Berlin

Internet: public.beuth-hochschule.de/~kempfer
www.kempfer.de
E-Mail: gerald@kempfer.de

Stand: 09. Februar 2020

Inhaltsverzeichnis

1. Erweiterungen und Änderungen gegenüber C (ANSI-C)

1.1. Die Entwicklung der Programmiersprache C++

Die Geschichte von C++ begann 1979 mit der Doktorarbeit von Bjarne Stroustrup. Dabei arbeitete er mit der Programmiersprache Simula 67, die hauptsächlich für Simulationen verwendet wurde. Es war gleichzeitig auch eine der ersten Sprachen, die auf einem objektorientierten Modell basierten. Bjarne Stroustrup empfand dieses Modell als sehr hilfreich für die Softwareentwicklung, aber die Programmiersprache Simula war viel zu langsam. Also begann er, dieses Modell basierend auf der Programmiersprache C weiter zu entwickeln, um eine schnelle und Hardware-nahe Programmiersprache mit Objektorientierung zu schaffen. Damit war der Grundstein für "C mit Klassen" gelegt. Erst 1983 wurde der Name C++ eingeführt. Dabei ist das ++ eine Anspielung auf den Inkrement-Operator in C, der den Wert einer Variablen um 1 erhöht.

Der erste Compiler für diese neue Programmiersprache war Cfront, der selber in "C mit Klassen" programmiert war. Cfront war sehr verbreitet im Einsatz - bis 1993 in C++ die Exceptions eingeführt wurden. Diese konnte in Cfront nicht implementiert werden, so dass die Weiterentwicklung des Compilers eingestellt wurde. Im Jahr 1990 kam der Compiler Turbo C++ von Borland auf den Markt. Obwohl die letzte stabile Aktualisierung dieses Compilers im 2006 veröffentlicht wurde, ist Turbo C++ bis heute sehr beliebt und noch vielfach im Einsatz.

Der erste Standard wurde 1998 unter dem Namen ISO/IEC 14882:1998 (kurz C++ 98) veröffentlicht. Fünf Jahr später wurden einige Verbesserungen und Korrekturen mit dem nächsten Standard ISO/IEC 14882:2003 (kurz C++ 03) veröffentlicht. Die nachfolgenden Standards ab 2011 werden ab Kapitel 15 beschrieben. Bis zu diesem Kapitel wird nur auf den C++ 03 - Standard eingegangen.

Da C++ eine Weiterentwicklung von C (genau genommen von ANSI-C) ist, können C-Programme von C++-Compilern übersetzt werden; es können auch die Standard-Bibliotheken von C verwendet werden. Allerdings lassen sich nicht immer C-Programme 1:1 in C++ übernehmen, da es einige kleine Änderungen und Unterschiede gibt. Auf diese Unterschiede wird in diesem Kapitel eingegangen.

1.2. Zeilen-Kommentare

Kommentare werden auch in C++ wie bisher in `/*` und `*/` eingesetzt. Des Weiteren können in C++ auch einzeilige Kommentare geschrieben werden. Diese müssen hinter einem doppelten Schrägstrich ("`//`") stehen. Eine Zeichenkombination für das Ende des Kommentars ist nicht nötig, da diese Kommentare immer bis zum Zeilenende reichen. Soll in der nächsten Zeile ein weiterer Kommentar stehen, muss wieder ein doppelter Schrägstrich vorangestellt werden.

Eigentlich ist dies keine wirkliche Neuerung, da die Zeilen-Kommentare bereits in der Programmiersprache BCPL (einem Vorgänger von C) vorhanden waren.

Beispiel:

```
int i;       // Laufvariable
double Erg; // Ergebnisvariable
```

1.3. Platzierung von Variablendeklarationen

Während in C die Variablendeklaration und -definition am Anfang einer Funktion stehen muss, kann in C++ die Deklaration und Definition an einer beliebigen Stelle durchgeführt werden, jedoch unbedingt **vor der ersten Benutzung der Variable**.

Beispiel:

kap01_01.cpp

```
01 #include <stdio.h>
02
```

```
03 int main()
04 {
05    printf("Die Variable Zahl hat folgenden Wert: ");
06    int Zahl = 5;
07    printf("%i\n", Zahl);
08
09    return 0;
10 }
```

Wird innerhalb eines Blockes (also innerhalb eines Paars geschweifter Klammern `{}`) eine Variable deklariert, ist die Variable nur innerhalb dieses Blockes gültig. Im folgenden Beispiel existiert die Variable `temp` nur innerhalb des Schleifenblocks. Wird die Variable außerhalb des Blocks verwendet, gibt der Compiler einen Fehler aus.

Beim Schleifenzähler `count` dagegen hängt es vom verwendeten Compiler (und der dahinter stehenden Philosophie) ab: Zum Einen kann argumentiert werden, dass jede `for`-Schleife in eine `while`-Schleife umgeschrieben werden kann. Der Schleifenzähler muss dann vor der `while`-Schleife definiert werden und ist damit außerhalb des Schleifenblocks definiert (Schließlich liegt die Variablendefinition bei der `for`-Schleife auch vor der sich öffnenden geschweiften Klammer!). Dann ist der Schleifenzähler natürlich auch nach der Schleife noch gültig. Zum anderen wird argumentiert, dass die `for`-Schleife mit dem Schlüsselwort `for` beginnt und die Variablendefinition erst dahinter liegt. Daraus würde folgen, dass die Variable nur innerhalb des Schleifenblocks gültig wäre.

Beispiel:

kap01_02.cpp

```
01 #include <stdio.h>
02
03 int main()
04 {
05    for (int count = 0; count < 10; count++)
06    {
07       int temp = 34;        // innerhalb des Schleifenblocks!
08       temp++;
09    }
10    printf("%i\n", temp);    // FEHLER!
11    printf("%i\n", count);   // FEHLER in Abhaengigkeit vom Compiler!
12
13    return 0;
14 }
```

Die Möglichkeit, die Variablen überall deklarieren und definieren zu können, sollte sparsam verwendet werden! Sonst leidet die Lesbarkeit des Programms darunter. Es sollten also die meisten Variablen am Anfang des Programms oder der Funktion deklariert und definiert werden und nur in Ausnahmefällen an anderen Stellen, wenn sich die Lesbarkeit des Programms damit erhöht, wie z.B. bei Schleifenzählern.

1.4. const

Mit dem Schlüsselwort `const` kann eine Variable zu einer unveränderlichen Variable erklärt werden, d.h. die Variable kann zwar mit einem Wert initialisiert, dann aber nicht mehr verändert werden. Während in C diese unveränderlichen Variablen nicht an den Stellen eingesetzt werden konnten, an denen konstante Zahlenwerte erwartet wurden (z.B. die Größe eines Arrays bei der Definition), ist dies in C++ möglich.

Beispiel:

kap01_03.cpp

```
01 #include <stdio.h>
02
```

```
03 int main()
04 {
05    int const Groesse = 10;
06    char Feld[Groesse];   // war in C nicht moeglich!!!
07
08    return 0;
09 }
```

Unveränderlichen Variablen, die mit `const` erzeugt werden, bieten einige Vorteile:

- Sie können mit einem Debugger (einem Hilfsprogramm zur Fehlersuche) angezeigt werden.
- Es können Zeiger auf solche unveränderlichen Variablen deklariert werden.
- Eine lokale Definition von unveränderlichen Variablen ist möglich.

1.5. struct, enum und union

Während in C bei der Deklaration einer Variablen des Typs `struct`, `union` oder `enum` das Schlüsselwort des Typs vorangestellt werden muss, kann es in C++ weggelassen werden. Dies gilt natürlich nicht für die Definition des Typs selbst.

Beispiel:

kap01_04.cpp

```
01 #include <stdio.h>
02
03 int main()
04 {
05    struct Adresse
06    {
07       char Name[50];
08       char Strasse[50];
09       int Hausnummer;
10       char Wohnort[50];
11    };
12    struct Adresse Adresse1; // mit "struct" in C
13    Adresse Adresse2;        // kein "struct" noetig in C++
14
15    return 0;
16 }
```

1.6. Funktionsprototypen

In C konnten Funktionsprototypen weggelassen werden, wenn auf die automatische Typüberprüfung durch den Compiler verzichtet werden konnte. Für C++-Compiler müssen die Funktionsprototypen immer angegeben werden, es sei denn, die Funktionsdefinition steht vor dem Funktionsaufruf.

1.7. Funktionsdeklaration ohne Angabe des Rückgabewertes

Während in C bei der Funktionsdeklaration und -definition das Weglassen des Rückgabetyps erlaubt war (es wurde dann implizit der Rückgabetyp `int` angenommen), erwartet C++ in jedem Fall einen Rückgabetyp. Wenn kein Rückgabewert geliefert werden soll, muss als Rückgabetyp `void` angegeben werden.

Beispiel:

```
main()          // moegliche (wenn auch veraltete) main-Funktion in C
{
   int i = 0;
   return i;
}

int main()      // gleiche Funktion in C++
{               // mit dem notwendigen Rueckgabetyp
   int i = 0;
   return i;
}
```

1.8. Referenzen

Bis jetzt gab es die Variablen und Zeiger auf Variablen. In C++ gibt es nun noch die **Referenzen**. Eine Referenz ist ein Alias für eine Variable; die Variable kann dann unter zwei Namen angesprochen werden. Dabei wird für die Referenz kein weiterer Speicherplatz benötigt (darin liegt der Hauptunterschied zu Zeigern).

Referenzvariablen

Referenzvariablen werden eher selten verwendet. Eine Referenzvariable wird genauso wie eine einfache Variable deklariert. Vor dem Variablennamen muss allerdings ein kaufmännisches Und (&, wird hier **Referenzoperator** genannt) stehen. Ferner muss eine Referenzvariable immer sofort initialisiert werden. Dazu wird ihr über den Zuweisungsoperator die zu referenzierende Variable selber (und nicht die Adresse der Variable) zugewiesen. In der folgenden Zeile wird die allgemeine Syntax für Referenzvariablen angegeben.

```
Datentyp &Referenzvariablenname = Variablenname;
```

Verwechseln Sie den Referenzoperator **nicht** mit dem Adressoperator. Der Referenzoperator steht immer links vom Zuweisungsoperator (=), der Adressoperator dagegen immer rechts davon.

Prinzipiell kann auch über Zeiger eine Variable referenziert werden. Bei der Verwendung von Referenzvariablen entfällt jedoch die Dereferenzierung wie bei einem Zeiger. Während eine Referenz ein Alias für eine Variable ist, stellt ein Zeiger eine unabhängige Variable dar.

Beispiel:

```
int i, *Zeiger;     // int-Variable und Zeiger auf int
int &Referenz = i; // Referenz auf die Variable i
Zeiger = &i;        // Zeiger zeigt nun auf i

Referenz = 10;      // i wird über die Referenz auf 10 gesetzt
*Zeiger = 10;       // i wird über den Zeiger auf 10 gesetzt
```

Der Einsatz von Referenzvariablen kann z.B. auch nützlich beim Zugriff auf verschachtelte Teile einer Struktur sein.

Beispiel:

```
struct Geburtstag
{
   int Tag, Monat, Jahr;
};

struct Person
{
   char Name[25];
   char Ort[25];
   Geburtstag Geb;
```

```
};

Person P;
int &GebMonat = P.Geb.Monat; // Referenz auf ein Feld
                             // in der Unterstruktur
GebMonat = 4;                // entspricht P.Geb.Monat = 4
```

Referenzparameter (Übergabe per Referenz)

Für den Datentransfer in die Funktion hinein gibt es in C++ neben den beiden schon bekannten Arten von C (Übergabe per Wert und Übergabe per Zeiger) noch die **Übergabe per Referenz**.

Soll ein übergebenes Objekt von der Funktion modifiziert werden können, kann die Übergabe per Referenz geschehen. Die Syntax für den Aufruf ist die gleiche wie bei der Übergabe per Wert, allerdings wird hier innerhalb der Funktion direkt mit dem Original gearbeitet, d.h. nach Beenden der Funktion kann (muss aber nicht) das Originalobjekt einen anderen Wert haben. Weil keine Kopie erzeugt wird, ergibt sich bei sehr großen Objekten eine kürzere Laufzeit. Schließlich wird ja nur die Referenz zum Objekt (ähnlich wie ein Zeiger) im Stack angelegt.

Der einzige Unterschied zwischen der Übergabe per Wert und per Referenz liegt bei den Datentypen der Parameter (bei Funktionsprototypen und -definitionen): Für eine Übergabe per Referenz wird an den Datentypen des Parameters ein kaufmännisches Und ('`&`') herangehangen (im Beispiel: `int& x` anstelle von `int x`) oder vor dem Variablennamen gesetzt (`int &x`). Vom Beispiel vom Kapitel *Funktionsschnittstelle* ausgehend wird bei der Funktion jetzt eine Übergabe per Referenz durchgeführt.

Beispiel:

kap01_05.cpp

```
01 #include <stdio.h>
02
03 int Addiere_3(int&);
04
05 int main()
06 {
07    int Ergebnis, Zahl = 2;
08
09    printf("Wert von Zahl = %i\n\n", Zahl);
10    Ergebnis = Addiere_3(Zahl);
11    printf("Ergebnis 'Addiere_3(Zahl)' = %i\n\n", Ergebnis);
12    printf("Wert von Zahl = %i (veraendert!!!)\n\n", Zahl);
13
14    return 0;
15 }
16
17 int Addiere_3(int& x)
18 {
19    x += 3;
20    return x;
21 }
```

Referenzen haben einen tückischen Nachteil: Sie sehen anhand der Parameterübergabe an eine Funktion nicht, ob diese den Parameter als Wert oder Referenz erhält. Wenn Referenzen eingesetzt werden sollen, um große Strukturen zeitsparend zu übergeben, diese aber nicht geändert werden sollen, kann in der Funktionsdefinition das Schlüsselwort `const` verwendet werden. Dadurch wird vermieden, dass der Parameter in der Funktion geändert werden kann, und trotzdem muss die gesamte Struktur nicht kopiert werden.

Beispiel:

```
int Addiere_3(const int& x)
{
```

```
   x += 3; // jetzt nicht mehr möglich!
   return x;
}
```

Referenzen als Funktionsergebnis

Funktionen können auch Referenzen als Funktionsergebnis zurückliefern. Dieser Mechanismus bringt im Zusammenhang mit dem Überladen von Operatoren (siehe Kapitel *Überladen von Operatoren*) entscheidende Vorteile.

Um in einer Funktion eine Referenz auf eine Variable zurückzugeben, wird ein kaufmännisches Und (`&`) entweder dem Rückgabedatentypen angehangen oder dem Funktionsname vorangestellt. Der Aufruf der Funktion erfolgt wie bisher.

Genauso wie bei der Rückgabe eines Zeigers dürfen keine Referenzen auf lokale Variablen einer Funktion zurückgegeben werden. Diese verlieren ihre Gültigkeit nach Verlassen der Funktion und existieren dann nicht mehr.

Im folgenden Beispiel erhält die Funktion `FindMax` einen Zeiger auf das Zahlenfeld sowie die Anzahl der Feldelemente und gibt eine Referenz auf das Element des Arrays zurück, das den größten Wert beinhaltet.

Beispiel:

kap01_06.cpp

```
01 #include <stdio.h>
02
03 int &FindMax(int *, int);
04
05 int main()
06 {
07    int Liste[] = {12, 165, 4, 3345, 345, 3434, 34, 87, 11, 100};
08
09    printf("Maximum: %i\n", FindMax(Liste, 10));
10    return 0;
11 }
12
13 int &FindMax(int Feld[], int Anzahl)
14 {
15    int Max = 0;
16
17    for (int i = 0; i < Anzahl; i++)
18       if (Feld[i] > Feld[Max])
19          Max = i;
20    return Feld[Max]; // Referenz auf groessten Wert
21 }
```

1.9. Variable Parameteranzahl

Funktionen können in C++ auch mit variabler Parameteranzahl aufgerufen werden. Dabei werden für die Parameter in den Funktionsprototypen **vorgegebene Werte** (**default values**) angegeben. Dabei wird ganz ähnlich wie bei der Variableninitialisierung vorgegangen:

Beispiel:

kap01_07.cpp

```
01 #include <stdio.h>
02
03 void PreisAnzeige(double, char * = "EUR");
04
```

```
05 int main()
06 {
07    PreisAnzeige(19.99);
08    printf("\n");
09    PreisAnzeige(14.99, "USD");
10    printf("\n");
11
12    return 0;
13 }
14
15 void PreisAnzeige(double Preis, char *Waehrung)
16 {
17    printf("%.2f %s", Preis, Waehrung);
18 }
```

Das Programm gibt folgende Zeilen aus:

```
19.99 EUR
14.99 USD
```

Beim ersten Funktionsaufruf wird die Standardwährung verwendet; daher braucht keine Währung angegeben werden. Beim zweiten Preis weicht die Währung vom Standard ab und wird als zweiten Parameter mitgegeben.

Wichtig: Die Parameter mit Vorgabewerten erscheinen in den Funktionsprototypen immer **nach** den anderen, festen Parametern! Bei mehreren Parametern mit Vorgabewerten können beim Funktionsaufruf immer nur die hintersten Parameter weggelassen werden!

1.10. Überladen von Funktionen

Funktionen können in C++ **überladen** werden. D.h. es dürfen mehrere Funktionen mit dem gleichen Funktionsnamen existieren, die aber alle unterschiedliche Parameter haben. Sinnvollerweise führen diese Funktionen gleichartige Operationen durch. In der Literatur wird dieses teilweise auch mit **Polymorphismus** (zu Deutsch Vielgestaltigkeit) bezeichnet. Polymorphismus bezieht sich aber auf Klassen und deren Vererbung und wird daher später noch einmal aufgegriffen und dort genauer definiert.

Während des Programmablaufs hängt die Entscheidung, welche Funktion von den mehreren Funktionen gleichen Namens aufgerufen wird, vom Kontext ab. Das Programm trifft die richtige Zuordnung anhand der **Signatur** der Funktion. Die Signatur besteht aus der Kombination des Funktionsnamens mit Reihenfolge und Typen der Parameter.

Beispiel:

kap01_08.cpp

```
01 #include <stdio.h>
02
03 int Maximum(int, int);
04 double Maximum(double, double);      // ueberladene Funktion
05
06 int main()
07 {
08    int a = 843, b = 362;
09    double c = 2343.43, d = 8723.23;
10
11    printf("%i\n", Maximum(a, b));    // Maximum(int, int)
12    printf("%f\n", Maximum(c, d));    // Maximum(double, double)
13
14    return 0;
15 }
16
```

```
17 int Maximum(int x, int y)
18 {
19    return (x > y) ? x : y;
20 }
21
22 double Maximum(double x, double y)
23 {
24    return (x > y) ? x : y;
25 }
```

Das Programm versucht immer, die beste Übereinstimmung mit den Parametertypen zu finden. Wird z.B. die Maximums-Funktion im obigen Beispiel mit zwei `float`-Zahlen aufgerufen, wird die zweite Funktion aufgerufen, da `float`-Zahlen implizit in `double`-Zahlen umgewandelt werden. Wird dagegen die Funktion mit zwei `char`-Zeichen aufgerufen, wird die erste Funktion aufgerufen, da `char`-Zeichen implizit in `int`-Zahlen umgewandelt werden. Problematisch wird es bei einem Aufruf mit einer `double`- und einer `int`-Zahl. Der Compiler kann sich nicht entscheiden und erzeugt eine Fehlermeldung.

1.11. Inline-Funktionen

Ein Funktionsaufruf kostet Rechnerzeit: Der Zustand der aufrufenden Funktion muss gesichert werden, die Parameter müssen evtl. kopiert werden (siehe Kapitel *Funktionen Abschnitt Funktionsschnittstelle, Übergabe per Wert* im Skript „*Programmieren in C*“). Dann springt das Programm an eine andere Stelle und nach dem Ende der Funktion wieder zurück zur Anweisung nach dem Aufruf. Der relative Verwaltungsaufwand und die dazugehörige Rechnerzeit fallen umso stärker ins Gewicht, je weniger Zeit die Abarbeitung der Funktion selber verbraucht. Der absolute Aufwand macht sich mit steigender Anzahl der Aufrufe bemerkbar, z.B. in Schleifen.

Um diesen Verwaltungsaufwand zu vermeiden, können Funktionen als `inline` definiert werden (muss also bei der Deklaration nicht angegeben werden). Dies bewirkt, dass bei der Kompilierung der Funktionsaufruf durch den Funktionskörper ersetzt wird. Dadurch erfolgt gar kein echter Funktionsaufruf.

Die Parameter werden entsprechend ersetzt. Auch die Syntaxprüfung bleibt erhalten.

Beispiel:

kap01_09.cpp

```
01 #include <stdio.h>
02
03 int Quadrat(int);
04
05 int main()
06 {
07    int x = 100, q;
08
09    q = Quadrat(x);
10    printf("Quadrat von %i ist %i\n", x, q);
11
12    return 0;
13 }
14
15 inline int Quadrat(int y)
16 {
17    return y * y;
18 }
```

Der Aufruf `q = Quadrat(x);` wird wegen des Schlüsselwortes `inline` vom Compiler durch `q = x * x;` ersetzt. Der Verwaltungsaufwand für den Funktionsaufruf entfällt; das Programm wird schneller.

Es ist nicht sinnvoll, diese Ersetzung selbst vorzunehmen, weil bei einer Änderung der Funktion alle betroffenen Stellen geändert werden müssten anstatt nur der Funktion selbst.

`inline` empfiehlt sich ausschließlich für Funktionen mit einer kurzen Ausführungszeit relativ zum Verwaltungsaufwand für den Aufruf. `inline` ist nur eine Empfehlung an den Compiler, die Ersetzung vorzunehmen. Er muss sich nicht daran halten.

`inline`-Definitionen müssen sich ausschließlich in der Datei, in der sie aufgerufen werden, oder in Header-Dateien befinden. Andernfalls kann der Linker die `inline`-Definition nicht finden und liefert eine Fehlermeldung.

1.12. Mischen von C und C++

Soll in einem Projekt C- und C++-Quellcode gemeinsam eingesetzt werden, müssen einige zusätzliche Angaben im Programm gemacht werden. C++-Compiler schreiben zusätzliche Informationen in die Funktionsköpfe eines kompilierten Programms, die in C fehlen. Diese Informationen enthalten die Typbeschreibungen der Funktionsparameter. Dadurch kann es unter C++ nicht passieren, dass eine Funktion mit falschen Parametern bezüglich der Datentypen aufgerufen wird.

Beispiel:

Die Funktion `int GetMax(int, int)` wird von einem C- und einem C++-Compiler wie folgt übersetzt:

C-Compiler: `_GetMax`

C++-Compiler: `@GetMax$qii` (wobei sich dieser interne Funktionsname von Compiler zu Compiler leicht unterscheiden kann).

Während bei einem C-Compiler für den internen Funktionsnamen also nur ein Unterstrich davor gesetzt wird, erhält der interne Funktionsname von einem C++-Compiler zusätzlich Informationen über die Datentypen der Parameter. Damit gibt es Probleme, wenn zu einem C++-Programm Objektdateien gelinkt werden, die mit einem C-Compiler kompiliert wurden. Um dieses Problem zu umgehen, müssen im C++-Programm vor den Funktionsprototypen der extern kompilierten Funktionen ein `extern "C"` vorgesetzt werden.

Beispiel:

In einer C-Datei wurde die Funktion `int Prod(int, int);` definiert und kompiliert. Um diese Funktion in C++ verwenden zu können, muss der Funktionsprototyp folgendermaßen aussehen:

```
extern "C" int Prod(int, int);
```

Dadurch weiß der C++-Compiler, dass diese Funktion extern (also in einem anderen Modul) mit einem C-Compiler kompiliert wurde.

Um gleich eine komplette Headerdatei mit Prototypen von Funktionen, die unter C kompiliert wurden, in einem C++-Programm einzubinden, muss die Headerdatei als `extern "C"` eingebunden werden:

```
extern "C"
{
   #include <mylib.h>
}
```

Der umgekehrte Weg, in C eine C++-Funktion einzubinden, funktioniert nur über einen Umweg. Dazu muss im C++-Programm die Funktion als `extern "C"` definiert werden.

Beispiel:

Die Funktion `int Prod(int, int);` wird in einem C++-Programm definiert und soll später in ein C-Programm gelinkt werden. Dann wird diese Funktion wie folgt definiert:

```
extern "C" int Prod(int a, int b)
{
   ...
}
```

Es können innerhalb von geschweiften Klammern vom `extern "C"` auch gleich mehrere Funktionen definiert werden.

```
extern "C"
{
   int Prod(int a, int b)
   {
      ...
   }

   int GetMax(int a, int b)
   {
      ...
   }
}
```

Dieser Weg kann auch genutzt werden, um C++-Funktionen in andere Sprachen wie Delphi, Pascal, Assembler, usw. einzubinden.

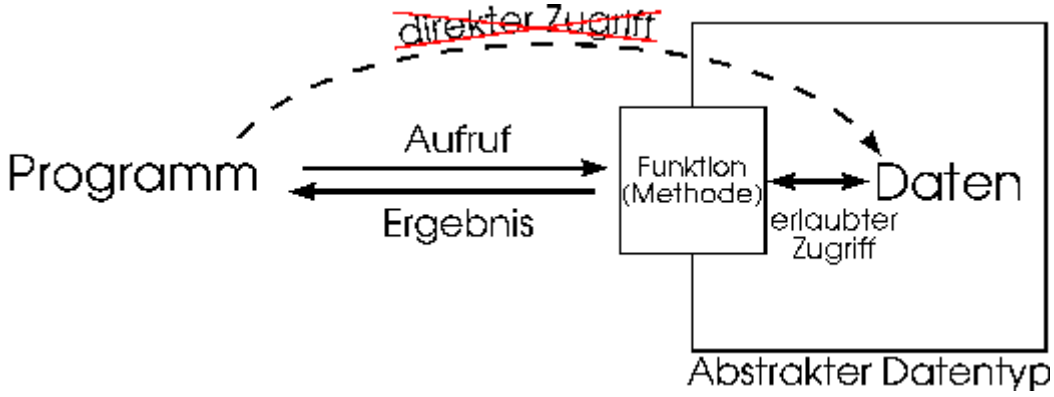
2. Objektorientierte Programmierung

Für das Objektorientierte Programmieren müssen erst einmal einige neue Begriffe eingeführt werden.

2.1. Grundlegende Begriffe der Objektorientierten Programmierung

Bisher wurden Funktionen und Daten getrennt behandelt. Dadurch ist es möglich, Daten auf Funktionen anzuwenden, die keinen Sinn ergeben, z.B. ein Geldbetrag wird auf eine Funktion angewendet, die die Temperatur von Grad Celsius nach Fahrenheit umrechnet.

Um diesem aus dem Weg zu gehen, werden Daten und Funktionen zusammengefasst zu einem **Abstrakten Datentypen**.



Die Daten eines Objektes werden jetzt **Eigenschaften** oder auch **Attribute** genannt. Die Eigenschaften eines Objektes lassen sich nur über die **Methoden** (ein anderer Begriff für Methoden ist **Elementfunktionen**) des Objektes abfragen oder verändern. Eigenschaften und Methoden sind in einem Objekt zusammengefasst.

Eine **Klasse** ist ein Abstrakter Datentyp plus Vererbung und Polymorphismus. (Vererbung und Polymorphismus werden erst etwas später beschrieben.)

Oder anders ausgedrückt:

Eine **Klasse** ist die Beschreibung von Objekten, d.h. die Abstraktion von ähnlichen Eigenschaften und Verhaltensweisen ähnlicher Objekte.

Ein **Objekt** (auch **Instanz einer Klasse** genannt) ist die konkrete Ausprägung einer Klasse, es belegt im Gegensatz zur Klasse Speicherbereiche, die Werte enthalten.

Zur Verdeutlichung werden Datentypen und Variablen, so wie wir sie bis jetzt verwendet haben, den Klassen und Objekten gegenübergestellt:

`int i;`	Dabei ist `int` der Datentyp und `i` die Variable.
`ifstream Quelle;`	`ifstream` ist die Klasse (der Abstrakte Datentyp) und `Quelle` das Objekt.

2.2. Deklarieren von Klassen und Objekten

Klassen werden meist in Header-Dateien deklariert. Mit dem Schlüsselwort `class` wird die Deklaration eingeleitet:

```
class Klassenname
{
   ...
};
```

Innerhalb der geschweiften Klammern werden lokale und globale Variablen (jetzt Eigenschaften des Objektes genannt) sowie lokale und globale Funktionen (jetzt Methoden des Objektes genannt) deklariert (NICHT definiert!). Zur Unterscheidung von lokalen und globalen Eigenschaften und Methoden gibt es die Schlüsselworte `private` und `public` (werden auch **Zugriffsattribute** bzw. **Spezifizierer** genannt). Beide können mehrmals angewendet werden, aber wegen der Übersicht werden meist zuerst die globalen und dann die lokalen Eigenschaften und Methoden angegeben. Dabei sind alle Deklarationen von Anfang an `private` bis zum ersten Erscheinen von `public`, ab da ist alles global (bis zum nächsten `private`). Ausgeschrieben sieht dieses folgendermaßen aus:

```
class Klassenname
{
   private:  // eigentlich ueberfluessig, da von Anfang an private
      ...    // lokale Eigenschaften und Methoden
   public:
      ...    // globale Eigenschaften und Methoden
};
```

An einer anderen Stelle werden die deklarierten Methoden der Klasse definiert. Dabei wird dem Funktions- (oder besser Methoden-) Namen der Klassenname und zwei Doppelpunkte vorangestellt, damit der Compiler weiß, für welche Klasse die Methoden definiert werden.

```
Rückgabetyp Klassenname::Methodenname(Parameter).
```

Die zwei Doppelpunkte werden auch **Bereichsoperator** genannt.

Im Programm muss dann zuerst das Objekt definiert werden. Dabei wird genauso wie bei der Definition einer Variablen vorgegangen.

```
Klassenname Objektname;
```

Für den Aufruf einer Methode reicht nicht der Methodenname alleine aus, da auch hier der Compiler dann nicht weiß, welches Objekt gemeint ist. Daher wird zum Aufruf einer Methode zuerst der Objektname gefolgt von einem Punkt und dem Methodennamen geschrieben. Genau wie beim Aufruf einer Funktion müssen hinter dem Methodennamen ein Klammerpaar folgen, die eventuelle Parameter einschließen. Z.B.

```
Objektname.Methodenname(evtl. Parameter);
```

Beispiel:

Es wird eine Klasse für eine Ampel konstruiert. Weiter unten folgt deren Definition.

kap02_ampel01.h

```
01 #ifndef ampel_h
02   #define ampel_h ampel_h
03
04    class Ampel
05    {
06       private:
07          enum Licht {aus, ein};
08          Licht Rot, Gelb, Gruen; // lokale Eigenschaften
09
10       public:                    // globale Methoden
11          void schalteRot();
12          void schalteRotGelb();
13          void schalteGruen();
14          void schalteGelb();
15          int istRotAn();
16          int istGelbAn();
17          int istGruenAn();
18    };
19
20 #endif // ampel_h
```

Da die drei Lichter `Rot`, `Gelb` und `Gruen` lokale Eigenschaften der Ampel-Klasse sind, kann kein Programmierer die Lichter direkt setzen. Die Lichter können nur noch mit Hilfe der globalen Methoden ein- und ausgeschaltet werden. Damit ist eine mögliche Fehlerquelle ausgeschaltet und auch Schummeln (z.B. die Lichter `Rot` und `Gruen` gleichzeitig einschalten) ist nicht mehr möglich. Nun kommt die Definition der Methoden in einer separaten Datei. Dabei ist zu beachten, dass auch in dieser Datei die Header-Datei mit der Klasse Ampel mittels `include` eingefügt werden muss!

kap02_ampel01.cpp

```
01 #include "kap02_ampel01.h"
02
03 void Ampel::schalteRot()
04 {  Rot = ein; Gelb = Gruen = aus; }
05
06 void Ampel::schalteRotGelb()
07 {  Rot = Gelb = ein; Gruen = aus; }
08
09 void Ampel::schalteGruen()
10 {  Rot = Gelb = aus; Gruen = ein; }
11
12 void Ampel::schalteGelb()
13 {  Rot = aus; Gelb = ein; Gruen = aus; }
14
15 int Ampel::istRotAn()
16 {  return (Rot == ein); }
17
18 int Ampel::istGelbAn()
19 {  return (Gelb == ein); }
20
21 int Ampel::istGruenAn()
22 {  return (Gruen == ein); }
```

Nun sind auch die Methoden definiert. Aber sie können so nicht aufgerufen werden, da noch keine Objekte existieren. Jetzt kommt das Hauptprogramm, das die Klasse Ampel anwendet und benutzt. Auch hier muss die Header-Datei eingefügt werden; die Datei mit den Definitionen der Methoden muss nur zum Projekt gehören, d.h. eigenständig kompiliert und dann zum Hauptprogramm dazu gelinkt werden.

kap02_01.cpp

```
01 #include <stdio.h>
02 #include "kap02_ampel01.h"
03
04 int main()
05 {
06    Ampel A1, A2;     // Deklaration & Definition zweier
07                      // Objekte der Klasse Ampel
08
09    A1.schalteRot(); // Aufruf der Methode ueber das Objekt
10    A2.schalteGruen();
11
12    printf("Ampel 1:\n");
13    printf("Rot  : %s, ", (A1.istRotAn()  ) ? "ein" : "aus");
14    printf("Gelb : %s, ", (A1.istGelbAn() ) ? "ein" : "aus");
15    printf("Gruen: %s\n", (A1.istGruenAn()) ? "ein" : "aus");
16    printf("Ampel 2:\n");
17    printf("Rot  : %s, ", (A2.istRotAn()  ) ? "ein" : "aus");
18    printf("Gelb : %s, ", (A2.istGelbAn() ) ? "ein" : "aus");
19    printf("Gruen: %s\n", (A2.istGruenAn()) ? "ein" : "aus");
20
```

```
21     return 0;
22 }
```

2.3. Konstruktoren

Bei der Deklaration von Variablen haben diese anschließend einen nicht bekannten Wert. Um dem abzuhelfen, können Variablen initialisiert werden, indem hinter dem Variablennamen ein definierter Wert geschrieben wird.

```
int i;      // Variable i hat irgendeinen Wert
int j = 0; // Initialisierung der Variable j mit 0
```

Bei der Initialisierung handelt es sich um **keine** Zuweisung!

Ganz ähnlich sieht es bei Objekten aus. Die Eigenschaften eines Objektes haben nach dessen Definition irgendeinen Wert. Auch Objekte können "initialisiert" werden. Dazu werden **Konstruktoren** verwendet.

Ein Konstruktor ist eine Methode (also eine Funktion innerhalb einer Klasse), die den gleichen Namen wie die Klasse hat. Sie wird automatisch beim Definieren eines Objektes der entsprechenden Klasse aufgerufen. Mit ihr können (müssen aber nicht) die Eigenschaften des Objektes initialisiert werden. Konstruktoren haben grundsätzlich keine Rückgabetypen.

In dem obigen Beispiel mit der Ampel wird nun ein Konstruktor `Ampel()` hinzugefügt.

kap02_ampel02.h

```
01 #ifndef ampel_h
02   #define ampel_h ampel_h
03
04    class Ampel
05    {
06       private:
07          enum Licht {aus, ein};
08          Licht Rot, Gelb, Gruen; // lokale Eigenschaften
09
10       public:                    // globale Methoden
11          Ampel();                // Konstruktor
12          void schalteRot();
13          void schalteRotGelb();
14          void schalteGruen();
15          void schalteGelb();
16          int istRotAn();
17          int istGelbAn();
18          int istGruenAn();
19    };
20
21 #endif // ampel_h
```

Natürlich muss auch ein Konstruktor implementiert werden, nämlich dort, wo auch die anderen Methoden implementiert sind. In unserem Beispiel ist dies in der Datei kap02_ampel02.cpp.

kap02_ampel02.cpp

```
01 #include "kap02_ampel02.h"
02
03 Ampel::Ampel()     // Konstruktor-Implementation,
04                     // KEIN Rueckgabetyp!
05 {  schalteRot(); }
06
07 void Ampel::schalteRot()
08 {  Rot = ein; Gelb = Gruen = aus; }
```

```
09
10 void Ampel::schalteRotGelb()
11 {  Rot = Gelb = ein; Gruen = aus; }
12
13 void Ampel::schalteGruen()
14 {  Rot = Gelb = aus; Gruen = ein; }
15
16 void Ampel::schalteGelb()
17 {  Rot = aus; Gelb = ein; Gruen = aus; }
18
19 int Ampel::istRotAn()
20 {  return (Rot == ein); }
21
22 int Ampel::istGelbAn()
23 {  return (Gelb == ein); }
24
25 int Ampel::istGruenAn()
26 {  return (Gruen == ein); }
```

Im Hauptprogramm dagegen ändert sich nichts. An der Stelle, an der die Ampel-Objekte A1 und A2 definiert werden, wird automatisch für jedes Objekt der Konstruktor aufgerufen und damit die Ampeln auf Rot geschaltet.

kap02_02.cpp

```
01 #include <stdio.h>
02 #include "kap02_ampel02.h"
03
04 int main()
05 {
06    Ampel A1, A2;     // an dieser Stelle werden die Konstruktoren
07                      // A1.Ampel() und A2.Ampel()
08                      // automatisch aufgerufen!
09
10    A1.schalteRot(); // Aufruf der Methode ueber das Objekt
11    A2.schalteGruen();
12
13    printf("Ampel 1:\n");
14    printf("Rot  : %i, ", A1.istRotAn());
15    printf("Gelb : %i, ", A1.istGelbAn());
16    printf("Gruen: %i\n", A1.istGruenAn());
17    printf("Ampel 2:\n");
18    printf("Rot  : %i, ", A2.istRotAn());
19    printf("Gelb : %i, ", A2.istGelbAn());
20    printf("Gruen: %i\n", A2.istGruenAn());
21
22    return 0;
23 }
```

Dieser Konstruktor hat keine Parameter. Er wird deshalb auch **Standardkonstruktor** genannt. Wird kein Konstruktor deklariert und definiert, wird automatisch vom Compiler ein Standardkonstruktor eingerichtet. Dies wird auch **implizite Konstruktordeklaration** genannt. Der vom Compiler eingerichtete Standardkonstruktor setzt alle Eigenschaften des Objektes auf null (also entweder auf die Zahl 0 oder auf den `NULL`-Zeiger). Auf die anderen Konstruktoren wird erst später eingegangen.

2.4. *Destruktoren*

Destruktoren dienen dazu, Aufräumarbeiten für nicht mehr benötigte Objekte zu leisten. Wie bei den Konstruktoren wird vom Compiler ein Destruktor eingerichtet, wenn keiner angegeben wird (**implizite Destruktordeklaration**). Sie werden automatisch am Ende des Gültigkeitsbereiches des Objektes aufgerufen und haben häufig den Zweck, den benutzten Speicherbereich (bei dynamischer Speicherverwaltung) wieder freizugeben.

Destruktoren haben keine Parameter und auch keine Rückgabetypen. Der Name eines Destruktors ist der des Konstruktors (also der Name der Klasse) mit einer vorangestellten Tilde ("~"; manchmal auch Schlange genannt). Im Folgenden wird ein Destruktor für das Beispiel der Ampel deklariert.

 *kap02_ampel03.h*

```
01 #ifndef ampel_h
02   #define ampel_h ampel_h
03
04    class Ampel
05    {
06       private:
07          enum Licht {aus, ein};
08          Licht Rot, Gelb, Gruen; // lokale Eigenschaften
09
10       public:                    // globale Methoden
11          Ampel();                // Konstruktor
12          ~Ampel();               // Destruktor
13          void schalteRot();
14          void schalteRotGelb();
15          void schalteGruen();
16          void schalteGelb();
17          int istRotAn();
18          int istGelbAn();
19          int istGruenAn();
20    };
21
22 #endif // ampel_h
```

Die Definition des Destruktors erfolgt parallel zu der des Konstruktors.

 *kap02_ampel03.cpp*

```
01 #include <stdio.h>
02 #include "kap02_ampel03.h"
03
04 Ampel::Ampel()     // Konstruktor-Implementation,
05                    // KEIN Rueckgabetyp!
06 {  schalteRot(); }
07
08 Ampel::~Ampel()    // Destruktor-Implementation,
09                    // KEIN Rueckgabetyp!
10 {  printf("Objekt wird freigegeben!\n"); }
11
12 void Ampel::schalteRot()
13 {  Rot = ein; Gelb = Gruen = aus; }
14
15 void Ampel::schalteRotGelb()
16 {  Rot = Gelb = ein; Gruen = aus; }
17
18 void Ampel::schalteGruen()
19 {  Rot = Gelb = aus; Gruen = ein; }
```

```
20
21 void Ampel::schalteGelb()
22 {  Rot = aus; Gelb = ein; Gruen = aus; }
23
24 int Ampel::istRotAn()
25 {  return (Rot == ein); }
26
27 int Ampel::istGelbAn()
28 {  return (Gelb == ein); }
29
30 int Ampel::istGruenAn()
31 {  return (Gruen == ein); }
```

Das Hauptprogramm braucht nicht verändert zu werden, da die Destruktoren automatisch am Ende des Gültigkeitsbereiches der Objekte aufgerufen werden.

kap02_03.cpp

```
01 #include <stdio.h>
02 #include "kap02_ampel03.h"
03
04 int main()
05 {
06    Ampel A1,A2;      // an dieser Stelle werden die Konstruktoren
07                      // A1.Ampel() und A2.Ampel()
08                      // automatisch aufgerufen!
09
10    A1.schalteRot(); // Aufruf der Methode ueber das Objekt
11    A2.schalteGruen();
12
13    printf("Ampel 1:\n");
14    printf("Rot  : %i, ", A1.istRotAn());
15    printf("Gelb : %i, ", A1.istGelbAn());
16    printf("Gruen: %i\n", A1.istGruenAn());
17    printf("Ampel 2:\n");
18    printf("Rot  : %i, ", A2.istRotAn());
19    printf("Gelb : %i, ", A2.istGelbAn());
20    printf("Gruen: %i\n", A2.istGruenAn());
21
22    return 0;         // hier werden die Destruktoren der Objekte
23 }                    // automatisch aufgerufen!
```

2.5. *Statische Klassenelemente*

Statische Klassenelemente werden noch mal unterschieden zwischen statischen Datenelementen und statischen Methoden.

Statische Datenelemente

Statische Datenelemente entsprechen im Prinzip den globalen Variablen unter Verwendung der Kapselungseigenschaften von Klassen. Sie existieren immer genau einmal, unabhängig davon, ob und wie oft die Klasse instanziiert wird. Sie beinhalten damit Daten, die allen Objekten dieser Klasse gemeinsam sind.

Ein statisches Datenelement wird nicht im Konstruktor sondern in einer dem Projekt zugehörigen Datei (meistens die Datei, in der die Methoden definiert sind) im globalen Bereich am Beginn der Datei unter Bezugnahme auf Klasse und Dateityp initialisiert. Der Konstruktor eignet sich nicht für die Initialisierung, da dieser bei jedem neuen Objekt aufgerufen wird. Auch innerhalb der Klassendefinition kann die Initialisierung nicht erfolgen, da hier nur konstante Werte initialisiert werden dürfen.

Beispiel:

kap02_04.cpp

```
01 #include <stdio.h>
02
03 class Test
04 {
05    public:
06       static int i;
07 };
08
09 int Test::i = 12;
10
11 int main()
12 {
13    Test T1, T2;
14
15    printf("%i\n", T1.i);
16    T2.i = 27;
17    printf("%i\n", Test::i);
18
19    return 0;
20 }
```

Wie im Beispiel gezeigt, ist es egal, über welches Objekt auf das statische Datenelement zugegriffen wird. Es kann sogar ohne Objekte darauf gegriffen werden; dazu muss nur der Klassenname und der Bereichsoperator verwendet werden.

Statische Methoden

Neben statischen Datenelementen können auch Methoden statisch definiert werden. Da diese aber kein aktuelles Objekt kennen (sie gehören ja allen Objekten), darf in statischen Methoden nur auf statische Komponenten zugegriffen werden. Statische Methoden werden z.B. eingesetzt, wenn für alle Objekte bzw. von außerhalb Zugriff auf statische Daten einer Klasse benötigt wird.

Beispiel:

kap02_05.cpp

```
01 #include <stdio.h>
02
03 class Test
04 {
05       static int Anzahl;
06    public:
07       Test();
08       ~Test();
09       static int GetAnzahl() { return Anzahl; };
10 };
11
12 int Test::Anzahl = 0; // am Anfang gibt es 0 Objekte
13
14 Test::Test()
15 {  Test::Anzahl++; }  // Anzahl der Objekte um eins erhoehen
16
17 Test::~Test()
18 {  Test::Anzahl--; }  // Anzahl der Objekte um eins verringern
19
20 int main()
21 {
```

```
22    Test T1, T2, T3;
23
24    printf("Anzahl der Objekte: %i\n", Test::GetAnzahl());
25
26    return 0;
27 }
```

2.6. *Read-Only-Methoden*

Eine Methode, die nur lesend auf die Eigenschaften der Klasse zugreift, kann zu einer **Read-Only-Methode** umgewandelt werden. Diese Methoden werden besonders gekennzeichnet, indem das Schlüsselwort `const` an den Funktionskopf (bei Deklaration und Definition) angehangen wird.

Beispiel:

kap02_06.cpp

```
01 #include <stdio.h>
02
03 class Punkt
04 {
05       float x, y;
06    public:
07       Punkt(float , float );
08       void set(float , float );
09       float get_x() const; // Read-Only-Methoden
10       float get_y() const;
11 };
12
13 Punkt::Punkt(float x_value, float y_value)
14 {
15    set(x_value, y_value);
16 }
17
18 void Punkt::set(float x_value, float y_value)
19 {
20    x = x_value;
21    y = y_value;
22 }
23
24 float Punkt::get_x() const
25 {
26    return x;
27 }
28
29 float Punkt::get_y() const
30 {
31    return y;
32 }
33
34 int main()
35 {
36    Punkt P1(3.1, 4.7);
37    float const x_Wert = P1.get_x();
38    float const y_Wert = P1.get_y();
39
40    printf("Punkt P1(%3.1f/%3.1f)\n", x_Wert, y_Wert);
41
```

```
42     return 0;
43 }
```

Durch die Verwendung von Read-Only-Methoden wird vom Compiler geprüft, ob in diesen Methoden die Eigenschaften der Klasse verändert werden. Es wird auch geprüft, ob andere Methoden aufgerufen werden, die keine Read-Only-Methoden sind. Die folgende Änderung des obigen Beispiels würde vom Compiler nicht übersetzt werden.

```
24 float Punkt::get_x() const
25 {
26     x = 0.0;          // Änderung der Eigenschaft nicht erlaubt!
27     set(1.0, 2.0);    // Aufruf von Methoden ohne const nicht erlaubt!
28     return x;
29 }
```

Eine Ausnahme bilden Eigenschaften, die mit dem Schlüsselwort `mutable` definiert werden; diese Eigenschaften können nämlich auch in Read-Only-Methoden verändert werden. Wird in dem obigen Beispiel die Eigenschaften `x` und `y` wie folgt definiert,

```
05         mutable float x, y;
```

dann wäre die direkte Veränderung der Eigenschaft `x` korrekt; der Aufruf der `set`-Methode dagegen ist auch mit `mutable` nicht erlaubt.

```
24 float Punkt::get_x() const
25 {
26     x = 0.0;          // mit mutable korrekt!
27     set(1.0, 2.0);    // auch mit mutable nicht erlaubt!
28     return x;
29 }
```

Durch das Verwenden einer Read-Only-Methode kann diese nicht mehr für veränderbare Variablen verwendet werden. Da das Schlüsselwort `const`, das die Methode zur Read-Only-Methode macht, mit in die Signatur der Methode einfließt, können zwei Versionen der Methode erstellt (also deklariert und definiert) werden - einmal mit und einmal ohne `const`. Damit können die Ergebnisse dieser Methode veränderbaren und unveränderbaren Variablen zugewiesen werden.

```
24 float Punkt::get_x()
25 {
26     return x;
27 }
28
29 float Punkt::get_x() const
30 {
31     return x;
32 }
```

2.7. Friends

In einigen Fällen muss eine Klasse einer Funktion oder anderen Klassen den Zugriff auf ihre privaten Daten erlauben. Dies kann z.B. aus Performancegründen der Fall sein, wenn eine Klasse rechenintensive Aufgaben mit Daten einer anderen Klasse durchführen muss.

Um einer Funktion (die also nicht zu einer Klasse gehört) Zugriff auf die privaten Daten zu erlauben, muss diese Funktion innerhalb der Klasse mit dem Schlüsselwort `friend` deklariert werden.

Beispiel:

kap02_07.cpp

```
01 #include <stdio.h>
02
```

```
03 class Test
04 {
05       int Num;
06       static int StaticNum;
07    public:
08       friend void resetNum(Test &T, int N, int SN);
09       void printNum();
10 };
11
12 int Test::StaticNum = 0;
13
14 void resetNum(Test &T, int N, int SN)
15 {
16    T.Num = N;
17    T.StaticNum = SN;
18 }
19
20 void Test::printNum()
21 {
22    printf("Num       = %i\n", Num);
23    printf("StaticNum = %i\n", StaticNum);
24 }
25
26 int main()
27 {
28    Test TestObject;
29
30    resetNum(TestObject, 3, 5);
31    TestObject.printNum();
32
33    return 0;
34 }
```

Damit die Funktion `resetNum` auf die Daten der Klasse `Test` zugreifen kann, muss eine Referenz auf ein Objekt der Klasse übergeben werden. Die Funktion `resetNum` hat nun die gleichen Zugriffsrechte wie eine Methode der Klasse. Innerhalb dieser Funktion muss jedoch immer über die Referenz (oder einen Zeiger) auf die Daten der Klasse zugegriffen werden!

Es können auch Methoden von Klassen oder gleich ganze Klassen als `friend` deklariert werden. Wird eine ganze Klasse als `friend` deklariert, sind automatisch alle Methoden dieser Klasse als `friend`-Methoden zu betrachten.

Beispiel:

kap02_08.cpp

```
01 #include <stdio.h>
02
03 class Test3; // Vorwaertsdeklaration
04
05 // Die komplette Klasse Test1 soll ein Freund von Klasse Test3 werden.
06 class Test1
07 {
08    public:
09       void Methode_von_Test1(Test3 *Zeiger_auf_Test3);
10 };
11
12 class Test2
13 {
14    public:
```

```
15        // Die folgende Methode soll ein Freund von Klasse Test3 werden.
16        void Methode_von_Test2(Test3 &Zeiger_auf_Test3);
17 };
18
19 class Test3
20 {
21        int Zahl_von_Test3; // private Eigenschaft!
22
23        // Um eine ganze Klasse zum Freund zu machen:
24        friend class Test1;
25        // Um nur eine Methode zum Freund zu machen:
26        friend void Test2::Methode_von_Test2(Test3 &Zeiger_auf_Test3);
27     public:
28        Test3()
29        {
30           Zahl_von_Test3 = 3;
31           printf("Konstruktor von Test3:\n");
32           printf("Zahl_von_Test3 auf %i gesetzt!\n\n", Zahl_von_Test3);
33        }
34 };
35
36 void Test1::Methode_von_Test1(Test3 *Zeiger_auf_Test3)
37 {
38    printf("class Test1:\n");
39    // Jetzt wird auf die private Eigenschaft von Test3 zugegriffen:
40    printf("Zahl_von_Test3: %i\n\n", Zeiger_auf_Test3->Zahl_von_Test3);
41 }
42
43 void Test2::Methode_von_Test2(Test3 &Zeiger_auf_Test3)
44 {
45    printf("class Test2:\n");
46    // Jetzt wird auf die private Eigenschaft von Test3 zugegriffen:
47    printf("Zahl_von_Test3: %i\n\n", Zeiger_auf_Test3.Zahl_von_Test3);
48 }
49
50 int main()
51 {
52    Test3 T3;
53    Test2 T2;
54    Test1 T1;
55
56    T1.Methode_von_Test1(&T3);
57    T2.Methode_von_Test2(T3);
58
59    return 0;
60 }
```

Der Einsatz von `friend` sollte wirklich nur dort erfolgen, wo er zwingend notwendig ist, da er die Kapselung innerhalb einer Klasse nach außen aufbricht und so wieder Sicherheitslücken entstehen können. Es gibt nur wenige Situationen, in denen keine andere Möglichkeit besteht.

2.8. this-Zeiger

In jeder Klassendefinition wird unsichtbar ein Zeiger auf die Klasse deklariert. Dieser Zeiger hat immer den Namen `this`. Z.B. für die Klasse `Test` wird automatisch der Zeiger `Test *const this` deklariert. Dieser Zeiger kann innerhalb der Klasse - außer in statischen Methoden - eingesetzt werden. Er wird allen

Methoden einer Klasse unsichtbar als weiterer Parameter übergeben, so dass die Methoden auch wissen, mit welchen Daten sie arbeiten.

Beispiel:

kap02_09.cpp

```
01 #include <stdio.h>
02
03 class Test
04 {
05       int Num;
06    public:
07       void setNum(int Num) {  this->Num = Num;                  }
08       void printNum()      {  printf("Num = %i\n", this->Num);  }
09       Test *getPointer()   {  return this;                      }
10 };
11
12 int main()
13 {
14    Test T;
15    Test *TPointer;
16
17    T.setNum(5);
18    T.printNum();
19
20    T.setNum(7);
21    TPointer = T.getPointer(); // TPointer zeigt jetzt auf T!
22    TPointer->printNum();
23
24    return 0;
25 }
```

`this` ist ein Schlüsselwort und kann deshalb in den eigenen Programmen als Bezeichner nicht eingesetzt werden.

2.9. Methodenzeiger

Ähnlich wie die Zeiger auf Funktionen (siehe Kapitel 9.5 im Skript "Programmieren in C") können in Objektorientierten Programmiersprachen auch Zeiger auf Methoden einer Klasse definiert werden. Dabei wird allgemein zwischen gebundenen und ungebundenen Methodenzeigern unterschieden.

Bei einem **gebundenen Methodenzeiger** (engl. *bound method*) wird bei der Zuweisung des Zeigers zusätzlich auch das Objekt festgelegt, auf dessen Methode gezeigt werden soll. D.h. der Methodenzeiger ist an das Objekt gebunden.

Dagegen zeigt ein **ungebundener Methodenzeiger** (engl. *unbound method*) nur auf die Methode einer Klasse, ohne dass der Zeiger an ein Objekt gebunden ist. Dafür muss beim Aufruf des Methodenzeigers das entsprechende Objekt angegeben werden.

In C++ gibt es nur ungebundene Methodenzeiger. Dies ist daher möglich, weil jede Methode einer Klasse nur einmal existiert - unabhängig davon, wie viele Objektinstanzen von der Klasse erzeugt wurden. Beim Aufruf der Methode wird dann mit Hilfe des `this`-Zeigers festgelegt, für welches Objekt die Methode aufgerufen wird.

Das folgende Programm gibt die Adresse einer Methode für mehrere Objekte einer Klasse aus. Dabei ist zu sehen, dass es immer die gleiche Adresse ist. Gleichzeitig zeigt dieses Programm, dass es möglich ist, mit Hilfe eines Methodenzeigers eine private Methode aufzurufen.

kap02_10.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 #define MAX 5
06
07 class K
08 {
09    private:
10       void PrivateMethode()
11       {
12          cout << "Adresse der privaten Methode: "
13               << (void *) &K::PrivateMethode << endl;
14       }
15    public:
16       typedef void (K::*MethPtr)();
17       MethPtr getPrivateMethode()
18       {
19          return &K::PrivateMethode;
20       }
21 };
22
23 int main()
24 {
25    K KArray[MAX];
26    int i;
27    K::MethPtr mptr = nullptr;
28
29    for (i = 0; i < MAX; i++)
30    {
31       mptr = KArray[i].getPrivateMethode();
32       (KArray[i].*mptr)();
33    }
34
35    return 0;
36 }
```

Anders als bei den Zeigern auf Funktionen, bei denen der Name der Funktion (ohne die anschließenden runden Klammern) ein Zeiger auf die Funktionen ist, muss beim Setzen eines Methodenzeigers immer der Adressoperator und der Klassenname mit angegeben werden, z.B. `return &K::PrivateMethode;`. Beim Aufruf der Methode wird der `.*`-Operator zwischen dem Objektnamen und dem Methodenzeiger verwendet (einer der wenigen Operatoren, die nicht überladen werden können!).

Hier noch ein weiteres Beispielprogramm:

kap02_11.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 class Auto
06 {
07       double Geschw;
08       bool MotorAn;
09    public:
10       Auto(): Geschw(0.0), MotorAn(false) { }
11       void starten()
12       {
13          cout << "Motor starten ... " << endl;
```

```
14          MotorAn = true;
15          cout << "Motor ist gestartet" << endl;
16       }
17       void beschleunigen(double Beschl, int Sek)
18       {
19          double f = Geschw;
20
21          if (MotorAn)
22          {
23             cout << "beschleunigen von " << Geschw << "km/h ... ";
24             for ( ; f < Geschw + (Beschl * Sek); f += 1.0)
25                ;
26             Geschw = f;
27             cout << "auf " << Geschw << "km/h" << endl;
28          }
29          else
30             cout << "bitte erst den Motor starten!" << endl;
31       }
32       void bremsen(double Verz, int Sek)
33       {
34          double f = Geschw;
35
36          if (MotorAn)
37          {
38             cout << "bremsen von " << Geschw << "km/h ... ";
39             for ( ; f >= Geschw - (Verz * Sek); f -= 1.0)
40                if (f < 0.0)
41                {
42                   f = 0.0;
43                   break;
44                }
45             Geschw = f;
46             cout << "auf " << Geschw << "km/h" << endl;
47          }
48          else
49             cout << "bitte erst den Motor starten!" << endl;
50       }
51       void ausschalten()
52       {
53          if (MotorAn)
54          {
55             cout << "Motor ausschalten ... " << endl;
56             while (Geschw > 0.0)
57                bremsen(2.0, 1);
58             MotorAn = false;
59             cout << "Motor ist ausgeschaltet" << endl;
60          }
61       }
62 };
63
64 int main() {
65    Auto a;
66    void (Auto::*mz1)() = &Auto::starten;
67    void (Auto::*mz2)(double, int) = &Auto::beschleunigen;
68
69    (a.*mz1)();
70    (a.*mz2)(3.5, 10);
71
```

```
72      mz2 = &Auto::bremsen;
73      (a.*mz2)(2.1, 10);
74      mz1 = &Auto::ausschalten;
75      (a.*mz1)();
76
77      return 0;
78  }
```

Der Typ eines Methodenzeigers ergibt sich aus der Signatur der Methode. D.h. ein Zeiger auf eine Methode kann nicht auf eine andere Methode mit einer anderen Signatur zeigen. Daher können im letzten Beispiel der Zeiger `mz1` auf die Methoden `starten` und `ausschalten` sowie der Zeiger `mz2` auf die Methoden `beschleunigen` und `bremsen` zeigen, da diese jeweils die gleichen Signaturen haben.

2.10. Designfehler

Das Auffinden der geeigneten Klassen und die richtige Verteilung der Aufgaben ist nicht einfach, und es gibt leider für diesen Vorgang auch keine formalen Regeln. Im Folgenden werden daher einige typische Designfehler beschrieben.

Klassen modifizieren direkt Daten von anderen Klassen.	Das Prinzip der Kapselung wird verletzt. Solche Klassen sind kaum wiederverwendbar, kaum wartbar und so entwickelte Produkte werden undurchschaubar. Jede Klasse sollte daher Schnittstellenfunktionen anbieten, die den Zugriff auf die Daten der Klasse realisieren.
Klassen werden zu komplex gestaltet.	Wird eine Klasse zu komplex, so ist sie nicht mehr überschaubar. Es ist in diesen Fällen besser, Hilfsklassen zu definieren und mit Vererbung zu arbeiten. Dies ist auch für die Fehlersuche hilfreich.
Klassen ohne Aufgaben	Die bloße Existenz einer Klasse, die jedoch keine eigenen Aufgaben übernimmt, ist in der Regel keine eigene Klasse wert. Ausnahmen sind jedoch die abstrakten Basisklassen, die ja als Vorlage für weitere Klassen dienen.
Klassen mit nicht beanspruchten Aufgaben	Eine Klasse sollte nicht "auf Verdacht hin" definiert werden. Dadurch entsteht unnötiger Ballast, der das Durchschauen einer Klasse und damit den Umgang mit ihr unnötig erschwert.
Irreführende Namen	Der Name einer Klasse (und natürlich auch seiner Methoden und Eigenschaften) sollte so gewählt werden, dass er den Zweck der Klasse beschreibt. In großen Klassenbibliotheken beginnen alle Klassennamen mit einem bestimmten Buchstaben bzw. einer standardisierten Buchstabenkombination.
Eine Klasse besitzt unterschiedliche Aufgaben ohne Zusammenhang.	Alle Methoden einer Klasse sollten der Lösung eines Problems dienen und nicht völlig verschiedene Aufgaben erledigen.
Fehlender Einsatz von Vererbung	Vererbung sollte dort eingesetzt werden, wo eine "ist ein"-Beziehung zwischen Klassen besteht. Abgeleitete Klassen sollten die Basisklassen erweitern bzw. Funktionalitäten der Basisklassen zusammenführen. Das Zusammenführen von Klassen, die nicht zusammenhängende Aufgaben erfüllen, widerspricht der Vererbung.
Duplizierte Funktionalität	Tritt eine Methode bzw. ein Programmcode in verschiedenen Klassen in gleicher Form auf, so kann das ein Hinweis dafür sein, dass eine gemeinsame Oberklasse (Basisklasse) fehlt.

3. Vererbung

3.1. Einführung

Häufig ist es der Fall, dass zwei oder mehrere Klassen einen gemeinsamen Kern besitzen. Über den Mechanismus der **Vererbung** wird dieser gemeinsame Kern in einer eigenen Klasse untergebracht. Von dieser **Oberklasse** (die oberste Oberklasse wird auch **Basisklasse** genannt) werden dann die anderen Klassen abgeleitet (**Unterklassen** oder **abgeleitete Klassen**). In den abgeleiteten Klassen können neue Elemente hinzugefügt oder bestehende verändert werden.

Beispiel:

In einem Programm wurde eine Klasse zur Verwaltung eines Datums entwickelt. Diese speichert genau ein Datum ab und besitzt Methoden, das Datum festzulegen und auszugeben.

kap03_01.cpp

```
01 #include <stdio.h>
02
03 class TDate
04 {
05    public:
06       int Day, Month, Year;
07
08       void setDate(int d, int m, int y)
09       {  Day = d; Month = m; Year = y;  }
10       void printDate()
11       {  printf("%02i.%02i.%04i\n", Day, Month, Year);  }
12 };
13
14 int main()
15 {
16    TDate Datum;
17
18    Datum.setDate(29, 2, 2000);
19    Datum.printDate();
20
21    return 0;
22 }
```

An einer anderen Stelle des Programms wird ebenfalls eine Klasse zur Verwaltung eines Datum-Wertes benötigt. Diese unterscheidet sich jedoch in bestimmten Details von der eben entwickelten Klasse `TDate`:

1. Ein Datum soll über die Methode `printDate` nicht in der Form `TT.MM.JJJJ`, sondern in amerikanischer Form (`MM/TT/JJJJ`) angezeigt werden.
2. Es wird eine zusätzliche Methode `isLeapYear` benötigt, die eine `1` zurückliefert, wenn es sich bei dem aktuellen Datum um ein Schaltjahr handelt, und ansonsten eine `0`.

Um diese zwei Zusätze zu realisieren, gibt es nun verschiedene Möglichkeiten:

- Die bestehende Klasse `TDate` kann um zwei Methoden erweitert werden, nämlich einer Methode, bei der das Datum im amerikanischen Format ausgegeben wird (z.B. `printAmericanDate`), und einer Methode, die als Ergebnis zurückliefert, ob es sich bei dem aktuellen Jahr um ein Schaltjahr handelt oder nicht. In diesem Beispiel ist eine derartige Erweiterung durchaus vorstellbar und vielleicht auch sinnvoll. Eine größere Klasse hingegen, die vielleicht in mehreren Details angepasst werden müssen, wird sehr stark aufgebläht und unübersichtlich. Für grundsätzlich gleiche Methoden, die sich nur in Details unterscheiden (wie in diesem Bespiel die Ausgabe), müssen verschiedene Namen vergeben werden, oder es werden zusätzliche Parameter eingeführt, die dieses Details berücksichtigen. Das Erweitern einer bestehenden Klasse hat weiterhin den Nachteil, dass in bestehenden, (hoffentlich) fehlerfreien

Programmcode eingegriffen wird und so die Gefahr besteht, neue Fehler einzubauen. Der bestehende Programmcode muss ferner nochmals übersetzt werden.

- Alle Deklarationen und Definitionen der bestehenden Klasse werden kopiert, umbenannt (z.B. auf `TAmericanDate`) und anschließend angepasst. Diese Vorgehensweise hat den Nachteil, dass viele Definitionen und Deklarationen von unverändert Übernommenem im Programm letztendlich doppelt vorkommen. Das Programm wird dadurch in seiner Summe wesentlich größer, Änderungen von eigentlich gemeinsamen Programmteilen müssen mehrfach vorgenommen werden und natürlich wird dadurch auch unnötig Speicherplatz verbraucht.
- Da derartige Aufgaben bei einer objektorientierten Vorgehensweise häufig vorkommen, bietet C++ eine sehr gute Möglichkeit, neue Klassen aus bestehenden abzuleiten. Dieser Mechanismus wird **Vererbung** genannt. Hierbei wird in den Code der Oberklasse nicht eingegriffen. Es muss nicht einmal der Quellcode der Oberklasse vorliegen, lediglich die Deklarationen müssen zur Verfügung stehen.

Im folgenden soll die gestellte Aufgabe durch das Ableiten einer neuen Klasse `TAmericanDate` aus der bestehenden `TDate` gelöst werden:

kap03_02.cpp

```
01 #include <stdio.h>
02
03 class TDate
04 {
05    public:
06       int Day, Month, Year;
07
08       void setDate(int d, int m, int y)
09       {  Day = d; Month = m; Year = y;  }
10       void printDate()
11       {  printf("%02i.%02i.%04i\n", Day, Month, Year);  }
12 };
13
14 class TAmericanDate: public TDate
15 {
16    public:
17       void printDate()
18       {  printf("%02i/%02i/%04i\n", Month, Day, Year);  }
19       int isLeapYear()
20       {  return !(Year % 400) || (!(Year % 4) && Year % 100));  }
21 };
22
```

Die in `TAmericanDate` deklarierten Methoden sind jene, die im Vergleich zur Klasse `TDate` verändert oder neu hinzugefügt werden müssen. Sie werden wie üblich definiert, wobei alle `public`-Klassenkomponenten der Oberklasse `TDate` wie Elemente der eigenen Klasse verwendet werden können. Die Klasse `TAmericanDate` kann jetzt wie die Klasse `TDate` verwendet werden:

```
23 int main()
24 {
25    TDate D;
26    TAmericanDate AD;
27
28    D.setDate(29, 2, 2000);
29    AD.setDate(29, 2, 2000);
30    D.printDate();
31    AD.printDate();
32    if (AD.isLeapYear())
33       printf("Dieses Jahr ist ein Schaltjahr!\n");
34
```

```
35     return 0;
36 }
```

3.2. Syntax der Vererbung

Die Syntax der Vererbung lautet wie folgt:

```
class Unterklasse: [Zugriffattribut] Oberklasse
{
   // Deklarationen
};
```

Die neue Klasse wird wie bisher deklariert. Dem Klassennamen folgt ein Doppelpunkt. Hinter dem Doppelpunkt kann eine Oberklasse angegeben werden. Vor dem Namen der Oberklasse kann ein Zugriffsattribut angegeben werden. Folgende Zugriffsattribute sind erlaubt: `public`, `private` und `protected`. Wird das Zugriffsattribut weggelassen, wird `private` angenommen. Näheres zu den Zugriffsattributen kommt im nächsten Abschnitt.

Wichtig: Konstruktoren und Destruktoren werden nicht vererbt und müssen für die abgeleitete Klasse neu definiert werden.

3.3. Art der Ableitung

Grundsätzlich werden beim Ableiten alle Komponenten der Oberklasse (Eigenschaften und Methoden) übernommen. Wesentlich für den Zugriff auf diese Elemente sind die in der Oberklasse verwendeten **Zugriffsattribute** (`public`, `private` und `protected`). Dieser Mechanismus wird auch **Zugriffsschutz** genannt.

Damit auf die Interna (also die privaten Komponenten) einer Klasse auch nicht durch einfaches Ableiten in eine neue Klasse zugegriffen werden kann, werden zwar auch private Klassenelemente übernommen, es besteht jedoch keine Zugrifferlaubnis – die privaten Klassenelemente sind im Prinzip unsichtbar. Private Elemente der Oberklasse können somit in der abgeleiteten Klasse nur durch entsprechende `public`- oder `protected`-Methoden der Oberklasse manipuliert werden. Somit gilt nach wie vor, dass auf `private`-Elemente ausschließlich die Methoden der Klasse selber und als `friend` deklarierte Funktionen oder Klassen zugreifen können.

Das bisher noch nicht benötigte Zugriffsattribut `protected` wird jetzt im Zusammenhang mit den Zugriffsrechten bei der Vererbung benötigt. Je nach vorhandenen Zugriffsattribut in der Oberklasse und verwendeten Zugriffattribut bei der Ableitung erhält man folgende Zugriffsattribute in der abgeleiteten Klasse:

		Zugriffsattribut bei Ableitung		
		`private`	`protected`	`public`
Oberklasse	`private`	**kein Zugriff**	**kein Zugriff**	**kein Zugriff**
	`protected`	`private`	`protected`	`protected`
	`public`	`private`	`protected`	`public`

Eine abgeleitete Klasse kann somit nur auf die `public`- und `protected`-Elemente der Oberklasse zugreifen.

Öffentliches Ableiten von Klassen

Wird eine Klasse `public` von einer Oberklasse abgeleitet, so muss bei der Klassendeklaration vor der Oberklasse das Schlüsselwort `public` stehen.

```
class Neu: public Oberklasse
{
   // Deklarationen
};
```

Klassenkomponenten der Oberklasse werden mit ihren Zugriffattributen wie oben angegeben vererbt. Das öffentliche Ableiten von Klassen ist somit sinnvoll, wenn in der abgeleiteten Klasse grundsätzlich die gleiche Schnittstelle (`public`-Deklarationen) bestehen soll. Natürlich können aber Komponenten neu hinzukommen oder verändert werden, indem sie für die abgeleitete Klasse neu deklariert und definiert werden.

Privates Ableiten von Klassen

Wird eine Klasse privat von einer Oberklasse abgeleitet, so kann bei der Klassendeklaration vor dem Namen der Oberklasse das Schlüsselwort `private` angegeben werden, was aber nicht zwingend erforderlich ist, da es die Standard-Vorgabe ist.

```
class Neu: [private] Oberklasse
{
   // Deklarationen
};
```

Klassenkomponenten der Oberklasse werden mit ihren Zugriffsattributen wie oben angegeben vererbt. Das private Ableiten von Klassen ist dann sinnvoll, wenn keine (oder nur wenige) Teile der Schnittstelle der Oberklasse (`public`-Deklarationen) auch in der abgeleiteten Klasse zur Schnittstelle gehören sollen, wenn also eine neue Schnittstelle entstehen soll.

Sollen einzelne `public`-Elemente der Oberklasse in der privat abgeleiteten Klasse ebenfalls wieder `public` sein, kann dies durch explizite Angabe des vollständigen Namens (also `Oberklassenname::Elementname`) im `public`-Bereich der abgeleiteten Klasse erreicht werden.

Beispiel:

Die öffentliche Elementfunktion `setDate` der Klasse `TDate` soll in der abgeleiteten Klasse `TAmericanDate` ebenfalls `public` sein und somit als Schnittstellenfunktion zur Verfügung stehen, obwohl `TAmericanDate` nur `private` abgeleitet wird.

```
class TAmericanDate: TDate    // also private
{
      // ...
   public:
      TDate::setDate;         // Methode ist nun wieder public!
};
```

Ererbte Klassenkomponenten können – sofern die Zugriffserlaubnis besteht – wie eigene verwendet werden. Da jedoch die abgeleitete Klasse auch Namen, die bereits in der Oberklasse definiert wurden, neu definieren darf (d.h. die Komponenten werden **überschrieben**), ist es für den Zugriff auf die entsprechende Oberklassenkomponente manchmal notwendig, mit dem **Bereichsoperator** zu arbeiten.

kap03_03.cpp

```
01 #include <stdio.h>
02
03 class TBasis
04 {
05       int Number;
06    public:
07       TBasis()
08       {  Number = 3;  }
09       void print()
10       {  printf("%i\n", Number);  }
11 };
```

```
12
13 class TAbleitung: public TBasis
14 {
15       int Number;
16    public:
17       TAbleitung()
18       {  Number = 17;  }
19       void print()
20       {
21          printf("Zahl der Basisklasse: ");
22          TBasis::print();
23
24          // folgendes ist falsch
25          // print(); // -> endlose Rekursion!
26
27          // auch folgendes ist falsch
28          // printf("TBasis::Number = %i\n", TBasis::Number);
29          // da TBasis::Number private ist
30
31          printf("TAbleitung::Number = %i\n", Number);
32       }
33 };
34
35 int main()
36 {
37    TBasis B;
38    TAbleitung AB;
39
40    printf("Zahl der Klasse TBasis: ");
41    B.print();
42
43    printf("Ausgabe der print-Methode der Klasse TAbleitung:\n");
44    AB.print();
45
46    return 0;
47 }
```

Wenn im obigen Beispiel die Methode `print()` ohne Angabe der Klasse `TBasis` aufrufen wird, wird die Methode `print()` der abgeleiteten Klasse rekursiv aufgerufen. Dabei entsteht eine Endlosrekursion, die in den meisten Fällen zum Programm- (Stack Overflow) oder Rechnerabsturz führt.

Protected-Ableiten von Klassen

Wenn Sie eine Klasse mit dem Zugriffsattribut `protected` ableiten, werden die `public`-Elemente der Oberklasse zu `protected`-Elementen in der abgeleiteten Klasse.

```
class Neu: protected Oberklasse
{
   // Deklarationen
}
```

Der Einsatz des Zugriffsattributes `protected` macht nur innerhalb der Vererbung Sinn. Gegeben sei z.B. die folgende Situation: Ein Klassenelement gehört grundsätzlich nicht zur Schnittstelle und soll daher nicht `public` deklariert werden. Eine abgeleitete Klasse benötigt aber Zugriffserlaubnis auf das Element, was wiederum eine `public`-Deklaration verlangen würde. Einen Ausweg aus dieser Situation stellt `protected` dar. Klassenmitglieder, welche `protected` deklariert werden, können grundsätzlich wie `private`-Komponenten außerhalb der Klasse nicht angesprochen werden. Sie werden aber mit in die abgeleitete Klasse vererbt und können dort verwendet und aufgerufen werden.

In der Klasse `TDate` wurden `Day`, `Month` und `Year` notwendigerweise `public` deklariert, da die Methoden `isLeapYear` und `printDate` der abgeleiteten Klasse `TAmericanDate` darauf zugreifen können müssen. Im Sinne eines abstrakten Datentyps sollten `Day`, `Month` und `Year` im allgemeinen aber besser nicht von außen zugänglich sein. Besser ist es, diese Daten `protected` zu deklarieren:

```
class TDate
{
   protected:
      int Day, Month, Year;
   public:
      void setDate(int d, int m, int y);
      void printDate();
};
```

Ableiten der Klasse `TAmericanDate` wie zuvor.

Jetzt ist folgendes nicht mehr möglich:

```
TDate d;
...
d.Month = 9; // Fehler, da Month protected!!!
```

3.4. Strukturen und Klassen

Dass Klassen eine Erweiterung von C-Strukturen sind, beweist die Tatsache, dass Klassen auch mit dem Schlüsselwort `struct` definiert werden können. Im Unterschied zur Definition mit `class` ist das Standard-Zugriffsattribut der Strukturkomponenten jedoch `public`. Ferner werden Oberklassen öffentlich übernommen (also `public`), wenn nicht explizit `private` angegeben ist.

Beispiele:

Eine `struct`-Klassendefinition entspricht folgender Definition mit dem Schlüsselwort `class`.

```
struct KLASSE: OBERKLASSE     -->  class KLASSE: public OBERKLASSE
{                                  {
                                      public:
      DEKLARATION1;                      DEKLARATION1;
   private:                           private:
      DEKLARATION2;                      DEKLARATION2;
   protected:                         protected:
      DEKLARATION3;                      DEKLARATION3;
   public:                            public:
      DEKLARATION4;                      DEKLARATION4;
};                                 };
```

Umgekehrt entspricht eine `class`-Definition einer `struct`-Definition wie folgt.

```
class KLASSE: OBERKLASSE    -->    struct KLASSE: private OBERKLASSE
{                                  {
                                      private:
      DEKLARATION1;                      DEKLARATION1;
   private:                           private:
      DEKLARATION2;                      DEKLARATION2;
   protected:                         protected:
      DEKLARATION3;                      DEKLARATION3;
   public:                            public:
      DEKLARATION4;                      DEKLARATION4;
};                                 };
```

3.5. Konstruktoren und Destruktoren abgeleiteter Klassen

Abgeleitete Klassen können genauso wie Oberklassen Kontruktoren und Destruktoren besitzen. Dabei sind zwei Punkte zu beachten.

Aufrufreihenfolge der Konstruktoren und Destruktoren

Grundsätzlich wird für ein Objekt einer abgeleiteten Klasse immer zuerst der **Standard**-Konstruktor der Oberklasse und erst danach der Konstruktor der abgeleiteten Klasse abgearbeitet. Bei den Destruktoren ist die Reihenfolge umgekehrt: Der Destruktor der abgeleiteten Klasse wird vor dem Destruktor der Oberklasse durchgeführt. So kann in allen Methoden einer abgeleiteten Klasse - also auch den Kontruktoren und Destruktoren - immer davon ausgegangen werden, dass alle Daten der Oberklasse bereits bzw. noch existieren und richtig initialisiert sind.

Beispiel:

kap03_04.cpp

```
01 #include <stdio.h>
02
03 class Basis
04 {
05    public:
06       Basis()  {  printf("Basis-Konstruktor\n");  }
07       ~Basis() {  printf("Basis-Destruktor\n");   }
08 };
09
10 class Ableitung: public Basis
11 {
12    public:
13       Ableitung()  {  printf("Ableitung-Konstruktor\n");  }
14       ~Ableitung() {  printf("Ableitung-Destruktor\n");   }
15 };
16
17 int main()
18 {
19    Ableitung A;
20
21    printf("\n");
22
23    return 0;
24 }
```

Dieses Programm erzeugt folgende Ausgabe:

```
Basis-Konstruktor
Ableitung-Konstruktor

Ableitung-Destruktor
Basis-Destruktor
```

Übergabe von Parametern an den Konstruktor einer Oberklasse

Verlangt der Konstruktor einer Oberklasse einen oder mehrere Parameter, oder wollen Sie einen speziellen Konstruktor der Oberklasse einsetzen, müssen diese bei der Definition des Konstruktors der abgeleiteten Klasse in der Initialisierungsliste unter dem Namen der Oberklasse angegeben werden. Diese wird durch einen Doppelpunkt in der Definition des Konstruktors angehängt. Mehrere Konstruktoren von mehreren Oberklassen (siehe Kapitel *Mehrfachvererbung*) werden durch Kommata getrennt.

```
class Basis
{
```

```
   public:
      // Konstruktor der Basisklasse mit Parametern:
      Basis(int i, char *s) { ... };
      // ...
};

class Ableitung: public Basis
{
   public:
      Ableitung(int i);
};

Ableitung::Ableitung(int i): Basis(i, "Test")
{
   // ...
}
```

Eine andere Variante wäre, den Konstruktor gleich in die Klasse `Ableitung` zu definieren, z.B.

```
class Ableitung: public Basis
{
   public:
      Ableitung(int i): Basis(i, "Test") { ... };
};
```

3.6. Kompatibilität in Klassenhierarchien

Zuweisung von Objekten abgeleiteter Klassen

Ein Objekt einer abgeleiteten Klasse kann einem Objekt einer Oberklasse zugewiesen werden. Die Ableitung muss dafür `public` durchgeführt werden. Das entspricht auch der Auffassung, dass ein Objekt einer abgeleiteten Klasse als besondere Form der Oberklasse angesehen werden kann.

Diese Beziehung zwischen abgeleiteter Klasse und Oberklasse ist eine typische *"ist eine"* Beziehung. Zum Beispiel gilt: Ein Auto *ist ein* Fahrzeug (aber nicht umgekehrt!). Das heißt, ein Auto hat zumindest alle Eigenschaften eines Fahrzeugs und noch ein paar mehr.

Die Daten in der abgeleiteten Klasse gehen bei einer solchen Zuweisung verloren, da in der Oberklasse dafür kein Platz ist. Dies ist vergleichbar mit der Zuweisung einer Fließkommazahl an eine ganze Zahl. Auch hierbei geht ein Teil der Informationen (nämlich die Nachkommastellen) der Fließkommazahl verloren. Es existiert also eine Art Typumwandlung "abgeleitete Klasse -> Oberklasse", die bei einer solchen Zuweisung implizit zur Anwendung kommt.

Für den umgekehrten Fall existiert keine Umwandlungsfunktion, so dass eine Zuweisung der Form "abgeleitetes Objekt = Oberobjekt" nicht möglich ist!

kap03_05.cpp

```
01 #include <stdio.h>
02
03 class Oberklasse
04 {
05    protected:
06       int OberNr;
07    public:
08       Oberklasse(int i): OberNr(i) { }
09       void print()
10       {  printf("Oberklasse: OberNr = %i\n", OberNr);  }
11 };
12
```

```
13 class Ableitung: public Oberklasse
14 {
15       int AbleitungNr;
16    public:
17       Ableitung(int i): AbleitungNr (i), Oberklasse(i) { }
18       void print()
19       {
20          printf("Ableitung: OberNr = %i\n", OberNr);
21          printf("Ableitung: AbleitungNr: = %i\n", AbleitungNr);
22       }
23 };
24
25 int main()
26 {
27    Oberklasse O(1);
28    Ableitung A(2);
29
30    O.print();
31    A.print();
32
33    O = A;
34    // A = O; nicht zulässig!
35
36    O.print();
37    A.print();
38
39    return 0;
40 }
```

Zeiger auf Oberklassen und abgeleitete Klassen

Zeiger auf Objekte können grundsätzlich wie Zeiger auf beliebige andere Variablen definiert und verwendet werden.

Bei Zeigern auf Oberklassen und abgeleitete Klassen stellt sich nun die Frage, unter welchen Bedingungen eine Zuweisung erlaubt ist. Dabei gilt ähnliches wie bei der Zuweisung von Objekten selbst.

Im folgenden soll dies am Beispiel einer Personalverwaltung dargestellt werden. Gegeben sei eine Klasse `Mitarbeiter`, deren Objekte Mitarbeiter einer Firma mit ihren Daten (Name, Personalnr., usw.) beschreiben. Der enthaltene Zeiger `Next` soll zum Verketten mehrerer Mitarbeiter-Objekte zu einer linearen Liste dienen.

```
class Mitarbeiter
{
   protected:
      Mitarbeiter *Next;
   public:
      char Name[100];
      int PersonalNr;
      char Adresse[100];
      // ... weitere Daten ...
};
```

Von der Klasse `Mitarbeiter` soll die Klasse `Chef` abgeleitet werden. Ein `Chef`-Objekt soll zusätzlich zu allen Daten normaler Mitarbeiter eine Liste der unterstellten Mitarbeiter beinhalten (Feld `Untergebene`).

```
class Chef: public Mitarbeiter
{
   protected:
```

```
      Mitarbeiter *Untergebene;
};
```

Will man nun mit diesen Klassen alle Daten aller Mitarbeiter (welche teilweise wieder Chefs sind) hierarchisch verwalten, so stellt sich folgende wesentliche Frage: *Darf einer Variablen vom Typ "Zeiger auf `Mitarbeiter`" die Adresse eines Objekts der Klasse `Chef` zugewiesen werden?*

Die Frage ist mit JA zu beantworten, da ein Objekt der Klasse `Chef` alle Komponenten der Klasse `Mitarbeiter` besitzt (siehe auch vorigen Abschnitt). Umgekehrt darf jedoch einem Zeiger vom Typ "Zeiger auf `Chef`" nicht die Adresse eines Objekts der Klasse `Mitarbeiter`, sondern nur die eines Objekts der Klasse `Chef` zugewiesen werden.

```
Mitarbeiter *Zeiger_auf_Mitarbeiter;
Chef *Zeiger_auf_Chef;

Zeiger_auf_Mitarbeiter = Zeiger_auf_Chef; // erlaubt!
Zeiger_auf_Chef = Zeiger_auf_Mitarbeiter; // NICHT erlaubt!!!

// Grundsätzlich ist die explizite Typumwandlung erlaubt,
// sollte aber vermieden werden,
// da dadurch Fehler entstehen können!
Zeiger_auf_Chef = (Chef *) Zeiger_auf_Mitarbeiter;
```

Diese Regelung ist für dieses Beispiel auch einsichtig: Jeder Chef ist ein Mitarbeiter, aber nicht jeder Mitarbeiter ist ein Chef.

Allgemein gilt: Überall dort, wo ein Zeiger oder eine Referenz auf ein Objekt einer Oberklasse erwartet wird, darf auch die Adresse eines Objektes einer von der Oberklasse **öffentlich** abgeleiteten Klasse (bzw. für Referenzen nicht die Adresse sondern das Objekt direkt) angegeben werden. Ist die abgeleitete Klasse von der Oberklasse nur privat abgeleitet, ist dies nicht erlaubt.

Achtung:

Wird ein Zeiger einer abgeleiteten Klasse in einen Zeiger auf eine Oberklasse explizit umgewandelt (wie oben gezeigt), so zeigt dieser neue Zeiger schließlich auf ein Oberklassenobjekt, das man sich als Teil des Objekts der abgeleiteten Klasse vorstellen kann. Dabei ändert sich in der Regel nicht nur der Typ des Zeigers, sondern auch sein Adresswert. Da bei `void`-Zeigern explizit keine Typisierung vorhanden ist, kann es bei der Verwendung von `void`-Zeigern zu schwerwiegenden Fehlern kommen.

```
class O
{
   // ...
};

class A: public O
{
   // ...
};

O *pO;
A *pA = new A;
void *vp;

vp = pA;
pO = (O *) vp; // schwerwiegender Fehler!
```

Bei der letzten Zuweisung wird die Adresse NICHT umgerechnet, da `void*` beteiligt ist. `pO` zeigt also auf einen Speicherplatz des Typs `A`, obwohl beim Zugriff der Typ `O` verwendet wird. Die Folge ist ein Speicherzugriffsfehler!

Dagegen würde alles gut funktionieren, wenn kein `void`-Zeiger verwendet wird.

```
class O
{
   // ...
};

class A: public O
{
   // ...
};

O *pO;
A *pA = new A;

pO = pA; // So funktioniert es!
```

3.7. *Virtuelle Methoden*

Im vorigen Abschnitt wurde erläutert, dass ein Oberklassenzeiger auch auf Objekte öffentlich abgeleiteter Klassen zeigen kann. Ferner wurde bereits erwähnt, dass eine abgeleitete Klasse eine Methode beinhalten kann, welche den gleichen Namen und die gleichen Parameter wie eine Methode der Oberklasse besitzt.

Im folgenden Fall stellt sich nun die Frage, welche Methode wirklich abgearbeitet wird - die Methode der Oberklasse oder die der abgeleiteten Klasse?

Beispiel:

kap03_06.cpp

```
01 #include <stdio.h>
02
03 class Oberklasse
04 {
05    public:
06       void Methode() {  printf("Methode der Oberklasse\n");  }
07 };
08
09 class Ableitung: public Oberklasse
10 {
11    public:
12       void Methode() {  printf("Methode der Ableitung\n");  }
13 };
14
15 int main()
16 {
17    Oberklasse *Zeiger_auf_Oberklasse;
18    Ableitung AbleitungObjekt;
19
20    Zeiger_auf_Oberklasse = &AbleitungObjekt;
21    Zeiger_auf_Oberklasse->Methode();
22    // welche Methode wird aufgerufen?
23
24    return 0;
25 }
```

Auf den ersten Blick ist nicht eindeutig, welche der beiden Methoden (die der Oberklasse oder die der abgeleiteten Klasse) aufgerufen wird. Für `Oberklasse::Methode` spricht, dass der Typ des Zeigers "Zeiger auf Oberklasse" ist. Da es sich aber bei dem Objekt, auf das der Zeiger `Zeiger_auf_Oberklasse` zeigt, um ein Objekt von `Ableitung` handelt, erwartet man, dass die Funktion `Ableitung::Methode` ausgeführt wird.

Tatsächlich wird in diesem Beispiel `Oberklasse::Methode` aufgerufen, da bereits zur Übersetzungszeit auf Grund des Zeigertyps die Methode ausgewählt wurde (**statische Bindung**).

Wird jedoch in der Oberklasse der Deklaration der Methode das Schlüsselwort `virtual` vorangestellt, so wird erst während der Laufzeit des Programms aufgrund des aktuellen Objekts entschieden, welche Funktion wirklich abzuarbeiten ist. Im obigen Beispiel wäre das somit die Methode `Ableitung::Methode`.

```
class Oberklasse
{
   public:
      virtual void Methode();
};

// der Rest bleibt unverändert!
```

Methoden, die auf diese Art mit dem Schlüsselwort `virtual` deklariert werden, werden **virtuelle Methoden** genannt.

Soll ohne den Mechanismus der virtuellen Methoden die Methode abgearbeitet werden, welche der Klasse des aktuellen Objekts entspricht, so kann dies auf folgende Weise gelöst werden:

kap03_07.cpp

```
01 #include <stdio.h>
02
03 class Oberklasse
04 {
05    public:
06       int Klasse;
07       Oberklasse(int K): Klasse(K) { }
08       void Methode() {  printf("Methode der Oberklasse\n");  }
09 };
10
11 class Ableitung: public Oberklasse
12 {
13    public:
14       Ableitung(int K): Oberklasse(K) { }
15       void Methode() {  printf("Methode der Ableitung\n");  }
16 };
17
18 void Aufruf(Oberklasse *Zeiger_auf_Oberklasse);
19
20 int main()
21 {
22    Oberklasse OberklasseObjekt(1);
23    Ableitung AbleitungObjekt(2);
24
25    Aufruf(&OberklasseObjekt);
26    Aufruf(&AbleitungObjekt);
27
28    return 0;
29 }
30
31 void Aufruf(Oberklasse *Zeiger_auf_Oberklasse)
32 {
33    switch(Zeiger_auf_Oberklasse->Klasse)
34    {
35       case 1: Zeiger_auf_Oberklasse->Methode();
36               break;
37       case 2: ((Ableitung *) (Zeiger_auf_Oberklasse))->Methode();
38               break;
```

```
39     }
40 }
```

Ähnlich wie hier dargestellt wird auch zur Laufzeit bei virtuellen Methoden vorgegangen, nur dass dies dem Benutzer vollkommen verborgen bleibt. Aufrufe virtueller Methoden sind daher etwas langsamer als normale Methodenaufrufe. Die Tabelle, in der die Adressen der virtuellen Methoden stehen, wird auch oft als **VFT** (**virtual function table**) bezeichnet. Da die Adresse der Methode erst zur Laufzeit bestimmt wird, wird dies auch **späte Bindung** genannt.

Hinweise:

- Eine in der Oberklasse virtuell deklarierte Methode ist automatisch auch in allen abgeleiteten Klassen virtuell.
- Eine abgeleitete Klasse muss nicht notwendigerweise alle in der Oberklasse vorhandenen virtuellen Methoden neu definieren. Ist in einer abgeleiteten Klasse eine virtuelle Methode der Oberklasse nicht neu definiert, so wird, wie bei normalen Methoden auch, immer die Methode der Oberklasse abgearbeitet.
- Virtuelle Methoden einer Oberklasse werden durch eine neue Definition **überschrieben**, im Gegensatz zum **Überladen** einer Funktion, bei dem mehrere Methoden mit dem gleichen Namen, aber unterschiedlichen Parametern vorhanden sind.
- Konstruktoren dürfen nicht virtuell deklariert werden, da beim Erzeugen von Objekten immer die tatsächliche Klasse bereits zur Übersetzungszeit feststeht. Bei Destruktoren ist das anders (siehe nächsten Abschnitt): Ist ein Destruktor in der Oberklasse virtuell deklariert und wird z.B. `delete` mit einem Oberklassenzeiger aufgerufen, so werden für das konktrete Objekt einer evtl. abgeleiteten Klasse alle Destruktoren entsprechend richtig abgearbeitet.

Virtuelle Methoden sollten dann eingesetzt werden, wenn Methoden einer Oberklasse in abgeleiteten Klassen umdefiniert werden sollen und erst zur Laufzeit die richtige Methode bestimmt werden soll.

3.8. Virtuelle Destruktoren

Im Gegensatz zu Konstruktoren können Destruktoren als virtuell deklariert werden. Dies kann besonders dann Bedeutung erlangen, wenn es um die dynamische Speicherverwaltung geht. Dass, wie im folgenden gezeigt wird, der Einsatz von Zeigern auf Oberklassenobjekte nichts Ungewöhnliches ist, kann z.B. anhand einer Liste von Grafikobjekten erläutert werden. Ein Vektorgrafikprogramm verwendet beispielsweise eine Oberklasse `TGrafikObject`, von der weitere grafische Klassen wie `TLinie`, `TRechteck` und `TEllipse` abgeleitet sind. Ein Zeiger auf die Oberklasse `TGrafikObject` kann nun auf eine Liste dieser Objekte zeigen, die dann in dieser Reihenfolge gezeichnet werden. Wenn alle diese Objekte nicht mehr benötigt werden, können diese auch mittels `delete` (siehe Kapitel *Dynamische Speicherverwaltung in C++*) wieder gelöscht werden. Wie das folgende Beispiel zeigt, ist dies allerdings nicht ganz unproblematisch.

In diesem Beispiel wird ein Zeiger auf die Oberklasse deklariert. Über `new` wird ein neues Objekt der abgeleiteten Klasse erzeugt und dem Zeiger `Zeiger_auf_Oberklasse` zugewiesen. Dies ist erlaubt aufgrund der Überlegungen des letzten Abschnitts. Probleme bereitet jedoch das Löschen des Objekts, auf das der Zeiger zeigt. In diesem Fall wird nur ein Objekt von Typ `Oberklasse` gelöscht, d.h. dessen evtl. vorhandener Destruktor aufgerufen. Ein möglicherweise vorhandener Destruktor der abgeleiteten Klasse wird dabei (aufgrund der statischen Bindung) ignoriert.

kap03_08.cpp

```
01 #include <stdio.h>
02
03 class Oberklasse
04 {
05    public:
06       Oberklasse()  {  printf("Konstruktor Oberklasse\n");  }
07       ~Oberklasse() {  printf("Destruktor Oberklasse\n");   }
```

```
08 };
09
10 class Ableitung: public Oberklasse
11 {
12    public:
13       Ableitung()    {  printf("Konstruktor Ableitung\n");    }
14       ~Ableitung()   {  printf("Destruktor Ableitung\n");     }
15 };
16
17 int main()
18 {
19    Oberklasse *Zeiger_auf_Oberklasse;
20
21    Zeiger_auf_Oberklasse = new Ableitung;
22    delete Zeiger_auf_Oberklasse;
23
24    return 0;
25 }
```

Wird dieses Programm aufgerufen, wird nur der Destruktor der Oberklasse aufgerufen, nicht aber der Destruktor der abgeleiteten Klasse. Das Programm gibt folgendes aus, womit deutlich wird, dass wohl beide Konstruktoren, aber nur der Destruktor der Oberklasse aufgerufen werden:

```
Konstruktor Oberklasse
Konstruktor Ableitung
Destruktor Oberklasse
```

Dieses Verhalten kann wie im folgenden Programm dargestellt zu Problemen mit der dynamischen Speicherverwaltung führen.

Das folgende Programm kann so, wie es im folgenden angegeben wird, übersetzt werden. Das hat zur Folge, dass der Speicher, auf den der Zeiger `ArrayZeiger` der Klasse `Ableitung` zeigt, nicht freigegeben wird, da der Destruktor dieser Klasse nicht aufgerufen wird. Durch das Einfügen von `virtual` vor dem Destruktor der Oberklasse wird nun auch der Destruktor der abgeleiteten Klasse aufgerufen und der Speicher ordnungsgemäß wieder freigegeben.

kap03_09.cpp

```
01 #include <stdio.h>
02
03 class Oberklasse
04 {
05       int *ArrayZeiger, Laenge;
06    public:
07       Oberklasse(int L): Laenge(L)
08       {
09          ArrayZeiger = new int[L];
10          for (int i = 0; i < L; i++)
11             *(ArrayZeiger + i) = i + 1;
12          printf("Konstruktor von Oberklasse\n");
13       }
14       virtual void printArray()
15       {
16          printf("Arraywerte der Oberklasse:\n");
17          for (int i = 0; i < Laenge; i++)
18             printf("%i: %i\n", i, *(ArrayZeiger + i));
19       }
20       ~Oberklasse()
21       {
22          delete [] ArrayZeiger;
23          printf("Destruktor von Oberklasse\n");
```

```
24        }
25 };
26
27 class Ableitung: public Oberklasse
28 {
29        int *ArrayZeiger, Laenge;
30    public:
31        Ableitung(int L): Oberklasse(2 * L), Laenge(L)
32        {
33           ArrayZeiger = new int[L];
34           for (int i = 0; i < L; i++)
35              *(ArrayZeiger + i) = L - i;
36           printf("Konstruktor von Ableitung\n");
37        }
38        void printArray()
39        {
40           Oberklasse::printArray();
41           printf("Arraywerte der Ableitung:\n");
42           for (int i = 0; i < Laenge; i++)
43              printf("%i: %i\n", i, *(ArrayZeiger + i));
44        }
45        ~Ableitung()
46        {
47           delete [] ArrayZeiger;
48           printf("Destruktor von Ableitung\n");
49        }
50 };
51
52 int main()
53 {
54    Oberklasse *Zeiger_auf_Oberklasse;
55
56    Zeiger_auf_Oberklasse = new Ableitung(3);
57    Zeiger_auf_Oberklasse->printArray();
58    delete Zeiger_auf_Oberklasse;
59
60    return 0;
61 }
```

Das Programm gibt folgendes aus. Dabei wird auch deutlich, dass beide Klassen unterschiedliche Arrays enthalten, da die Arrays `private` definiert sind.

```
Konstruktor Oberklasse
Konstruktor Ableitung
Arraywerte der Oberklasse:
0: 1
1: 2
2: 3
3: 4
4: 5
5: 6
Arraywerte der Ableitung:
0: 3
1: 2
2: 1
Destruktor Oberklasse
```

4. Mehrfachvererbung

4.1. Mehrfachvererbung

C++-Compiler ab dem AT&T-Standard 2.0 erlauben beim Ableiten von Klassen die Angabe mehrerer Oberklassen. Eine so abgeleitete Klasse erbt alle Eigenschaften und Methoden aller angegebenen Oberklassen, wobei grundsätzlich dieselben Regeln gelten wie beim Ableiten von nur einer Oberklasse.

Syntax der Mehrfachvererbung

```
class Oberklasse1
{
   // ...
};

class Oberklasse2
{
   // ...
};

class MehrfachAbgeleitet: public Oberklasse1, public Oberklasse2
{
   // ...
};
```

Die neue Klasse `MehrfachAbgeleitet` wird deklariert wie gewohnt. Dem Klassennamen folgt ein Doppelpunkt. Dahinter können mehrere Oberklassen - mit Kommata voneinander getrennt - angegeben werden. Vor jeder Oberklasse kann ein Zugriffsattribut (`public`, `protected` oder `private`) angegeben werden. Wird ein Zugriffsattribut weggelassen, wird für die Ableitung `private` angenommen.

Die gesamten Eigenschaften und Methoden der Oberklassen (entsprechend der Zugriffsattribute) befinden sich nun automatisch in der abgeleiteten Klasse `MehrfachAbgeleitet`. Probleme, die durch Namensgleichheit von Eigenschaften und/oder Methoden in den Oberklassen auftreten können, werden im nächsten Abschnitt *Virtuelle Oberklassen* behandelt.

Die Konstruktoren der Oberklassen werden in der Reihenfolge ihrer Deklaration ausgeführt; die Destruktoren entsprechend in umgekehrter Reihenfolge.

Das folgende Beispielprogramm verwendet die Mehrfachvererbung für die Implementation der Klasse `TTimestamp`, die von den Klassen `TDate` und `TTime` abgeleitet ist.

Beispiel:

kap04_01.cpp

```
01 #include <stdio.h>
02
03 class TDate
04 {
05    private:
06       int Day, Month, Year;
07    public:
08       TDate(int D, int M, int Y)
09       {  Day = D; Month = M; Year = Y;  }
10       virtual void print()
11       {  printf("%02i.%02i.%04i", Day, Month, Year);  }
12 };
13
14 class TTime
15 {
```

```
16    private:
17       int Hour, Minute, Second;
18    public:
19       TTime(int H, int M, int S)
20       {  Hour = H; Minute = M; Second = S;  }
21       virtual void print()
22       {  printf("%02i:%02i:%02i", Hour, Minute, Second);  }
23 };
24
25 class TTimestamp: public TTime, public TDate
26 {
27    public:
28       TTimestamp(int D, int Mon, int Y, int H, int Min, int S)
29          : TDate(D, Mon, Y), TTime(H, Min, S) { }
30       void print()
31       {
32          TDate::print();
33          printf(", ");
34          TTime::print();
35       }
36 };
37
38 int main()
39 {
40    TTimestamp t(25, 1, 2010, 20, 52, 45);
41
42    t.print();           printf("\n");
43    t.TDate::print();    printf("\n");
44    t.TTime::print();    printf("\n");
45
46    return 0;
47 }
```

In der Klasse `TTimestamp` kann somit ein vollständiger Zeitpunkt gespeichert werden, ohne dass Datum und Uhrzeit und deren Funktionalität neu definiert werden müssen. Der Konstruktor der Klasse `TTimestamp` hat nur die Aufgabe, die Daten an die Konstruktoren der beiden Oberklassen weiterzugeben.

Auch in der Klasse `TTimestamp` gibt es eine Methode `print`. Diese greift jedoch nur auf die gleichnamigen Methoden der beiden Oberklassen zu. Dazu muss aber der Name der Oberklasse mit dem Bereichsoperator vorgesetzt werden.

In der `main`-Funktion wird zuerst die `print`-Methode der Klasse `TTimestamp` aufgerufen. Danach werden die `print`-Methoden der beiden Oberklassen aufgerufen. Auch hier muss dazu der Name der Oberklasse mit dem Bereichsoperator verwendet werden. Damit ist es problemlos möglich, auch auf die Methoden der Oberklassen zurückzugreifen.

4.2. Virtuelle Oberklassen

Dadurch, dass eine Klasse mehrere Oberklassen besitzen kann, können komplexe Klassenhierarchien entstehen. Ein Beispiel dafür stellt die Stream-Klassenhierarchie dar, auf welche im Kapitel *Datei- und String-Streams* näher eingegangen wird. Ein wesentlicher Punkt ist bei der Mehrfachvererbung jedoch zu beachten: Eine Klasse darf nicht direkt von ein und derselben Klasse mehrfach abgeleitet werden. Dadurch würden Namenskonflikte entstehen, da jeder Name doppelt vorkommen würde.

```
class A
{
   // ...
};
```

```
class B: public A, public A // nicht möglich!
{
   // ...
};
```

Es ist aber möglich, eine Klasse `D` von zwei Klassen `B` und `C` abzuleiten, die die gleiche Oberklasse besitzen.

```
class A
{
   // ...
};

class B: public A
{
   // ...
};

class C: public A
{
   // ...
};

class D: public B, public C
{
   // alle Eigenschaften und Methoden von A
   // existieren hier doppelt!!!
   // ...
};
```

In diesem Szenario existieren in der Klasse `D` zweimal die gleichen Eigenschaften und Methoden von der Klasse `A`. Sollen alle Eigenschaften nur ein einziges Mal in `D` enthalten sein, müssen `B` und `C` virtuell von `A` abgeleitet werden. Damit ist die Klasse `A` eine **virtuelle Oberklasse** (in der Literatur häufig auch **virtuelle Basisklasse** genannt).

```
class A
{
   // ...
};

class B: virtual public A
{
   // ...
};

class C: virtual public A
{
   // ...
};

class D: public B, public C
{
   // alle Eigenschaften und Methoden von A
   // existieren hier nur einmal!!!
   // ...
};
```

Syntax der virtuellen Oberklasse

Die von einer virtuellen Oberklasse abgeleitete Klasse wird deklariert wie bisher. Dem Klassennamen folgt ein Doppelpunkt. Den jetzt folgenden Oberklassen wird das Schlüsselwort `virtual` vorangestellt. Zwischen `virtual` und dem Namen der Oberklasse können dann die – wie gehabt – die Zugriffsattribute angegeben werden.

Wenn jetzt von diesen (virtuell abgeleiteten) Klassen andere Klassen abgeleitet werden, achtet der Compiler darauf, dass die Eigenschaften und Methoden von der virtuelle Oberklasse jeweils nur einmal vorhanden sind. Der Bereichsoperator braucht dann nicht mehr angegeben werden.

Bei der Aufrufreihenfolge der Konstruktoren werden immer die der virtuellen Oberklassen zuerst aufgerufen, die Destruktoren entsprechend in umgekehrter Reihenfolge.

Ob das mehrfache Vorhandensein einer Oberklasse gewünscht ist oder nicht, hängt von der konkreten Situation (Aufgabenstellung) ab. Die beiden folgenden Beispiele sollen dies veranschaulichen:

Beispiel 1:

In der Klasse `Ehe` sollen von beiden Klassen `Mann` und `Frau` die gesamten Eigenschaften vererbt werden. Eine virtuelle Oberklasse `Mensch` ist hier daher nicht angebracht.

```
class Mensch
{
   protected:
      int Alter;
      int Groesse;
      char Name[50];
      // ...
};

class Frau: public Mensch
{
   protected:
      int Kinder;
      // ...
};

class Mann: public Mensch
{
   protected:
      int Bart;
      // ...
};

class Ehe: public Mann, public Frau
{
   // ...
};
```

Beispiel 2:

Im zweiten Beispiel sollen in der Klasse `Rechteck` die Eigenschaften der Klasse `Zeichnung` nur einmal vorkommen. Daher ist die Ableitung mit `virtual` notwendig.

```
class Zeichnung
{
   protected:
      int X, Y;
      char Name[40];
      // ...
};
```

```
class Linie: virtual public Zeichnung
{
   protected:
      int Breite;
      int Farbe;
      // ...
};

class Flaeche: virtual public Zeichnung
{
   protected:
      int Muster;
      int Farbe;
      // ...
};

class Rechteck: public Linie, public Flaeche
{
   // ...
};
```

Hinweis:

Leider kann die virtuelle Ableitung einer Oberklasse nur dort definiert werden, wo direkt von der gemeinsamen Oberklasse abgeleitet wird. Bereits hier muss der Compiler nämlich die nötigen Schritte veranlassen. Hat man auf diese Ableitung – z.B. in einer Klassenbibliothek – keinen Zugriff mehr, so muss das Problem anders behandelt werden.

5. (Datei-)Ein- und Ausgabe in C++

Die Ein- und Ausgabe geschieht in C++ genauso wie in C mit Datenströmen (Streams). In C++ werden die Datenströme - wie sollte es anders sein - mit Objekten realisiert. Für die Bildschirmausgabe und Tastatureingabe wird dabei anstelle der `stdio.h` die Headerdatei `iostream` (ohne `.h`!; dafür muss der Namensraum `std` angegeben werden; siehe auch Kapitel *Namensräume*) benötigt. Durch das Einfügen dieser Headerdatei werden automatisch vier globale Objekte (`cout`, `cin`, `cerr` und `clog`) angelegt, die die verschiedenen Datenströme verkörpern.

Die Implementierung der Datenströme erfolgt in verschiedenen Klassen. Dabei werden folgende Gruppierungen vorgenommen. Die Klassen der Ein- und Ausgabe eines Streams (Standardein-/ausgabe, Datei, Speicher) werden jeweils in einer Klasse zusammengefasst.

	Standardein-/ausgabe	**Datei**	**Speicher**
Ausgabe	`ostream`	`ofstream`	`ostrstream`
Eingabe	`istream`	`ifstream`	`istrstream`
Ein- und Ausgabe	`iostream`	`iofstream`	`strstream`
Headerdatei	`iostream`	`fstream`	`strstream`

5.1. Bildschirmausgabe mit cout

Für die Bildschirmausgabe wird das Objekt `cout` verwendet. Zusammen mit diesem Objekt wird in der Headerdatei `iostream` ein neuer Operator definiert: Der Operator << erkennt automatisch den Datentypen des auszugebenden Wertes bzw. der auszugebenden Variable und formt die Ausgabe in eine Textdarstellung um. Damit entfallen die lästigen Formatangaben, die beim `printf` immer angegeben werden mussten.

Die Syntax sieht folgendermaßen aus:

`cout << Wert oder Variable;`

Sollen mehrere Werte oder Variablen ausgegeben werden, müssen diese jeweils mit dem <<-Operator getrennt werden. Z.B.:

kap05_01.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 int main()
06 {
07    int Zahl = 5;
08
09    cout << 7 << "Text" << 3.14153 << Zahl;
10
11    return 0;
12 }
```

Dieses Beispiel erzeugt folgende Ausgabe:

```
7Text3.141535
```

Es gibt auch zwei Methoden für das `cout`-Objekt, mit denen Zeichen ausgegeben werden können:

- Methode `put()`:

  Syntax: `ostream& put(char);`

  Beschreibung: Mit `put()` wird ein einzelnes Zeichen auf dem Bildschirm ausgegeben. Den Rückgabewert ignorieren Sie einfach. Z.B.:
  ```
  cout.put('A');
  ```
  gibt ein `A` an der aktuellen Cursorposition aus.

- Methode `write()`:

  Syntax: `ostream& write(char *,int);`

  Beschreibung: Mit `write()` wird eine Zeichenkette ausgegeben. Ist die Zeichenkette länger als die Anzahl, die mit dem zweiten Parameter angegeben wird, wird die angegebene Anzahl von Zeichen ausgegeben. Z.B.:
  ```
  cout.write("Hallo, Welt!",5);
  ```
  gibt `Hallo` - also die ersten 5 Zeichen - auf dem Bildschirm aus.

Natürlich können auch Formatangaben gemacht werden. Dazu gibt es im `cout`-Objekt einige Methoden:

- Methode `width()`:

  Syntax:
  ```
  int width();
  int width(int);
  ```

  Beschreibung: Mit `width()` wird die Weite der nächsten Zahlen- oder Textausgabe abgefragt (kein Parameter) bzw. festgelegt (der Parameter gibt die Weite an). Nach der Ausgabe wird die Weite automatisch auf 0 zurückgesetzt. Z.B.:
  ```
  cout << 7;
  cout.width(6); //Ausgabeweite: 6 Zeichen
  cout << 11;
  ```
  erzeugt die Ausgabe `7     11`, weil vor der zweiten Zahl vier Leerzeichen eingefügt wurden, um die Gesamtweite von sechs Zeichen zu erreichen.

- Methode `fill()`:

  Syntax:
  ```
  char fill();
  char fill(char);
  ```

  Beschreibung: Mit `fill()` lassen sich auch andere Füllzeichen anstelle des Leerzeichens angeben. Auch hier wird ohne Parameter abgefragt und mit Parameter ein neuer Wert gesetzt. Z.B.:
  ```
  char AltesZeichen = cout.fill(); // Füllzeichen merken
  cout.width(6);
  cout.fill('0');
  cout << 17 << " und " << 4;
  cout.fill(AltesZeichen); // altes Füllzeichen setzen
  ```
  erzeugt die Ausgabe `000017 und 4`, da die Angabe der Weite ja nur für die nächste Ausgabe gilt.

Die beiden Methoden `width()` und `fill()` gelten nicht für Ausgaben des Datentyps `char`! Z.B.:

```
cout.width(6);
cout.fill('_');
cout << '(' << 12 << ')';
```

erzeugt die Ausgabe `(____12)`, da die erste Ausgabe (die Klammer `'('`) vom Typ `char` ist und damit die Formatangaben erst für die nächste Ausgabe gelten.

- Methode `precision()`:

  Syntax:
  ```
  int precision();
  int precision(int);
  ```

Beschreibung: Mit `precision()` wird die Weite von Fließkommazahlen gesteuert. Dabei gibt der Parameter an, wieviele Ziffern insgesamt vor und nach dem Komma von der Zahl ausgegeben werden sollen (ohne den Dezimalpunkt mitzurechnen!), sofern die Flags (siehe nächsten Abschnitt) `fixed` oder `scientific` nicht gesetzt sind. Andernfalls legt `precision()` die Anzahl der Nachkommastellen fest. Auch hier wird ohne Parameter abgefragt und mit Parameter ein neuer Wert gesetzt. Z.B.:

```
int AlterWert = cout.precision(); // Einstellung merken
cout.precision(8);
cout << 1234.56789 << "\n";
cout.precision(4);
cout << 1234.56789 << "\n";
cout.precision(AlterWert); // wiederherstellen
```

erzeugt die Ausgabe (mit automatischer Rundung)

```
1234.5678
1235
```

Falls die Anzahl der Ziffern den Wert von `precision()` überschreitet, wird auf die wissenschaftliche Notation - also Ausgabe mit Exponent - umgeschaltet.

Mit `cout` wird eine gepufferte Ausgabe durchgeführt, d.h. die auszugebenden Zeichen werden nicht sofort, sondern erst etwas später ausgegeben. Gerade bei der Fehlersuche, aber auch bei Benutzerhinweisen und -aufforderungen, führt dies zu ungewollten Effekten. Mit der Ausgabe von `endl` wird dies vermieden: Es bewirkt, dass der Ausgabepuffer komplett geschrieben wird und beinhaltet gleichzeitig einen Zeilenvorschub. Z.B.

```
cout << 1234.56789 << endl;
```

Dies entspricht der Methode `flush()`:

```
cout << 1234.56789;
cout.flush();
```

Dieses Verhalten einer nicht gepufferten Ausgabe kann auch mit `cout.setf(ios::unitbuf);` erreicht werden.

5.2. Steuerung der Ausgabe über Flags

Ein **Flag** (engl. für *Flagge, Fahne*) ist ein Zeichen für ein Merkmal, das entweder vorhanden (Flag ist gesetzt) oder nicht vorhanden ist (Flag ist nicht gesetzt). Dafür werden meist ganze Zahlen verwendet, wobei jedes einzelne Bit dieser Zahl ein Flag ist.

Zur Formatsteuerung sind in der Klasse `ios` Flags definiert, die auch kombiniert werden können. In der folgenden Tabelle sind die Flags und deren Bedeutung zusammengestellt.

Flagname	**Bedeutung**
`ios::boolalpha`	Wahrheitswerte `true`/`false` ausgeben oder lesen
`ios::skipws`	Zwischenraumzeichen (White Spaces) ignorieren
`ios::left`	linksbündige Ausgabe
`ios::right`	rechtsbündige Ausgabe
`ios::internal`	zwischen Vorzeichen und Wert auffüllen
`ios::dec`	dezimale Ausgabe
`ios::oct`	oktale Ausgabe
`ios::hex`	hexadezimale Ausgabe
`ios::showbase`	Basis anzeigen
`ios::showpoint`	nachfolgende Nullen ausgeben
`ios::uppercase`	`E` und `X` statt `e` und `x` ausgeben
`ios::showpos`	+ bei positiven Zahlen anzeigen

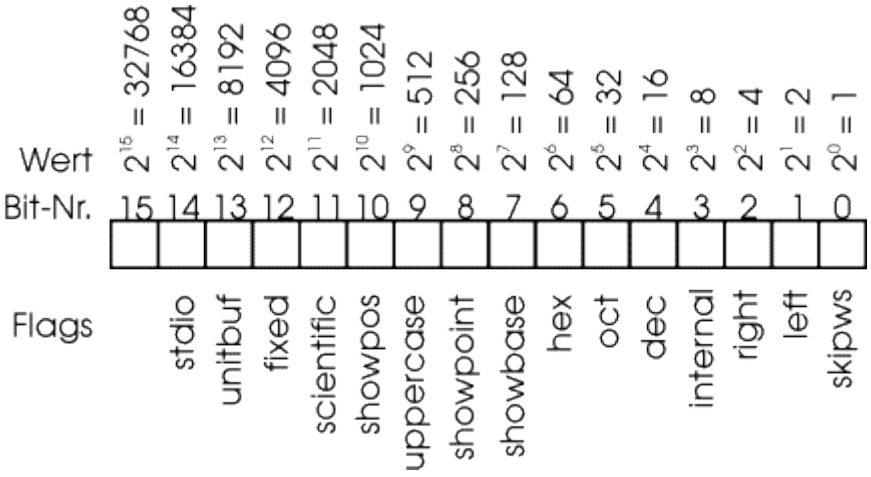
`ios::scientific`	Exponential-Format
`ios::fixed`	Gleitkomma-Format
	Puffer leeren (flush):
`ios::unitbuf`	nach jeder Ausgabeoperation
`ios::stdio`	nach jedem Textzeichen

Mit den nächsten Methoden lassen sich die Flags verändern:

- Methode `flags()`:

  Syntax:
  ```
  long flags();
  long flags(long);
  ```

  Beschreibung: Mit `flags()` werden mit Hilfe einer Bitmaske alle Flags je nach gesetzten Bits gesetzt oder nicht gesetzt. Eine **Bitmaske** ist eine ganze Zahl, wobei jedes Bit ein Flag ist. Durch die Bitmaske lassen sich auch Flags kombinieren. Dazu werden die einzelnen Flagwerte mit dem 'Bitweise Oder' (Operator '`|`'; siehe Kapitel *Datentypen in C* im Skript "Programmieren in C") verknüpft. Das folgende Bild zeigt die Bitmaske mit allen Flags. Da die Flags zur Klasse `ios` gehören, muss immer ein `ios::` vor den Flagnamen gesetzt werden.



  Im folgenden Beispiel werden die Flags `left`, `oct` und `fixed` gesetzt und alle anderen nicht gesetzt.
  Wichtig ist, dass gerade in Funktionen und Methoden der aktuelle Zustand der Flags zwischengespeichert und am Ende der Funktion oder Methode wieder zurückgesetzt wird. Z.B.:
  ```
  long AlterWert; // zum Zwischenspeichern
  long NeuerWert = ios::left | ios::oct | ios::fixed;
  cout << 32 << '\n';
  // gleichzeitiges Lesen und Setzen aller Flags:
  AlterWert = cout.flags(NeuerWert);
  cout << 32 << '\n';
  cout.flags(AlterWert); // alten Zustand wiederherstellen
  cout << 32 << '\n';
  ```
  erzeugt die folgende Ausgabe. Dabei ist 40 der oktale Wert von dezimal 32.
  ```
  32
  40
  32
  ```

- Methode `setf()`:

Syntax:
```
long setf(long);
long setf(long,long);
```

Beschreibung: Mit `setf()` kann ein einzelnes Flag gesetzt werden. Z.B.
```
cout.setf(ios::hex); // hexadezimale Ausgabe
cout << 64 << "\n";
```
erzeugt die Ausgabe `40` (hexadezimaler Wert für dezimal 64). Diese Variante ist aber ungünstig, da auf diesem Weg z.B. die Flags `dec`, `hex` und `oct` gleichzeitig gesetzt werden können. Diese schließen sich aber gegenseitig aus! Wenn ein Flag gesetzt werden soll, ist es besser, sicherheitshalber sich gegenseitig ausschließende Flags zurückzusetzen. Die Methode `setf()` mit zwei Parametern sorgt dafür. Der erste Parameter ist nach wie vor das zu setzende Flag. Der zweite Parameter ist eine Bitmaske, mit der sich mehrere Flags zurücksetzen lassen. Die Bits, deren Flags zurückgesetzt werden sollen, müssen 1 sein, die anderen 0. In der Klasse `ios` sind bereits drei Bitmasken vordefiniert:

Name	Wert
`adjustfield`	`ios::left | ios::right | ios::internal`
`basefield`	`ios::dec | ios::oct | ios::hex`
`floatfield`	`ios::scientific | ios::fixed`

Im folgenden Beispiel wird genau das gleiche Flag wie beim letzten Beispiel gesetzt (`ios::hex`). Diesmal werden aber die beiden Flags `ios::dec` und `ios::oct` durch die Bitmaske ausgeschaltet.
```
cout.setf(ios::hex,ios::basefield); // hexadezimale Ausgabe
cout << 64 << "\n";
```

- Methode `unsetf()`:

  Syntax:
  ```
  long unsetf(long);
  ```

  Beschreibung: Mit `unsetf()` kann ein einzelnes Flag zurückgesetzt werden. Z.B.
  ```
  cout.setf(ios::hex); // hexadezimale Ausgabe
  cout << 64 << " = ";
  cout.unsetf(ios::hex); // hexadez. Ausgabe aus
  xout << 64 << "\n";
  ```
  erzeugt die Ausgabe `40 = 64`.

5.3. Steuerung der Ausgabe über Manipulatoren

Für die Formatierung der auszugebenen Daten stehen sogenannte **Manipulatoren** zur Verfügung. Diese werden genau wie die Daten über den Operator << ausgegeben. Einen Manipulator haben wir bereits kennengelernt: `endl`.

Mit den Manipulatoren `dec`, `oct` und `hex` wird die darauffolgende Dezimalzahl entsprechend als Dezimal-, Oktal bzw. Hexadezimalzahl ausgegeben. Bei manchen Compilern (so z.B. beim MS Visual C++) wird nicht nur die darauffolgende Zahl sondern alle folgenden Zahlen im angegebenen Format ausgegeben. Die Verwendung dieser Manipulatoren ist natürlich deutlich einfacher als die Verwendung der Flags im vorigen Abschnitt.

Für die weiteren Manipulatoren wird die Headerdatei `iomanip.h` benötigt. Sie muss nach der `iostream.h` eingebunden werden. In der folgenden Tabelle werden alle Manipulatoren aufgelistet.

Manipulator	Beschreibung	benötigt `iomanip.h`
`dec`	Die folgende Zahl wird im entsprechenden	nein

`oct` `hex`	Zahlenformat ausgegeben.	
`flush`	Der Ausgabepuffer wird geleert, in dem alle gepufferten Ausgaben sofort ausgeführt werden.	nein
`endl`	Bewirkt das sofortige Schreiben aller gepufferten Ausgaben mit einen anschließenden Zeilenumbruch.	nein
`setw(int x)`	Setzt für die nächste **Ausgabe (und nur für diese)** die Ausgabebreite. Entspricht der Methode `width` von `cout`.	ja
`setfill(int x)`	Setzt das Füllzeichen für **alle** weiteren Ausgaben. Entspricht der Methode `fill` von `cout`.	ja
`setprecision(int x)`	Setzt die Genauigkeit **aller** nachfolgenden Fließkommazahlenausgaben. Die letzte Stelle wird gerundet. Entspricht der Methode `precision` von `cout`.	ja
`setiosflags(ios::`*`flag`*`)`	Setzt Ausgabeformatierungsflags. Dabei können die im vorigen Abschnitt vorgestellten Flags verwendet werden. Entspricht den Methoden `flags` bzw. `setf` von `cout`.	ja
`resetioflags(ios::`*`flag`*`)`	Setzt die angegebenen Ausgabeformatierungsflags wieder zurück. Dabei können die im vorigen Abschnitt vorgestellten Flags verwendet werden. Entspricht der Methode `unsetf` von `cout`.	ja

Da Manipulatoren nur gewöhnliche Funktionen sind, lassen sich auch eigene Manipulatoren definieren. Ein Manipulator erhält als Parameter eine Referenz auf den entsprechenden Datenstrom (z.B. `ostream&`) und gibt eine Referenz auf den entsprechenden Datenstrom wieder zurück. In der Funktion wird an den bestehenden Datenstrom (Parameter) die eigene Ausgabe herangehangen. Dieser erweiterte Datenstrom wird dann zurückgegeben.

Beispiel:

kap05_02.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 ostream& Tab(ostream& Datenstrom)
06 {
07    return Datenstrom << "\t";
08 }
09
10 ostream& Euro(ostream& Datenstrom)
11 {
12    return Datenstrom << " EUR";
13 }
```

```
14
15 int main()
16 {
17     cout << "Summe:" << Tab << 69.99 << Euro << endl;
18
19     return 0;
20 }
```

5.4. *Tastatureingabe mit cin*

Für die Tastatureingabe wird das Objekt `cin` verwendet. Auch mit diesem Objekt wird in der Headerdatei `iostream.h` ein neuer Operator definiert: Der Operator `>>` erkennt automatisch den Datentypen der Eingabe von der Tastatur, formt die Eingabe in den Datentypen der angegebenen Variablen um und speichert dort den eingegebenen Wert. Damit entfallen auch hier die lästigen Formatangaben, die beim `scanf` immer angegeben werden mussten.

Die Syntax sieht folgendermaßen aus:

```
cin >> Variable;
```

Sollen mehrere Variablen eingelesen werden, müssen diese jeweils mit dem >>-Operator getrennt werden. Z.B.:

```
int Zahl,Wert;
char Text[100];
cin >> Zahl >> Text >> Wert;
```

Die Eingabe `17 Mustermann 12` setzt die drei Variablen auf folgende Werte:

```
Zahl = 17
Text = "Mustermann"
Wert = 12
```

Auch das `cin`-Objekt hat eine Reihe von Methoden, die hier vorgestellt werden:

- Methode `get()`:

  Syntax:
  ```
  int get();
  istream& get(char&);
  istream& get(char*, int [, char]);
  ```

  Beschreibung: Mit der ersten Variante von `get()` wird genau ein Zeichen von der Tastatur eingelesen und als ganze Zahl zurückgegeben. Auch mit der zweiten Variante wird genau ein Zeichen eingelesen, das in der als Parameter übergebenen Variable gespeichert wird. Die dritte Variante lässt das Lesen von mehreren Zeichen (also einer Zeichenkette) zu, wobei angegeben wird, wieviele Zeichen gelesen werden sollen. Z.B.:
  ```
  unsigned char c,d,Eingabe[100];
  c = cin.get();              // erste Variante
  cin.get(d);                 // zweite Variante
  cin.get(Eingabe,20,'\n'); // dritte Variante
  // Die dritte Variante entspricht:
  cin >> Eingabe;
  ```
  Während die ersten beiden Varianten nur ein Zeichen einlesen können, liest die dritte Variante eine ganze Zeichenkette (einschließlich Leerzeichen!) ein. Dabei wird als zweiter Parameter die maximale Länge der einzulesenden Zeichenkette angegeben. Der dritte Parameter (optional) ist ein Abschlusszeichen, in den meisten Fällen ein `'\n'`. Dieses Abschlusszeichen wird aber nicht mehr in der Zeichenkette gespeichert!

- Methode `getline()`:

  Syntax:
  ```
  int getline(char *, int [, char]);
  ```

Beschreibung: Die Methode `getline()` hat die gleiche Wirkung wie die dritte Variante der Methode `get()`, mit dem kleinen Unterschied, dass mit `getline()` auch das Abschlusszeichen aus dem Eingabepuffer entfernt wird!

- Methode `putback()`:

  Syntax: `istream& putback(char);`

  Beschreibung: Mit dieser Methode wird das zuletzt gelesene Zeichen, das als Parameter übergeben wird, wieder in den Eingabestrom zurückgetan. Wenn ein anderes Zeichen als das zuletzt gelesene in den Eingabestrom zurückgetan wird, ist der Zustand undefiniert und kann von Compiler zu Compiler variieren.

- Methode `peek()`:

  Syntax: `int peek();`

  Beschreibung: Diese Methode liefert genauso wie die erste Variante der Methode `get()` das nächste Zeichen des Eingabestroms, allerdings wird dieses Zeichen nicht aus dem Eingabestrom herausgenommen. Diese Methode entspricht dem aufeinanderfolgenden Aufruf der beiden Methoden `get()` und `putback()`:

```
unsigned char c = cin.get();
cin.putback(c);
```

- Methode `ignore()`:

  Syntax: `istream& ignore([int [,int]]);`

  Beschreibung: Mit der `ignore()`-Methode können Zeichen aus einer Datei überlesen - d.h. gelesen und ignoriert - werden. Ohne Parameter wird nur das nächste Zeichen überlesen. Mit dem ersten (optionalen) Parameter wird die Anzahl der zu überlesenen Zeichen angegeben werden. Bei zwei Parametern gibt der erste die Anzahl der zu überlesenen Zeichen und der zweite Parameter ein Stopzeichen an. Mit dem Stopzeichen kann das Überlesen schon vor dem Erreichen der Zeichenanzahl abgebrochen werden. Das nächste Beispiel überliest alle Zeichen (maximal 80), bis ein Leerzeichen erscheint. Erscheint in den 80 Zeichen kein Leerzeichen, wird das Überlesen erst nach dem 80. Zeichen beendet.

```
cin.ignore(80,' ');
```

5.5. Datei-Ein- und Ausgabe

Das Prinzip des Umgangs mit Dateien ist das gleiche wie in der Programmiersprache C (siehe Kapitel *Datei-Ein- und Ausgabe in C* im Skript "Programmieren in C"): Datei öffnen, Lesen und/oder Schreiben von Daten und wieder Datei schließen. Im folgenden werden die verschiedenen Funktionen in C den Methoden in C++ gegenübergestellt. Im Anschluss wird das Beispielprogramm aus Kapitel *Datei-Ein- und Ausgabe in C*, das eine Datei zeichenweise in eine andere kopiert, noch einmal mit C++-Befehlen vorgestellt.

Befehl	**Funktion in C**	**Methode in C++**
Headerdatei	`stdio.h`	`fstream`
Datentyp	`FILE *`	`ifstream` *(zum Lesen)* `ofstream` *(zum Schreiben)*
öffnen	`FILE *fopen(char *FName, char *Modus);`	`open(char *FName, int Modus);`
Zeichen lesen	`int fgetc(FILE *);`	`fstream& get(int);`
Zeile lesen	`int fscanf(FILE *, char *, ...);`	`int getline(char *, int [, char]);`
Zeichen schreiben	`int fputc(int,FILE *);`	`fstream& put(int);`
Zeile schreiben	`int fprintf(FILE *, char *, ...);`	`fstream& write(char*,int);`
schließen	`int fclose(FILE *);`	`void close();`

	`int _fcloseall();`	`int fcloseall();`
Dateiende	`int feof(FILE *);`	`int eof();`

Da die Objekte `ifstream` und `ofstream` Eigenschaften und Methoden der Klasse `iostream` geerbt haben, sind eigentlich alle Methoden bereits bei `cout` und `cin` beschrieben worden.

Die Angabe, in welchem Modus eine Datei geöffnet werden soll, war in C eine Zeichenkette, hier ist es eine Bitmaske. Die verschiedenen Werte sind in öffentlichen Eigenschaften der Klasse `ios` vorgegeben. In der nächsten Tabelle werden diese Werte vorgestellt und den Zeichen des C-Modus gegenübergestellt.

Modus	**in C**	**in C++**
Lesen	`"r"`	`ios::in`
Schreiben	`"w"`	`ios::out`
Anhängen	`"a"`	`ios::app`
Dateiinhalt beim Öffnen löschen	*automatisch bei* `"w"`	`ios::trunc`
Binärmodus	`"b"`	`ios::binary`
Textmodus	`"t"`	*Standardeinstellung*

Auch in C++ lassen sich die einzelnen Modi kombinieren. Dazu werden die einzelnen Werte mit einem Bitweise Oder ('|') verknüpft.

Hier nun das Beispiel vom Ende des Kapitels *Datei-Ein- und Ausgabe in C*, jetzt aber in objektorientierter Programmierung:

Beispiel:

kap05_03.cpp

```
01 #include <iostream>
02 #include <fstream>
03
04 using namespace std;
05
06 int main()
07 {
08    ifstream Quelle;
09    ofstream Ziel;
10    char c = 0;
11    char fname1[] = "TEST.TXT", fname2[] = "TEST.BAK";
12
13    Quelle.open(fname1,ios::binary | ios::in);
14    if (!Quelle)
15       cout << "\nDatei " << fname1
16            << " konnte nicht geoeffnet werden!" << endl;
17    else
18    {
19       Ziel.open(fname2,ios::binary | ios::out);
20       if (!Ziel)
21          cout << "\nDatei " << fname2
22               << " konnte nicht geoeffnet werden!" << endl;
23       else
24          while (Quelle.get(c))
25             Ziel.put(c);
26    }
27    fcloseall(); // alle Dateien schließen
28
```

```
29     return 0;
30 }
```

6. Namensräume

Ein Namensraum (englisch: **namespace**) ist dasselbe wie ein Sichtbarkeitsbereich (englisch: scope). Namensräume sind eingeführt worden, damit verschiedene Programmteile zusammenarbeiten können, die vorher - also ohne Namensräume - aufgrund von Namenskonflikten im globalen Gültigkeitsbereich nicht zusammen verwendet werden konnten.

Beispiel:

kap06_01.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 // nuetzliche Funktionen der ABC GmbH:
06 int print(const char * T) { cout << "ABC: " << T << endl;     }
07 void func(double Zahl)    { cout << "ABC: " << Zahl << endl; }
08
09 // nuetzliche Funktionen der XYZ GmbH:
10 int print(const char * T) { cout << "XYZ: " << T << endl;     }
11                           // Fehler: Redefinition von print!
12 void func()               { cout << "XYZ: func()" << endl;    }
13
14 int main()
15 {
16    print("Hallo Welt!");  // welches print?
17    func(123.456);         // ok, da ueberladen
18    func();                // (unterschiedliche Parameter)
19
20    return 0;
21 }
```

Es ist offensichtlich nicht möglich, die Funktionsbibliotheken beider Firmen gleichzeitig zu benutzen. Eine häufig benutzte Abhilfe besteht in der Verwendung von Namenszusätzen:

Beispiel:

kap06_02.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 // nuetzliche Funktionen der ABC GmbH:
06 int ABC_print(const char * T) { cout << "ABC: " << T << endl;     }
07 void ABC_func(double Zahl)    { cout << "ABC: " << Zahl << endl; }
08
09 // nuetzliche Funktionen der XYZ GmbH:
10 int XYZ_print(const char * T) { cout << "XYZ: " << T << endl;     }
11 void XYZ_func()               { cout << "XYZ: func()" << endl;    }
12
13 int main()
14 {
15    ABC_print("Hallo Welt!"); // jetzt eindeutig!
16    ABC_func(123.456);
17    XYZ_func();
18
19    return 0;
20 }
```

6.1. Definition von Namensräumen

Dieser Weg ist jedoch unelegant bei den vielen existierenden Bibliotheken. Die Lösung besteht in der Einführung von zusätzlichen, übergeordneten Gültigkeitsbereichen, den Namensräumen. Die Definition von Namensräumen ähnelt der Deklaration von Klassen:

```
namespace ABC
{
   int print(const char *);
   void func(double);
} // Hier am Ende ist im Gegensatz zur
  // Deklaration von Klassen kein ; notwendig!
```

6.2. using-Direktive

Klassen und Funktionen in Namensräumen werden durch die **`using`-Direktive** nutzbar gemacht. Das Einleitungs-Beispiel würde dann wie folgt aussehen:

Beispiel:

kap06_03.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 namespace ABC
06 {
07    // nuetzliche Funktionen der ABC GmbH:
08    int print(const char * T) { cout << "ABC: " << T << endl;     }
09    void func(double Zahl)    { cout << "ABC: " << Zahl << endl; }
10 }
11
12 namespace XYZ
13 {
14    // nuetzliche Funktionen der XYZ GmbH:
15    int print(const char * T) { cout << "XYZ: " << T << endl;     }
16    void func()               { cout << "XYZ: func()" << endl;    }
17 }
18
19 int main()
20 {
21    using namespace ABC;  // using-Direktive
22                          // macht alle Namen aus ABC zugaenglich
23
24    print("Hallo Welt!"); // Ruft die Funktionen
25    func(123.456);        // des Namensraumes ABC auf!
26
27    func();               // Fehler!
28                          // Diese Funktion ist nicht in ABC definiert!
29
30    return 0;
31 }
```

6.3. *Qualifizierte Namen*

Eine Möglichkeit, um den Fehler im vorigen Programm durch den Aufruf von `func()` zu beheben, ist die Verwendung eines **qualifizierten Namens**. Dazu wird vor dem Funktionsnamen der Name des Namensraumes und der Bereichsoperator eingefügt.

Beispiel:

kap06_04.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 namespace ABC
06 {
07    // nuetzliche Funktionen der ABC GmbH:
08    int print(const char * T) { cout << "ABC: " << T << endl;     }
09    void func(double Zahl)    { cout << "ABC: " << Zahl << endl; }
10 }
11
12 namespace XYZ
13 {
14    // nuetzliche Funktionen der XYZ GmbH:
15    int print(const char * T) { cout << "XYZ: " << T << endl;     }
16    void func()               { cout << "XYZ: func()" << endl;    }
17 }
18
19 int main()
20 {
21    using namespace ABC;  // using-Direktive
22                          // macht alle Namen aus ABC zugaenglich
23
24    print("Hallo Welt!"); // Ruft die Funktionen
25    func(123.456);        // des Namensraumes ABC auf!
26
27    XYZ::func();          // jetzt durch Verwendung des qualifizierten
28                          // Namens (Namensraum) korrekt!
29
30    return 0;
31 }
```

Alle Klassen und Funktionen der C++-Standardbibliothek sind im Namensraum `std`. Aus diesem Grund wird in den Programmen immer `using namespace std;` benutzt. Alternativ möglich, aber umständlicher ist der Zugriff über den qualifizierten Namen, zum Beispiel

```
std::cout << "keine using-Direktive notwendig!";
```

6.4. *using-Deklaration*

Eine andere Möglichkeit ist der gezielte Zugriff auf Teile eines Namensraums durch eine **`using`-Deklaration**. Diese ist ähnlich wie eine using-Direktive aufgebaut.

Beispiel:

kap06_05.cpp

```
01 #include <iostream>
02
03 using namespace std;
```

```
04
05 namespace ABC
06 {
07    // nuetzliche Funktionen der ABC GmbH:
08    int print(const char * T) { cout << "ABC: " << T << endl;    }
09    void func(double Zahl)    { cout << "ABC: " << Zahl << endl; }
10 }
11
12 namespace XYZ
13 {
14    // nuetzliche Funktionen der XYZ GmbH:
15    int print(const char * T) { cout << "XYZ: " << T << endl;    }
16    void func()               { cout << "XYZ: func()" << endl;   }
17 }
18
19 int main()
20 {
21    using namespace ABC;       // using-Direktive
22                               // macht alle Namen aus ABC zugaenglich
23
24    print("Hallo Welt!");      // Ruft die Funktionen
25    func(123.456);             // des Namensraumes ABC auf!
26
27    XYZ::func();               // Jetzt durch direkte Angabe
28                               // des Namensraumes korrekt!
29
30    using XYZ::print;          // using-Deklaration
31                               // Nur die print-Funktion aus XYZ
32                               // wird zugaenglich gemacht.
33    print("Noch mal Hallo!"); // Die print-Funktion aus ABC wurde
34                               // damit ausgeblendet.
35
36    func(123.456);             // Trotzdem gilt fuer die anderen
37                               // Funktionen der Namespace ABC.
38
39    return 0;
40 }
```

6.5. Namensräume abkürzen

Bei sehr langen Namen besteht die Möglichkeit der Abkürzung.

Beispiel:

```
namespace VerySecialSoftwareGmbH_Co_KG_ClassLibrary
{
   // ...
}
```

Dann kann durch folgende Direktive der Name des Namensraums abgekürzt werden:

```
namespace VSS_CL VerySecialSoftwareGmbH_Co_KG_ClassLibrary;

// Abkürzung verwenden:

using namespace VSS_CL;
```

6.6. *using-Direktiven in Klassen*

Ein Namensraum ist ein Gültigkeitsbereich ähnlich wie der einer Klasse, der ebenfalls einen Namen hat. Die `using`-Direktive ist für Klassenmethoden erlaubt, um in einer abgeleiteten Klasse den gezielten Zugriff auf eine Oberklassenmethode zu ermöglichen.

kap06_06.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 class Oberklasse
06 {
07    protected:
08       void f(int i) {  cout << "Protected Methode f" << endl;  }
09 };
10
11 class Abgeleitet: public Oberklasse
12 {
13    public:
14       using Oberklasse::f;
15 };
16
17 int main()
18 {
19    Oberklasse O;
20    Abgeleitet A;
21
22 // O.f(0);   // Fehler, da Methode f in Oberklasse nur protected ist!
23    A.f(0);   // ok, da Methode f in Abgeleitet public ist!
24
25    return 0;
26 }
```

Mit dieser `using`-Direktive ist `Abgeleitet::f()` ein **öffentliches** Synonym für `Oberklasse::f()`. Private Methoden der Klasse `Oberklasse` können auf diese Art nicht öffentlich gemacht werden.

6.7. *Geschachtelte Namensräume*

Namensräume können auch geschachtelt werden. Dabei müssen natürlich auch alle bisher bekannten Regeln von Gültigkeitsbereichen und Sichtbarkeit berücksichtigt werden.

Beispiel:

kap06_07.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 namespace OuterNamespace
06 {
07    int i = 3;
08
09    namespace InnerNamespace
10    {
11       void Fkt1() {  cout << i << endl;  } // OuterNamespace::i
12
```

```
13        int i = 9;
14
15        void Fkt2() {  cout << i << endl;  } // InnerNamespace::i
16    }
17 }
18
19 using namespace OuterNamespace::InnerNamespace;
20
21 int main()
22 {
23    Fkt1(); // gibt eine 3 aus
24    Fkt2(); // gibt eine 9 aus
25
26    return 0;
27 }
```

6.8. *Namenslose Namensräume*

Wird ein Namensraum ohne Namen definiert, so wird vom Compiler automatisch ein einmaliger Name für diesen Namensraum vergeben. Da dieser Name aber unbekannt ist, wird er meist mit `unique` bezeichnet.

Namenslose Namensräume sind automatisch zugänglich; es muss also keine `using`-Direktive oder `using`-Deklaration verwendet werden, um auf die Elemente dieser Namensräume zugreifen zu können.

Beispiele:

kap06_08.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 namespace
06 {
07    int i = 3;
08 }
09
10 int main()
11 {
12    cout << i << endl; // gibt eine 3 aus
13
14    return 0;
15 }
```

kap06_09.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 namespace
06 {
07    int i = 3;
08
09    void Fkt1() { cout << i << endl; } // unique::i
10 }
11
12 namespace A
```

```
13 {
14    namespace
15    {
16       int i = 5;
17       int j = 9;
18    }
19    void Fkt2() { cout << i << endl; } // A::unique::i
20 }
21
22 using namespace A;
23
24 int main()
25 {
26    A::i++;    // erhoeht A::unique::i um 1
27    j++;       // erhoeht A::unique::j um 1
28 // i++;       // Fehler! Welches i?
29               // unique::i  oder  A::unique::i ?
30    ::i++;     // jetzt eindeutig: unique::i!
31    Fkt1();    // gibt eine 4 aus
32    Fkt2();    // gibt eine 6 aus
33
34    return 0;
35 }
```

7. Datentypen in C++

In C++ können alle Datentypen von C verwendet werden. Zusätzlich gibt es einige neue Datentypen, die in diesem Kapitel vorgestellt werden.

7.1. Komplexe Zahlen

Komplexe Zahlen werden im Computer immer durch zwei reelle Zahlen dargestellt: dem Real- und dem Imaginärteil. Beide Zahlen können aus einem der drei möglichen reellen Zahlentypen bestehen (`float`, `double` oder `long double`). Die komplexen Zahlen gehören also nicht zu den Grunddatentypen, sondern zu den zusammengesetzten Datentypen.

Mit komplexen Zahlen kann wie mit reellen Zahlen gerechnet werden. Insbesondere können alle arithmetischen Operatoren für reelle Zahlen einschließlich der Kurzform-Operatoren verwendet werden. Von den relationalen Operatoren können nur == und != (Prüfung auf Gleichheit bzw. Ungleichheit) verwendet werden, da die anderen bei komplexen Zahlen keinen Sinn ergeben.

Um komplexe Zahlen verwenden zu können, wird die Headerdatei `complex` (**ohne .h !!!**) benötigt. Die folgenden Beispiele sollen zeigen, wie Objekte von komplexen Zahlen definiert und verwendet werden. Dabei werden die komplexen Zahlen mit `complex<float>` deklariert. Anstelle von `float` kann auch `double` bzw. `long double` eingesetzt werden. D.h. es werden Objekte der Klasse `complex` verwendet, wobei in den spitzen Klammern angegeben wird, welche Datentypen die Real- und Imaginärteile haben sollen.

Um ein Objekt vom Typ komplexe Zahl bei der Definition gleich zu initialisieren, werden Real- und Imaginärteil in runden Klammern und mit einem Komma getrennt direkt hinter das Objekt geschrieben. Damit wird der entsprechende Konstruktor aufgerufen, der die angegebenen Werte in dem Real- und Imaginärteil speichert.

Werden für Real- oder Imaginärteil direkt Zahlen angegeben, müssen an diese Zahlen der Suffix des Datentyps angehangen werden: `f` für `float` und `l` für `long double`. Für `double` ist kein Suffix nötig (siehe auch Kapitel *Datentypen in C*, Abschnitt *Reele Zahlen* im Skript "Programmieren in C"). Wird der Suffix bei `float`- oder `long double`-Zahlen weggelassen, meldet der Compiler Warnungen bei der impliziten Typumwandlung.

Beispiel:

kap07_01.cpp

```
01 #include <iostream>
02 #include <complex>
03
04 using namespace std;
05
06 int main()
07 {
08    complex<float> c1(1.2f, 3.4f), c2, c3;
09
10    c2 = c1 * 5.0f; // Suffix f fuer float notwendig!
11    c3 = conj(c1);  // Konjugiert komplexe Zahl
12
13    cout << c1 << endl;
14    cout << c2 << endl;
15    cout << c3 << endl;
16
17    return 0;
18 }
```

Wichtig ist hier, dass der Namensraum `std` angegeben wird, da die Klasse `complex` (generell alle Klassen aus den Headerdateien ohne der Dateiendung .h) im Standard-Namensraum definiert ist.

Methoden der Klasse `complex`:

- Methode `real()`:

  Syntax: `float real(void);`

  Beschreibung: Gibt den Realteil der komplexen Zahl zurück. Der Rückgabewert ist hier mit `float` angegeben, kann aber entsprechend auch `double` oder `long double` sein.

- Methode `imag()`:

  Syntax: `float imag(void);`

  Beschreibung: Gibt den Imaginärteil der komplexen Zahl zurück. Der Rückgabewert ist hier mit `float` angegeben, kann aber entsprechend auch `double` oder `long double` sein.

Funktionen zu komplexen Zahlen:

- Funktion `conj()`:

  Syntax: `complex<float> conj(complex<float>);`

  Beschreibung: Gibt die konjugiert komplexe Zahl der als Parameter übergebenen, komplexen Zahl zurück. Rückgabewert und Parameter sind hier mit `float` angegeben, können aber entsprechend auch `double` oder `long double` sein.

- Funktion `polar()`:

  Syntax: `complex<float> polar(float,float);`

  Beschreibung: Gibt die komplexe Zahl, die sich aus dem Betrag (1. Parameter) und der Phase (2. Parameter im Bogenmaß) ergibt, zurück. Rückgabewert und Parameter sind hier mit `float` angegeben, können aber entsprechend auch `double` oder `long double` sein.

- Funktion `abs()`:

  Syntax: `float abs(complex<float>);`

  Beschreibung: Gibt den Betrag der komplexen Zahl zurück. Rückgabewert und Parameter sind hier mit `float` angegeben, können aber entsprechend auch `double` oder `long double` sein.

- Funktion `arg()`:

  Syntax: `float arg(complex<float>);`

  Beschreibung: Gibt die Phase im Bogenmaß der komplexen Zahl zurück. Rückgabewert und Parameter sind hier mit `float` angegeben, können aber entsprechend auch `double` oder `long double` sein.

- Funktion `norm()`:

  Syntax: `float norm(complex<float>);`

  Beschreibung: Gibt das Betragsquadrat der komplexen Zahl zurück. Rückgabewert und Parameter sind hier mit `float` angegeben, können aber entsprechend auch `double` oder `long double` sein.

Beispiel:

kap07_02.cpp

```
01 #include <iostream>
02 #include <complex>
03
```

```
04 using namespace std;
05
06 int main()
07 {
08    complex<float> c1(3.2f, 1.4f), c2, c3;
09    // Berechnung von Pi:
10    const float pi = 4.0 * atan(1.0);
11    // oder Benutzung der Konstanten M_PI aus cmath:
12    // (Headerdatei cmath wird von complex geladen)
13    // const float pi = M_PI;
14    float Betrag, Phase, BetrQuadr;
15
16    Betrag = 100.0;
17    Phase = pi / 4.0; // pi/4 = 45° = 0.785398
18    c2 = polar(Betrag, Phase);
19    BetrQuadr = norm(c2);
20    Betrag = abs(c2);
21    Phase = arg(c2);
22
23    cout << "c2 = "            << c2        << endl;
24    cout << "Betrag: "         << Betrag    << endl;
25    cout << "Phase : "         << Phase     << endl;
26    cout << "Betragsquadrat: " << BetrQuadr << endl;
27
28    return 0;
29 }
```

7.2. Logischer Datentyp

Der logischer Datentyp wird nach dem englischen Mathematiker George Boole (1815 - 1864), der auch die nach ihm benannte Boolesche Algebra entwickelt hatte, mit `bool` bezeichnet. Variablen vom Typ `bool` können nur die Werte `true` (wahr) und `false` (falsch) haben.

Intern werden die logischen Werte – genauso wie in C – als ganze Zahlen gespeichert. Dabei ist `true` gleich 1 und `false` gleich 0. Bei der Umwandlung von ganzen Zahlen in den logischen Datentyp werden alle Zahlen ungleich 0 auf `true` und alle Zahlen gleich 0 auf `false` gesetzt.

Da logische Variablen intern als ganze Zahlen gespeichert werden, können alle Operatoren der ganzen Zahlen hier auch angewendet werden, wobei nicht alle Operationen auch Sinn machen (beispielsweise die Addition `true` und `true`). Daher noch einmal eine Tabelle mit den Operatoren, die für logische Variablen sinnvoll sind.

Operator	Beispiel	Bedeutung
`!`	`!i`	logische Negation
`&&`	`i && j`	logisches UND
`||`	`i || j`	logisches ODER
`=`	`h = i && j`	Zuweisung

Bei der Bildschirmausgabe von logischen Werten werden diese in eine Zahl umgewandelt und ausgegeben. Um logische Werte als Text – `true` oder `false` – auszugeben, muss das Flag `ios::boolalpha` gesetzt werden (siehe Abschnitt *Steuerung der Ausgabe über Flags* im Kapitel *(Datei-)Ein- und Ausgabe*).

Beispiel:

kap07_03.cpp

```
01 #include <iostream>
02
```

```
03 using namespace std;
04
05 int main()
06 {
07    bool Wahrheitswert = true;
08    unsigned char c;
09
10    cout << "Bitte einen Buchstaben eingeben: ";
11    cin >> c;
12
13    Wahrheitswert = (c >= 'A') && (c <= 'Z');
14    cout << Wahrheitswert << endl;   // gibt 0 oder 1 aus
15
16    // Wahrheitswerte als Text ausgeben:
17    cout.setf(ios::boolalpha);
18    cout << Wahrheitswert << endl;   // gibt true oder false aus
19
20    // Flag wieder loeschen:
21    cout.unsetf(ios::boolalpha);
22
23    return 0;
24 }
```

7.3. Zeichenketten

In C ist eine Zeichenkette ein Array von Zeichen. In C++ gibt es zusätzlich eine Klasse mit dem Namen `string` (in der gleichnamigen Headerdatei), in der viele Methoden bereits enthalten sind, die in C noch umständlich von Hand programmiert werden mussten, wie z.B. Textlänge, Texte kopieren, vergleichen, verbinden, usw. Der größte Unterschied ist, dass eine Zeichenkette der Klasse `string` nicht mit einem Nullzeichen abgeschlossen sein muss.

Um die Zeichen-Arrays in C von den Strings in C++ besser unterscheiden zu können, werden die Zeichen-Arrays in C auch häufig als C-Strings bezeichnet.

Beispiel:

kap07_04.cpp

```
01 #include <iostream>
02 #include <string>
03
04 using namespace std;
05
06 int main()
07 {
08    string EinString("Hallo"), NochEinString;
09    cout << EinString << endl;
10    // -> Hallo
11
12    // String zeichenweise ausgeben (wie in C):
13    for (int i = 0; i < EinString.size(); i++)
14    cout << EinString[i];
15    cout << endl;
16    // -> Hallo
17
18   // Strings koennen direkt zugewiesen werden:
19    NochEinString = EinString;
20    cout << NochEinString << endl;
```

```
21    // ist identisch mit:
22    NochEinString.assign(EinString);
23    cout << NochEinString << endl;
24    // -> Hallo
25
26    // Strings koennen mit arithmetischen Operatoren
27    // verknuepft werden:
28    NochEinString = ", Welt!";
29    EinString += NochEinString;
30    cout << EinString << endl;
31    // -> Hallo, Welt!
32
33    // Strings in andere einfuegen:
34    EinString.insert(7, "kleine ");
35    cout << EinString << endl;
36    // -> Hallo, kleine Welt!
37
38    // Teile eines Strings durch anderen String ersetzen:
39    EinString.replace(7, 6, "grosse");
40    cout << EinString << endl;
41    // -> Hallo, grosse Welt!
42
43    // Teile eines String entfernen (loeschen):
44    EinString.erase(7, 7);
45    cout << EinString << endl;
46    // -> Hallo, Welt!
47
48    return 0;
49 }
```

Im folgenden werden die wichtigsten Methoden der Klasse `string` aufgelistet:

- Methode `size()`:

  Syntax: `int size();`

  Beschreibung: Gibt die Länge des Strings zurück.

- Methode `resize()`:

  Syntax: `void resize(int n, char c = '\0');`

  Beschreibung: Verkürzt (`n < size()`) oder verlängert (`n > size()`) den String auf `n` Zeichen. Wird der String verlängert, werden die neuen Zeichen mit dem 2. Parameter `c` - wenn vorhanden - oder mit dem Nullzeichen gefüllt.

- Methode `clear()`:

  Syntax: `void clear();`

  Beschreibung: Löscht den Inhalt des Strings.

- Methode `empty()`:

  Syntax: `bool empty();`

  Beschreibung: Gibt `true` zurück, wenn der String die Länge 0 hat (`size() == 0`), sonst ein `false`.

- Methode `assign()`:

  Syntax:
  ```
  string& assign(string&);
  string& assign(char *);
  string& assign(char);
  ```

Beschreibung: Weist dem String einen anderen String (1. Variante), ein Array von Zeichen (eine Zeichenkette in C; daher auch häufig C-String genannt) (2. Variante) oder ein einzelnes Zeichen (3. Variante) zu. Der alte Inhalt des Strings wird gelöscht.

- Methode `append()`:

  Syntax:
  ```
  string& append(string&);
  string& append(char *);
  string& append(char);
  ```

  Beschreibung: Hängt an den String einen anderen String (1. Variante), einen C-String (2. Variante) oder ein einzelnes Zeichen (3. Variante) an. Anstelle dieser Methode können auch die Operatoren + und += verwendet werden.

- Methode `insert()`:

  Syntax:
  ```
  string& insert(int pos, string& Str);
  string& insert(int pos, char *c);
  ```

  Beschreibung: Fügt einen String `Str` bzw. einen C-String `c` an der Position `pos` des Strings ein.

- Methode `erase()`:

  Syntax:
  ```
  string& erase(int pos, int n);
  ```

  Beschreibung: Löscht `n` Zeichen im String ab der Position `pos`.

- Methode `replace()`:

  Syntax:
  ```
  string& replace(int pos, int n, string& Str);
  string& replace(int pos, int n, char *c);
  ```

  Beschreibung: Ersetzt `n` Zeichen des Strings ab der Position `pos` durch den String `Str` oder dem C-String `c`.

- Methode `copy()`:

  Syntax:
  ```
  int copy(char *c, int n, int pos);
  ```

  Beschreibung: Kopiert `n` Zeichen des Strings in den C-String `c` an die Position `pos`. Zurückgegeben wird die Anzahl der tatsächlich kopierten Zeichen.

- Methode `swap()`:

  Syntax:
  ```
  string& swap(string& Str);
  ```

  Beschreibung: Vertauscht den Inhalt des Strings mit dem des Strings `Str`.

- Methode `find()`:

  Syntax:
  ```
  int find(string& Str, int pos = 0);
  ```

  Beschreibung: Sucht im String ab der Position `pos` nach dem Vorkommen des Strings `Str` (wird auch Substring genannt). Zurückgegeben wird die erste Position des gesuchten Substrings. Wird der Substring nicht gefunden, wird die größtmögliche `unsigned int`-Zahl zurückgegeben.

- Methode `compare()`:

  Syntax:
  ```
  int compare(string& Str);
  ```

  Beschreibung: Vergleicht den String zeichenweise mit dem String `Str`. Es wird `0` zurückgegeben, wenn beide Strings identisch sind, sonst eine Zahl ungleich `0`.

Es gibt noch viele andere Methoden in dieser Klasse. Diese werden aber nicht weiter behandelt, da viele Methoden deutlich komplizierter sind und noch weitere Kenntnisse voraussetzen.

7.4. Vektor

Auch für eindimensionale Arrays gibt es in C++ einen neuen Datentypen bzw. eine Klasse: `vector`. Diese Klasse ist in der gleichnamigen Headerdatei `vector` definiert.

Auf den ersten Blick lassen sich Vektoren genauso handhaben wie Arrays, wie das nächste Beispiel zeigt. Auch wird genauso wie in C der Index nicht kontrolliert, ob er noch innerhalb des zulässigen Bereichs liegt (im Gegensatz zur Methode `at()` weiter unten).

Beispiel:

kap07_05.cpp

```
01 #include <iostream>
02 #include <vector>
03
04 using namespace std;
05
06 int main()
07 {
08    // Vektor mit 10 Zahlen definieren:
09    vector<int> Zahlen(10);
10
11    for (int i = 0; i < 10; i++)
12    {
13       Zahlen[i] = 17;
14       cout << Zahlen[i] << endl;
15    }
16
17    return 0;
18 }
```

Da der Datentyp `vector` eine Klasse ist, gibt es natürlich wieder einige Methoden, die das Leben erleichtern. Viele dieser Methoden überprüfen - im Gegensatz zu den Arrays - auch, ob der Index sich innerhalb der erlaubten Grenzen liegt. Außerdem lassen sich Vektoren während des Programmablaufs vergrößern oder verkleinern, d.h. Vektoren sind dynamisch.

Bei den folgenden Methoden ist der Datentyp `T` durch den Datentypen der Vektorelemente zu ersetzen.

- Methode `assign()`:

  Syntax: `void assign(int n, T& t);`

  Beschreibung: Weist dem Vektor `n` Elemente des Datentyps `T` zu. Die Größe des Vektors ist anschließend `n` - mit dieser Methode wird also gleichzeitig die Größe verändert - und alle Elemente des Vektors haben den Wert `t`.

- Methode `at()`:

  Syntax: `T& at(int n);`

  Beschreibung: Mit dieser Methode kann auf die Elemente des Vektors zugegriffen werden. Im Gegensatz zum Zugriff auf die Elemente mit den eckigen Klammern wird bei dieser Methode überprüft, ob der Index in dem zulässigen Bereich liegt. Wenn nicht, wird das Programm mit einer Fehlermeldung abgebrochen.

  Beispiel:

  kap07_06.cpp

```
01 #include <iostream>
02 #include <vector>
03
04 using namespace std;
```

```
05
06 int main()
07 {
08    vector<int> Zahlen(10);
09
10    for (int i = 0; i < 20; i++)
11    {
12       // zulaessig fuer i = 0 ... 19:
13       Zahlen[i] = i;
14       cout << Zahlen[i] << endl;
15       // dies ist mit folgender Zeile identisch:
16       cout << Zahlen.at(i) << endl;
17       // allerdings: Programmabbruch bei i == 10 !!!
18    }
19
20    return 0;
21 }
```

- Methode `push_back()`:

  Syntax: `void push_back(T& t);`

  Beschreibung: Fügt das Element `t` am Ende des Vektors ein, d.h. die Größe des Vektors wird um 1 erhöht.

- Methode `pop_back()`:

  Syntax: `void pop_back();`

  Beschreibung: Löscht das letzte Element des Vektors, d.h. die Größe des Vektors wird um 1 verringert.

- Methode `insert()`:

  Syntax:
  ```
  T& insert(int pos, T& t);
  void insert(int pos, int n, T& t);
  ```

  Beschreibung: Mit dieser Methode lassen sich Elemente an der Stelle `pos` einfügen. Die erste Variante fügt ein Element vom Typ `T` ein, zurückgegeben wird eine Referenz auf das eingefügte Element. Die zweite Variante fügt `n` Elemente vom Datentyp `T` an der Stelle `pos` ein und gibt nichts zurück.

- Methode `erase()`:

  Syntax:
  ```
  T& erase(int pos);
  T& erase(int pos1, int pos2);
  ```

  Beschreibung: Löscht ein oder mehrere Elemente des Vektors. Die Größe des Vektors verringert sich entsprechend. Die erste Variante löscht das Element an der Stelle `pos`, die zweite Variante alle Elemente im Bereich von `pos1` bis `pos2`. Beide Variante geben eine Referenz auf die Stelle, an der das letzte, gelöschte Element stand.

- Methode `clear()`:

  Syntax: `void clear();`

  Beschreibung: Löscht alle Elemente des Vektors. Die Größe des Vektors ist anschließend gleich 0.

- Methode `size()`:

  Syntax: `int size();`

  Beschreibung: Gibt die Größe des Vektors - also die Anzahl der Elemente im Vektor - an.

- Methode `resize()`:

  Syntax: `void resize(int n, T t);`

Beschreibung: Diese Methode ändert die Größe des Vektors auf n Elemente. Es werden also `n - size()` Elemente am Ende herangehangen oder am Ende gelöscht. Werden Elemente herangehangen, werden die neuen Elemente auf den Wert `t` gesetzt.

7.5. Konvertierung zwischen den Datentypen (Typumwandlung)

Bei der Typumwandlung in C (siehe letzten Abschnitt im Kapitel *Datentypen in C* im Skript "Programmieren in C") wird die Typkontrolle umgangen. Dadurch entstehen Fehlerquellen. Die etwas sichere Methode der Typumwandlung (im englischen *cast*) ist die Verwendung der Typumwandlungsoperatoren, die es nur in C++ gibt. Der einfachste Typumwandlungsoperator heißt `static_cast<`*`Typ`*`>(`*`Wert`* `bzw.` *`Variable`*`).`

Beispiel:

```
char c = 'A';
int i;
// Konvertierung 'A' in ganze Zahl 65:
i = static_cast<int>(c);
```

Genauso lassen sich auch Zeichen in reelle Zahlen konvertieren.

Beispiel:

```
char c = 'A';
double d;
// Konvertierung 'A' in reelle Zahl 65.0:
d = static_cast<double>(c);
```

Die weiteren Typumwandlungsoperatoren `dynamic_cast<`*`Typ`*`>(`*`Wert`* `bzw.` *`Variable`*`)`, `const_cast<`*`Typ`*`>(`*`Wert`* `bzw.` *`Variable`*`)` und `reinterpret_cast<`*`Typ`*`>(`*`Wert`* `bzw.` *`Variable`*`)` werden hier nicht weiter betrachtet.

8. Dynamische Speicherverwaltung in C++

Auch in C++ gibt es die Möglichkeit, Speicherbereiche während des Programmlaufs zu reservieren. Alle Hinweise, die im Kapitel *Dynamische Speicherverwaltung* im Skript "Programmieren in C" aufgelistet wurden, gelten natürlich auch hier.

8.1. Speicherbereiche reservieren

Zum Reservieren von Speicherbereichen wird der Operator `new` verwendet. `new` erkennt selbständig die benötigten Menge Speicher anhand des anzugebenden Datentyps. Dieser Operator liefert einen Zeiger auf den reservierten Speicherbereich zurück.

Beispiel:

kap08_01.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 int main()
06 {
07    int *p;       // Zeiger fuer den neuen Speicherbereich
08
09    p = new int; // Speicherbereich fuer 1 int reservieren
10    *p = 15;
11    cout << *p << endl;
12    delete p;
13
14    return 0;
15 }
```

Zum Zeitpunkt des Compilierens wird nur der Platz für den Zeiger `p` eingeplant. Mit `p = new int;` wird Speicherplatz in der Größe von `sizeof(int)` Bytes zur Laufzeit des Programms bereitgestellt. `p` zeigt anschließend auf diesen Speicherplatz.

Mit `new` kann auch ein Speicherbereich reserviert werden, der als Array interpretiert wird. Dazu wird hinter dem Datentypen die Anzahl der Elemente in eckigen Klammern angegeben, genau wie bei den bisherigen Arrays. Auch der Zugriff auf die einzelnen Elemente des Arrays geschieht wie gewohnt.

Beispiel:

kap08_02.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 int main()
06 {
07    int *pa;          // Zeiger fuer den neuen Speicherbereich
08
09    pa = new int[4]; // fuer Array von 4 int reservieren
10    pa[0] = 4;       // oder *pa = 4;
11    pa[1] = 7;       // oder *(pa + 1) = 7
12    pa[2] = 1;       // oder *(pa + 2) = 1
13    pa[3] = 1;       // oder *(pa + 3) = 1
14    cout << pa[0] << pa[1] << pa[2] << pa[3] << endl;
15    delete pa;
```

```
16
17     return 0;
18 }
```

Auch für Struktur- und Klassenobjekte lässt sich Speicher reservieren.

Beispiel:

kap08_03.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 int main()
06 {
07     struct KomplexeZahl
08     {
09        double Real;
10        double Imaginaer;
11     };
12     KomplexeZahl *Kp = new KomplexeZahl;
13
14     Kp->Real = 8;
15     Kp->Imaginaer = 15;
16     cout << Kp->Real << " + " << Kp->Imaginaer << "i" << endl;
17     delete Kp;
18
19     return 0;
20 }
```

8.2. Reservierte Speicherbereiche freigeben

Der `delete`-Operator gibt den reservierten Speicherbereich wieder frei, damit dieser von neuem belegt oder anderen Programmen zur Verfügung gestellt werden kann. Dazu wird hinter `delete` der Zeiger angegeben, der auf den freizugebenden Speicherbereich zeigt. Im folgenden Beispiel werden die drei Speicherbereiche, die im vorigen Abschnitt reserviert wurden, wieder freigegeben.

Beispiel:

```
delete p;      // int freigeben
delete [] pa; // Array freigeben
delete Kp;     // Struktur freigeben
```

Nach dem Freigeben der reservierten Speicherbereiche kann auf diese nicht mehr zugegriffen werden!

8.3. Größe des reservierten Speicherbereichs ändern

In der Programmiersprache C gibt es neben den Funktionen `malloc`, `calloc` und `free` auch noch die Funktion `realloc`, um einen bereits reservierten Speicherbereich zu vergrößern bzw. zu verkleinern. In C++ gibt es aber keine entsprechende Funktion.

Um sich einen entsprechende Funktion selber zu programmieren, müssen folgende Schritte durchgeführt werden:

- den neuen Speicherbereich reservieren,
- den Inhalt des alten Speicherbereichs in den neuen Speicherbereich kopieren,
- den alten Speicherbereich freigeben und
- den Zeiger, der auf den alten Speicherbereich zeigt, auf den neuen Speicherbereich zeigen lassen.

Als Parameter erhält die Funktion einen Zeiger auf den alten Speicherbereich, die Größe des alten Speicherbereichs sowie die gewünschte Größe des neuen Speicherbereichs. Die Funktion gibt als Ergebnis einen Zeiger auf den neuen Speicherbereich zurück.

Beispiel:

kap08_05.cpp

```
01 #include <iostream>
02 #include <new>
03
04 using namespace std;
05
06 int *renew(int *, int, int);
07
08 int main()
09 {
10    int *Array = new int[3];
11    int i;
12
13    for (i = 0; i < 3; i++)
14       Array[i] = i;
15    Array = renew(Array, 3, 5);
16    for (i = 3; i < 5; i++)
17       Array[i] = i;
18
19    for (i = 0; i < 5; i++)
20       cout << Array[i] << endl;
21    delete [] Array;
22
23    return 0;
24 }
25
26 int *renew(int *oldMem, int oldSize, int newSize)
27 {
28    int *newMem = new int[newSize];      // neuen Speicher reservieren
29    int i;
30    int copySize = (oldSize < newSize) ? oldSize : newSize;
31
32    for (i = 0; i < copySize; i++)       // alten Inhalt in den neuen
33       *(newMem + i) = *(oldMem + i);    // Speicherbereich kopieren
34
35    delete [] oldMem;                    // alten Speicher freigeben
36
37    return newMem;
38 }
```

8.4. Fehlerbehandlung

Ein Programm sollte immer das Ergebnis einer Speicheranforderung prüfen. Auch wenn im System eigentlich genügend Hauptspeicher vorhanden ist, kann es durch Abstürze oder fehlerhafte Programme vorkommen, dass dieser Speicher nicht mehr ausreicht bzw. blockiert ist. Kann der Operator `new` keinen Speicherbereich bereitstellen, wird bei Deaktivierung der Exceptions (mit dem Schlüsselwort `nothrow` in Klammern; siehe auch Kapitel 13) die Konstante `NULL` als Ergebnis geliefert – analog zur Programmiersprache C. In diesem Fall kann über eine `if`-Abfrage eine eigene Fehlerbehandlung erfolgen.

Beispiel:

```
int *Array = new (nothrow) int[10];
if (Array != NULL)
{
   // Anweisungen ...
}
else
   cout << "Zu wenig Speicher!" << endl;
```

Eine weitere Möglichkeit bietet C++ über einen sogenannten new-Handler als Exception (siehe Kapitel 13), der immer dann aufgerufen wird, wenn eine Speicheranforderung fehlgeschlagen ist. Ein new-Handler ist eine typische C-Funktion, die void zurückgibt und keine Argumente besitzt. Diese Funktion wird über die Funktion set_new_handler registriert. Sie müssen in Ihr Programm in diesem Fall die Header-Datei new einbinden.

```
#include <new>

void my_new_handler()
{  cout << "Zu wenig Speicher!" << endl;
}

set_new_handler(my_new_handler);
```

Jedes Mal, wenn eine Speicheranforderung fehlschlägt, wird der Text "Zu wenig Speicher" angezeigt. Je nach Compiler muss im new-Handler noch eine Fehlerbehandlung erfolgen bzw. das Programm abgebrochen werden. Denn durch die Verwendung des new-Handlers liefert new nicht mehr die Konstante NULL zurück!

Hinweis:

Je nach eingesetztem Compiler variiert der Einsatz eines new-Handlers bzw. er ist in einigen älteren Compiler-Versionen noch nicht implementiert.

Beispiel:

kap08_05.cpp

```
01 #include <iostream>
02 #include <cstdlib>
03 #include <new>
04
05 using namespace std;
06
07 void my_new_handler();
08
09 int main()
10 {
11    int i = 1;
12    long double *Zeiger;
13
14    set_new_handler(my_new_handler);
15    while (true)
16    {
17       Zeiger = new long double [100000];
18       cout << "100.000 long doubles zum "
19            << i << ". Mal reserviert!" << endl;
20       i++;
21
22       // folgendes funktioniert nicht, da new NICHT
23       // mehr den NULL-Zeiger zurueckliefert, wenn
24       // nicht genuegend Speicher vorhanden ist!
```

```
25        if (Zeiger == NULL)
26        {
27           cout << "Zeiger == NULL: Zu wenig Speicher!" << endl;
28           break;
29        }
30     }
31
32     return 0;
33  }
34
35  void my_new_handler()
36  {
37     cout << "my_new_handler: Zu wenig Speicher!" << endl;
38     exit(1);
39  }
```

9. Polymorphismus

Unter Polymorphismus (zu deutsch "Vielgestaltigkeit") versteht man, dass mit der "gleichen" Methode für verschiedene Objekte unterschiedliche Resultate erzielt werden können. In C++ kann das beispielsweise durch das Verwenden von Zeigern auf Oberklassenobjekte geschehen, die ja auch auf Objekte öffentlich abgeleiteter Klassen gerichtet werden kann (siehe Kapitel *Vererbung* Abschnitt *Virtuelle Methoden*).

Beispiel:

Es wurde die Klasse Fahrzeug entwickelt. Davon werden die Klassen `Auto` und `Motorboot` abgeleitet. Jede dieser drei Klassen besitzt eine Methode `fahren()` (gleicher Methodenname). Jede Klasse implementiert die Methode `fahren()` jedoch anders, nämlich ihren Ansprüchen entsprechend.

Bisher beinhalteten die Oberklassen alle Methoden und Eigenschaften, die allen abgeleiteten Klassen gemeinsam sind. Im folgenden soll dies erweitert werden, so dass die Oberklasse eine Vorlage für alle abgeleiteten Klassen darstellt, die die Vorlage ihren Gegebenheiten anpassen.

Definition:

Eine Klasse ist polymorph, wenn sie mindestens eine Methode besitzt - geerbt oder deklariert -, die virtuell ist.

Im folgenden wird dieses Konzept weiter ausgebaut.

9.1. Rein virtuelle Methoden

Rein virtuelle Methoden werden auch als *pure virtual functions* bezeichnet. Sie sind virtuelle Methoden, die als Vorlage für alle abgeleiteten Klassen dienen. In der Klasse, in der diese Methoden deklariert sind, werden sie allerdings nicht implementiert. Das heißt, sie dienen nur als Vorlage und müssen in allen abgeleiteten Klassen implementiert werden. Eine rein virtuelle Methode wird durch eine an die Deklaration angehängte Zeichenfolge "`= 0`" erzeugt.

Beispiel:

kap09_01.cpp

```
01 class Basisklasse
02 {
03    public:
04       virtual void Methode() = 0; // rein virtuelle Methode
05       // Muss in allen abgeleiteten Klassen definiert werden!
06 };
07
08 int main()
09 {
10    Basisklasse B; // Fehler! Basisklasse ist abstrakt!
11
12    return 0;
13 }
```

Dadurch, dass in der Klasse `Basisklasse` die rein virtuelle Methode nicht implementiert ist, können keine Objekte von dieser Klasse gebildet werden. In den abgeleiteten Klassen hat dies zur Folge, dass die Methode erst definiert werden muss. Erst dann können von den abgeleiteten Klassen auch Instanzen erzeugt werden.

9.2. Abstrakte Basisklasse

Eine Klasse mit zumindest einer rein virtuellen Methode wird abstrakte Klasse genannt, da sie nicht direkt zum Erzeugen von Objekten verwendet werden darf. Der Begriff *Basisklasse* wird meist deshalb verwendet,

da eine abstrakte Klasse immer Basisklasse einer abgeleiteten Klasse sein muss, da von ihr selbst keine Instanzen gebildet werden können.

Die Tatsache, dass von einer abstrakten Basisklasse keine Objekte erzeugt werden können, merkt der Compiler an der Stelle, an der ein konkretes Objekt erzeugt werden soll.

Wenn der Konstruktor einer Basisklasse nicht im öffentlichen, sondern im geschützten Bereich einer Klasse liegt, können auch keine Instanzen der Klasse gebildet werden. In diesem Fall ist die Klasse auch ohne einer rein virtuellen Methode eine abstrakte Klasse.

Beispiel:

kap09_02.cpp

```
01 class Basisklasse
02 {
03    protected:
04       Basisklasse();
05 };
06
07 int main()
08 {
09    Basisklasse B; // Fehler! Konstruktor ist protected!
10
11    return 0;
12 }
```

Das folgende Programm besitzt eine abstrakte Basisklasse Zahl, die eine rein virtuelle Methode Ausgeben deklariert. Für die abgeleiteten Klassen Integer und Gleitkomma muss diese Methode für deren spezielle Erfordernisse implementiert werden. Durch die rein virtuelle Methode muss in jeder abgeleiteten Klasse diese Methode für sich definieren. Eine teilweise Implementation der Methode Ausgeben in der Basisklasse wäre hier sinnlos, da die Ausgabe keine Gemeinsamkeiten besitzt.

Beispiel:

kap09_03.cpp

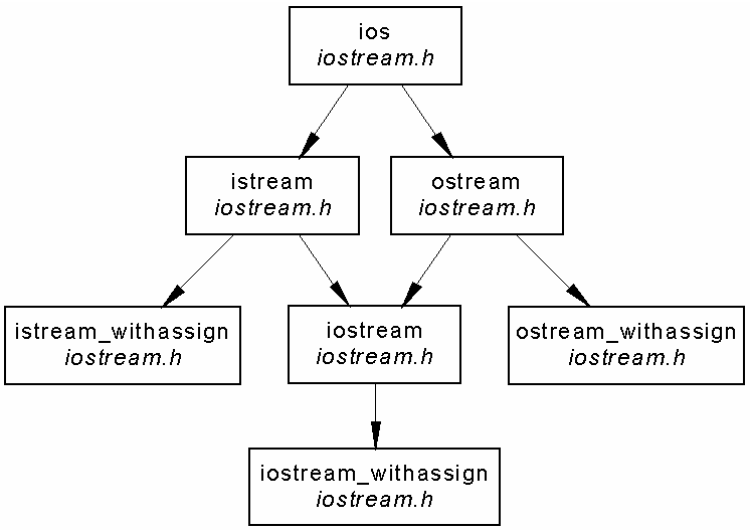
```
01 #include <iostream>
02
03 using namespace std;
04
05 class Zahl
06 {
07    public:
08       virtual void Ausgeben() = 0;
09 };
10
11 class Integer: public Zahl
12 {
13       int i; // private
14    public:
15       Integer(int I)       { i = I;                                    };
16       void Ausgeben()      { cout << "Integerzahl = " << i << endl;    };
17 };
18
19 class Gleitkomma: public Zahl
20 {
21       double d; // private
22    public:
23       Gleitkomma(double D) { d = D;                                    };
24       void Ausgeben()      { cout << "Gleitkommazahl = " << d << endl; };
25 };
```

```
26
27 int main()
28 {
29    Gleitkomma Gl(3.1415);
30    Integer In(17);
31
32    Gl.Ausgeben();
33    In.Ausgeben();
34
35    return 0;
36 }
```

10. Datei- und String-Streams

10.1. Die I/O-Stream-Klassenhierarchie

Die Hierarchie der I/O-Stream-Klassen (I/O: Input/Output) kann wie folgt dargestellt werden:



Die I/O-Stream-Klassenbibliothek besitzt zwei wesentliche Basisklassen: `ios` und `streambuf`, die (bzw. deren abgeleitete Klassen) grundsätzlich die bestimmten Streams darstellen.

Die `ios`-Basisklasse

Die Klasse `ios` (und daher auch alle von ihr abgeleiteten Klassen) enthält einen Zeiger auf ein `streambuf`-Objekt. Ferner sind diverse Status-Variablen der Schnittstelle zu `streambuf` (z.B. Formatierungsflags) und für die Fehlerbehandlung enthalten.

`ios` besitzt zwei abgeleitete Klassen: `istream` (für Eingaben) und `ostream` (für Ausgaben), die bereits im Kapitel *(Datei-)Ein- und Ausgabe in C++* vorgestellt wurden. Die Klasse `istream` unterstützt formatierte und unformatierte Konvertierung von Zeichen, die aus `streambuf` geholt werden. `ostream` unterstützt die formatierte und unformatierte Konvertierung von Daten, die an `streambuf` übergeben werden.

`iostream` ist von `istream` und `ostream` abgeleitet und beinhaltet daher die Formatierungen und Konvertierungen für die Ein- und Ausgabe.

Die abgeleiteten `..._withassign`-Klassen unterstützen vier vordefinierte Standard-Streams, die bereits bekannt sind: `cin`, `cout`, `cerr` und `clog`. Diese Klassen fügen Zuweisungsoperatoren zu ihren jeweiligen Basisklassen hinzu, so dass eine Ein- bzw. Ausgabeumleitung durch den Operator = sehr einfach durchgeführt werden kann.

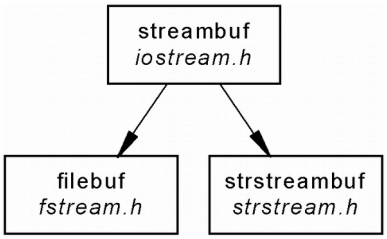
Die Klasse `ios` besitzt verschiedene Konstanten, die im Zusammenhang mit allen oder nur bestimmten Stream-Objekten benötigt werden.

Konstante	**Bedeutung**
`ios::in`	Lese-Modus (Standard bei den i-Stream-Klassen, z.B. `istream`)
`ios::out`	Schreibe-Modus (Standard bei den o-Stream-Klassen, z.B. ostream)
`ios::app`	Daten an den Stream anhängen
`ios::ate`	an das Ende des Streams gehen

`ios::trunc`	alten Inhalt des Streams löschen
`ios::nocreate`	eine Datei muss vorhanden sein (Datei-Streams)
`ios::noreplace`	eine Datei darf nicht vorhanden sein (Datei-Streams)
`ios::binary`	öffnet eine Datei im binären Modus (`'\n'`, `'\0'`, ... werden als normale Zeichen gelesen)

Streambuf

Die Stream-Klassen selbst verwenden ferner Objekte folgender wichtiger Klassen für Pufferungszwecke:



Die `streambuf`-Klasse bietet allgemeine Methoden zum Puffern und Bearbeiten von Streams mit nur wenigen bzw. keinen Formatierungsmöglichkeiten. Puffern heißt, dass nicht einzelne Zeichen an den Stream gesendet werden, sondern dass die Daten blockweise übertragen werden. `streambuf` ist eine nützliche Basisklasse, die von anderen Teilen der I/O-Stream-Klassen benutzt wird. Als wesentliche, konkrete Klassen sind `filebuf` (Pufferung für Ein- und Ausgabe mit Dateien) und `strstreambuf` (Pufferung für Ein- und Ausgabe mit Strings im Speicher) von `streambuf` abgeleitet. Die meisten Elementfunktionen dieser Klassen sind `inline`-Funktionen und dadurch sehr effizient.

Da die Ausgabe erst erfolgt, wenn eine Blockeinheit übertragen werden kann (z.B. wenn der Puffer voll ist), können keine bestimmten Voraussetzungen an die Ein- und Ausgabe geknüpft werden (z.B. dass bestimmte Zeichen schon ausgegeben wurden). Dies kann bei Programm- bzw. Rechnerabstürzen unangenehme Auswirkungen haben, da die gepufferten Informationen verlorengehen. Alle Stream-Objekte besitzen daher eine Methode `flush()`, die das explizite Leeren des Puffers bewirkt.

Beispiel:

kap10_01.cpp

```
01 #include <iostream>
02 #include <fstream>
03
04 using namespace std;
05
06 int main()
07 {
08    ofstream outf("logfile.txt");
09
10    outf << "Ausgabe-Protokoll";
11    outf.flush();
12
13    return 0;
14 }
```

Umleiten der Standardkanäle

Die Standardkanäle `cin`, `cout`, `cerr` und `clog` können in einen anderen Stream umgeleitet werden. Um die Umleitung vorzunehmen, ist eine Zuweisung notwendig. Die Klassen `..._withassign` besitzen diese Eigenschaft, d.h. `cin`, `cout`, `cerr` und `clog` sind nicht direkt von den `...stream`-Klassen, sondern von den `..._withassign`-Klassen abgeleitet.

Beispiel:

In diesem Beispiel wird die Standardausgabe in die Datei `logfile.txt` umgeleitet. Beachten Sie, dass das Objekt `cout` ungültig wird, wenn der Block des Objekts `LogFile` verlassen wird, da die Datei dann geschlossen wird. Darum benötigt man ein temporäres Objekt, in dem der aktuelle Ausgabe-Stream gespeichert wird.

kap10_02.cpp

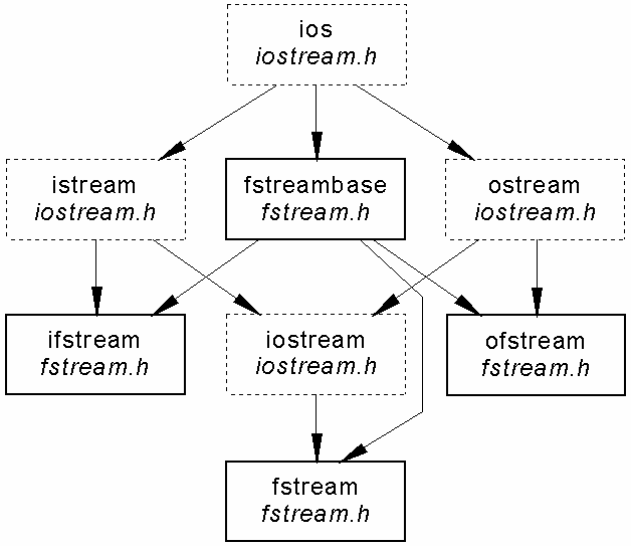
```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
   ofstream LogFile ("logfile.txt");

   // mit GNU C++-Compiler (Linux):
   _IO_ostream_withassign Tmp_cout;

   // mit MS Visual C++-Compiler:
   // ostream_withassign Tmp_cout;
   // Ferner müssen mit dem Visual C++-Compiler
   // die Headerdateien iostream.h und fstream.h
   // verwendet werden (ohne namespace)!

   cout << "Umleitung der Standardausgabe" << endl;

   Tmp_cout = cout;
   cout = LogFile;

   cout << "Hallo Welt!" << endl;

   cout = Tmp_cout;
   cout << "Test beendet!" << endl;

   return 0;
}
```

Hinweis:

Die Klassen `..._withassign` sind in den verschiedenen Compilern unterschiedlich implementiert. Das Beispielprogramm funktioniert mit dem GNU C++-Compiler unter Linux. Dem Klassennamen muss hier noch ein `_IO_` voran gesetzt werden. Mit MS Visual C++ müssen dagegen die Headerdateien mit der Dateiendung `.h` verwendet werden (entsprechend wird der `namespace` nicht benötigt!).

10.2. Datei-Streams

Die Klassen `ifstream` (Eingabe), `ofstream` (Ausgabe) und `fstream` (Ein- und Ausgabe) werden für die Ein- und Ausgabe mit Dateien verwendet. Für die Dateimanipulation spezifische Funktionen sind in der gemeinsamen Basisklasse `fstreambase` zusammengefasst.



Die Konstruktoren können gleich mit einem Dateinamen aufgerufen werden, so dass die betreffenden Dateien gleich geöffnet werden. Verlässt das Objekt seinen Gültigkeitsbereich, wird die Datei automatisch über den Destruktor der Klasse wieder geschlossen. Alternativ kann vorher die Methode `close()` aufgerufen werden.

Für das Öffnen einer Datei können verschiedene Konstanten der Klasse `ios` (siehe auch voriger Abschnitt) eingesetzt werden. Standardmäßig wird z.B. beim Öffnen einer Datei deren voriger Inhalt überschrieben.

Beispiel:

Die folgende Anweisung öffnet z.B. die Datei `test.txt` zum Anfügen weiterer Daten im Binärmodus.

```
ofstream outf ("test.txt", ios::app | ios::binary);
```

Die Klasse `ios` besitzt zu den oben angegebenen Konstanten einige Methoden, die insbesondere bei der Arbeit mit Datei-Streams interessant sind. Alle liefern gleich 0 als falsch und ungleich 0 als wahr.

- Methode `eof()`:

  Syntax: `int eof();`

  Beschreibung: Liefert einen Wert ungleich 0, wenn das Ende des Datenstroms erreicht wurde.

- Methode `good()`:

  Syntax: `int good();`

  Beschreibung: Liefert einen Wert ungleich 0, wenn die letzte Operation auf dem Datenstrom keinen Fehler verursacht hat.

- Methode `bad()`:

  Syntax: `int bad();`

  Beschreibung: Liefert einen Wert ungleich 0, wenn die letzte Operation einen ernsthaften Fehler verursacht hat. Es wird empfohlen, die Ein- bzw. Ausgabe auf dem Datenstrom zu beenden.

- Methode `fail()`:

Syntax: `int fail();`

Beschreibung: Liefert einen Wert ungleich 0, wenn die letzte Operation auf dem Datenstrom einen Fehler verursacht hat. Wenn der Aufruf der Methode `bad()` in diesem Fall gleich 0 ist, kann unter Umständen die Ein- bzw. Ausgabe auf dem Datenstrom wiederholt bzw. fortgesetzt werden, nachdem die Fehlerbits mit der Methode `clear()` gelöscht wurden.

- Methode `rdstate()`:

  Syntax: `int rdstate();`

  Beschreibung: Liefert den aktuellen Fehlerstatus der letzten Operation auf dem Datenstrom. Dafür sind folgende Konstanten in der Klasse `ios` definiert.

Konstante	Beschreibung
`ios::goodbit`	kein Fehler aufgetreten
`ios::eofbit`	Ende des Datenstroms erreicht
`ios::failbit`	unkritischer Fehler
`ios::badbit`	kritischer Fehler oder unbekannter Status

Beispiel:

```
ofstream Datei ("test.txt");
int Status = Datei.rdstate();
if (Status | ios::eofbit)
   cout << "Dateiende erreicht!" << endl;
```

- Methode `clear()`:

  Syntax: `int clear(int Status = 0);`

  Beschreibung: Setzt den Fehlerstatus auf den angegebenen Status; mögliche Werte: siehe Methode `rdstate()`. Wird der Parameter weggelassen, wird der Fehlerstatus auf 0 gesetzt, d.h. alle Fehlerbits werden gelöscht.

Beispiel:

Das folgende Beispielprogramm öffnet die Datei `test.txt`, die im aktuellen Verzeichnis erstellt wird, über die Angabe des Dateinamens im Konstruktor. Sie soll zur Ausgabe von einer Textzeile dienen. Danach wird die Datei geschlossen und noch einmal zum Einlesen geöffnet. Diesmal wird zum Öffnen der Aufruf der Methode `open()` verwendet.

kap10_03.cpp

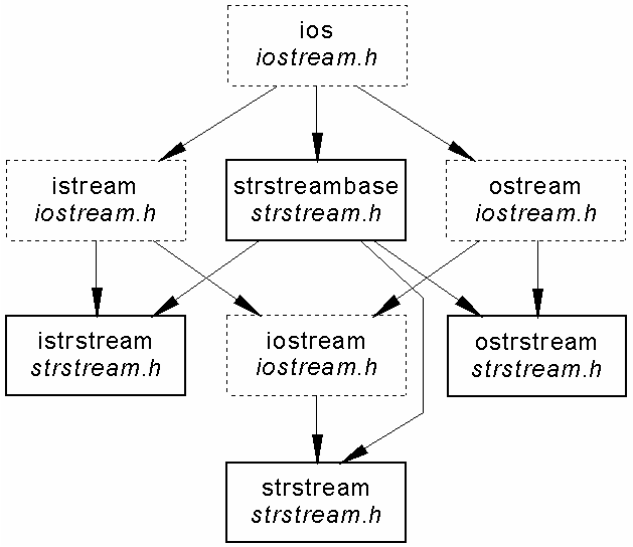
```
01 #include <iostream>
02 #include <fstream>
03
04 using namespace std;
05
06 int main()
07 {
08    char str[100];
09    ofstream outf ("test.txt");
10    ifstream inf;
11
12    if (outf)
13       outf << "Hallo Welt!" << endl;
14    outf.close();
15
16    inf.open("test.txt");
```

```
17     while (inf.getline(str, 100))
18        cout << str << endl;
19     inf.close();
20
21     return 0;
22  }
```

10.3. String-Streams

Anstatt in eine Datei kann die Ausgabe durch String-Streams auch in einen Speicherbereich (einen String) erfolgen. So können Strings aus Daten zusammengestellt werden, ohne dass diese unmittelbar in eine Datei ausgegeben werden müssen (vgl. `sprintf` in C). Umgekehrt ist es auch möglich, mit Hilfe von String-Streams Zeichenketten in einzelne Datenelemente zu zerlegen (vgl. `sscanf` in C).

String-Streams bieten jedoch einen wesentlichen flexibleren Mechanismus als die Funktionen `sprintf` und `sscanf`. Sie können (fast genau) wie Datei-Streams verwendet werden und werden daher auch gern für Pufferungszwecke - statt temporärer Dateien - verwendet.



Grundsätzlich werden in der Header-Datei `strstream.h` folgende Stream-Klassen definiert:

`istrstream`	Eingabestream zum Lesen von einem String (Zerlegen eines Strings)
`ostrstream`	Ausgabestream zum Schreiben in einen String (Zusammenstellen eines Strings)
`strstream`	Ein- und Ausgabestream zum Lesen und Schreiben von bzw. in einen String (Kombination von istrstream und ostrstream)

Zerlegen von Strings

Der Konstruktor eines `istrstream`-Objekts verlangt als Parameter einen String (`char *`). Hier sollte der zu zerlegende String angegeben werden. Anschließend können Daten von diesem String, wie bei Eingabe-Streams üblich, eingelesen werden. Das Zeichen `'\n'` wird als Endezeichen interpretiert.

Beispiel:

In diesem Beispiel soll der String `s` zerlegt werden. Die einzelnen Datenelemente des Strings sind durch Leerzeichen (Whitespaces) getrennt. Dadurch wird die Leseoperation für ein Datenelement beendet und das nächste Element kann gelesen werden.

kap10_04.cpp

```
01 #include <iostream>
02 #include <strstream>
03
04 using namespace std;
05
06 int main()
07 {
08    char s[] = "123 123.456 Hallo Z";
09    int i;
10    double d;
11    char c, t[20];
12    istrstream str(s);
13
14    str >> i >> d >> t >> c;
15
16    cout << "i = " << i << endl;
17    cout << "d = " << d << endl;
18    cout << "t = " << t << endl;
19    cout << "c = " << c << endl;
20
21    return 0;
22 }
```

Das Beispielprogramm liest folgende Werte in die Variablen:

```
i = 123
d = 123.456
t =: "Hallo"
c = 'Z'
```

Zusammensetzen von Strings

`ostrstream`-Objekte können auf zwei verschiedene Arten erzeugt werden: Mit oder ohne der Angabe eines Zielspeicherbereichs. Wird kein Zielspeicherbereich angegeben, so wird dynamisch Speicherplatz reserviert und dieser bei längeren Ausgaben auch entsprechend erweitert, so dass kein Überlauf eintritt.

Dynamische String-Ausgabe-Streams

Bei der Definition eines solchen `ostrstream`-Objekts wird kein Initialisierungsparameter angegeben. Die letzte Ausgabe auf einen solchen Stream sollte '`\0`' (oder äquivalent: `ends`) sein, damit der String - wie in C / C++ üblich - durch eine Null abgeschlossen wird. Über die parameterlose Methode `str()` kann der Zeiger auf den zusammengestellten String ermittelt werden. Diese Methode sperrt gleichzeitig den gesamten String für weitere Ausgaben (solche würden dann ein Fehlerflag setzen). Die Anzahl der geschriebenen Bytes (inklusive der abschließenden Null) kann mit der Methode `pcount()` ermittelt werden. Für das Freigeben des dynamisch erzeugten String ist anschließend nicht mehr das Stream-Objekt, sondern das Anwenderprogramm zuständig.

Beispiel:

kap10_05.cpp

```
01 #include <iostream>
02 #include <strstream>
03
04 using namespace std;
05
06 int main()
07 {
```

```
08    ostrstream outf; // dynamischen String-Ausgabe-Stream erzeugen
09
10    // Stream wie ueblich beschreiben
11    outf << 2.2 * 2 << " abc";
12    outf << ends; // als Stringabschluss noch eine 0!
13
14    //String-Stream an String Puffer uebergeben
15    char *Puffer = outf.str();
16
17    cout << "Anzahl geschriebene Bytes: " << outf.pcount() << endl;
18    cout << "zusammengestellter String: " << Puffer << endl;
19
20    delete [] Puffer; // dynamischen String wieder freigeben
21
22    return 0;
23 }
```

Das Programm erzeugt die folgende Ausgabe:

```
Anzahl geschriebene Bytes: 8
zusammengestellter String: 4.4 abc
```

String-Ausgabe-Streams mit angegebenem Zielspeicherbereich

Bei dieser Art von String-Ausgabe-Streams werden der Zielspeicherbereich in Form eines `char *`, die Größe dieses Bereichs und optional ein Modus für das Öffnen angegeben. Wird als Öffnen-Modus `ios::ate` oder `ios::app` angegeben, werden Ausgaben an den übergebenen String (ab dem ersten '`\0`') angehängt.

Beispiel:

kap10_06.cpp

```
01 #include <iostream>
02 #include <strstream>
03
04 using namespace std;
05
06 int main()
07 {
08    char Puffer[100];
09    ostrstream outf(Puffer, sizeof(Puffer));
10
11    // Stream wie ueblich beschreiben
12    outf << 2.2 * 2 << " abc";
13    outf << ends; // als Stringabschluss noch eine 0!
14
15    cout << "Anzahl geschriebene Bytes: " << outf.pcount() << endl;
16    cout << "zusammengestellter String: " << Puffer << endl;
17
18    return 0;
19 }
```

Das Programm erzeugt die gleiche Ausgabe wie das vorige Beispielprogramm.

Kombination von String-Ein- und Ausgabe

Die Klasse `strstream` stellt eine Kombination der Klassen `istrstream` und `ostrstream` dar. Der Pufferspeicher kann wie bei `ostrstream` entweder bei der Initialisierung (mit Größe und eventuell Öffnen-Modus) angegeben oder automatisch dynamisch erzeugt werden.

Ein solcher String-Stream kann grundsätzlich wie ein Datei-Stream mit Ein- und Ausgabemöglichkeit verwendet werden. Einziger Unterschied: Ein Datei-Stream besitzt eine gemeinsame Schreib-/Leseposition, während ein äquivalenter String-Stream eine Leseposition und eine davon unabhängige Schreibposition besitzt. So kann bequem an einer Stelle geschrieben und an einer anderen gelesen werden, ohne dass eine ständige Neupositionierung notwendig wäre.

Die Klasse `strstream` wird gern für Pufferungszwecke, statt langsamer temporärer Dateien, eingesetzt. Allgemein können String-Streams auch als "Dateien im Hauptspeicher" angesehen werden.

11. Überladen von Operatoren

Mit Hilfe des Überladen von Operatoren können auf Objekte selbstdefinierter Klassen (fast) alle Operatoren von C++ mit einer neuen Bedeutung versehen werden. Diese Bedeutung wird durch spezielle Funktionen bzw. Methoden festgelegt. Der Gedanke ist dabei der gleiche wie beim Überladen von Funktionen und Methoden. Das Überladen von Operatoren wird allgemein auch als Operator-Overloading bezeichnet.

Sie haben das Überladen von Operatoren bei der Daten-Ein- und Ausgabe bereits indirekt kennengelernt. Mittels des Ausgabeoperators << können Objekte verschiedener Typen auf einem Ausgabe-Datenstrom ausgegeben werden. Der Eingabeoperator >> dient dem Einlesen von Objekten verschiedener Typen von einem Eingabe-Datenstrom.

Um den Mechanismus genau zu verstehen, ist etwas Vorarbeit notwendig. Prinzipiell existiert für jeden Operator auch eine entsprechende Funktionsnotation (deren explizite Verwendung allerdings bei den meisten C++-Compilern für Objekte nicht selbstdefinierter Klassen untersagt ist). Das heißt, für die Standarddatentypen wie `int`, `float`, `double` usw. sind z.B. die Operatoren +, -, *, /, = usw. definiert. Für z.B. Zeichenketten wie `char[100]` (C-Strings) ist dagegen kein Operator definiert. Sie könnten sich nun eine Klasse `CString` definieren, die die Zuweisung von Zeichenketten über den Operator = erlaubt.

Genaugenommen ist in C++ die Operator-Notation nur eine andere Schreibweise für bestimmte Funktionsaufrufe. Die vordefinierten Datentypen kann man auch als "vordefinierte Klassen" verstehen, für die bereits Operator-Funktionen definiert sind.

Beispiele:

Operator-Schreibweise	**Funktionsaufruf-Schreibweise**
`a + b`	`operator+ (a, b)`
`cout << a`	`operator<< (cout, a)`
`!x`	`operator! (x)`
`a += b`	`operator+= (a, b)`
`f[3]`	`operator[] (f, 3)`
`func (a, b, c)`	`operator() (func, a, b, c)`
`cout << a << b`	`operator<< (operator<< (cout, a), b))`
`-(a + b * 3)`	`operator- (operator+ (a, operator* (b, 3)))`

Das Prinzip des Überladens von Operatoren besteht nun darin, dass diese Operator-Funktionen wie normale Funktionen überladen werden können. Die einzige Bedingung dabei ist, dass zumindest ein Argument den Typ einer (selbstdefinierten) Klasse oder eine Referenz darauf besitzt.

Auf diese Weise können grundsätzlich fast alle Operatoren überladen werden. Folgende Operatoren können nicht überladen werden:

```
.
.*
::
sizeof
?:
```

Die Ausführungsreihenfolgen und Prioritäten der Operatoren können nicht geändert werden. Außerdem können keine eigenen Operatoren definiert werden, sondern Sie müssen auf die vorhandenen Operatoren zurückgreifen. Allerdings werden Ihnen keine Vorschriften gemacht, wie ein Operator einzusetzen ist (der Operator + kann z.B. durch eine eigene Definition eine völlig andere Bedeutung - z.B. die der Multiplikation - erlangen).

Es gibt verschiedene Möglichkeiten, einen Operator zu überladen: Einerseits über `friend`-Funktionen der betreffenden Klasse oder über direkte Methoden der Klasse. Im folgenden Abschnitt wird die erste Möglichkeit betrachtet, die zweite Möglichkeit wird im darauf folgenden Abschnitt behandelt.

11.1. Überladen von Operatoren durch friend-Funktionen

Im folgenden Beispiel soll für eine Klasse `complex`, die zur Darstellung komplexer Zahlen dient, der Operator + so überladen werden, dass zwei komplexe Zahlen auf die bei Standardtypen wie `int` oder `double` übliche Weise addiert werden können.

Die Operator-Funktion für den Operator + hat grundsätzlich folgendes Aussehen:

```
complex operator+ (complex a, complex b);
```

Das folgende Programm demonstriert den Einsatz von `friend`-Funktionen zum Überladen von Operatoren. Die Operator-Funktion muss auf private Elemente der Klasse `complex` zugreifen. Daher ist es notwendig, sie in der Definition der Klasse als Freund zu deklarieren.

Beispiel:

kap11_01.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 class complex
06 {
07    private:
08       double re, im; // Real- und Imaginaerteil
09    public:
10       complex(): re(0), im(0) {}
11       complex(double r, double i): re(r), im(i) {}
12       friend complex operator+ (const complex &a, const complex &b);
13       void Ausgabe()
14       {
15          cout << "re = " << re << endl;
16          cout << "im = " << im << endl;
17       }
18 };
19
20 complex operator+ (const complex &a, const complex &b)
21 {
22    complex c;
23
24    c.re = a.re + b.re;
25    c.im = a.im + b.im;
26
27    return c;
28 }
29
30 int main()
31 {
32    complex a(1, 1), b(2, 2), c;
33
34    c = a + b; // ist identisch mit
35    c = operator+ (a, b);
36    c.Ausgabe();
37
```

```
38     return 0;
39 }
```

Da Objekte der Klasse `complex` immerhin eine Größe von 16 Byte haben, ist es aus Zeit- und Speicherplatzgründen günstiger, die Parameter der Operator-Funktion als Referenzparameter zu deklarieren.

Dieses Programm enthält eine Methode `Ausgabe` für die Ausgabe einer komplexen Zahl. Besser wäre es jedoch, die Operatoren << und >> zu überladen und damit die Ausgabe über die üblichen Operatoren durchzuführen. Hier stellt sich die Frage, welche Typen die Parameter der Operator-Funktionen für << und >> haben.

Die Operator-Notation des Ausdrucks "`cout << a`" ist folgende:

```
operator<< (cout, a)
```

Der erste Parameter ist somit der Ausgabe-Datenstrom (`class ostream`), der zweite das auszugebende Objekt. Um den Operator << auch mehrfach hintereinander verwenden zu können, ist es notwendig, als Funktionsergebnis wieder den Ausgabe-Datenstrom selbst zurückzuliefern. Aus `cout << a << b;` wird dann `operator<< (operator<< (cout, a), b);`.

Da Datenstromobjekte meist aus vielen Daten bestehen und diese ferner bei einer Ausgabe auch verändert werden müssen, ist es hier notwendig, das Datenstromobjekt als Referenzparameter zu deklarieren. (Deshalb sind die Referenzen überhaupt in C++ eingeführt worden!) Dass das Datemstromobjekt nicht kopiert wird, wenn es als Funktionsergebnis bei einem geschachtelten Aufruf zurückgeliefert wird, kann dadurch erreicht werden, dass ebenfalls nur eine Referenz auf das Objekt und nicht das Objekt selbst zurückgeliefert wird.

Der Operator << würde also für die Klasse `complex` am besten wie folgt überladen werden. Der Aufruf der Methode `Ausgabe` kann nun durch `cout << c;` ausgetauscht werden.

Beispiel:

kap11_02.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 class complex
06 {
07    private:
08       double re, im; // Real- und Imaginaerteil
09    public:
10       complex(): re(0), im(0) {}
11       complex(double r, double i): re(r), im(i) {}
12       friend complex operator+ (const complex &a, const complex &b);
13       friend ostream &operator<< (ostream &ostr, const complex &a);
14       friend istream &operator>> (istream &istr, complex &a);
15 };
16
17 complex operator+ (const complex &a, const complex &b)
18 {
19    complex c;
20
21    c.re = a.re + b.re;
22    c.im = a.im + b.im;
23
24    return c;
25 }
26
27 ostream &operator<< (ostream &ostr, const complex &a)
28 {
29    ostr << '(' << a.re << " + i * " << a.im << ')';
```

```
30
31     return ostr;
32 }
33
34 istream &operator>> (istream &istr, complex &a)
35 {
36     istr >> a.re >> a.im;
37
38     return istr;
39 }
40
41 int main()
42 {
43     complex a(1, 1), b(2, 2), c;
44
45     c = a + b;
46     cout << c << endl;
47
48     return 0;
49 }
```

11.2. Überladen von Operatoren durch Methoden

Eine Operator-Funktion kann auch als Methode einer Klasse implementiert werden, wenn das erste Argument ein Objekt dieser Klasse ist. Dieses erste Argument wird dann nicht mehr angegeben, da es für die Operator-Methode das aktuelle Objekt darstellt.

Zu der oben vorgestellten Klasse `complex` soll der binäre Operator - (Subtraktion zweier komplexer Zahlen) in Form einer Operator-Methode hinzugefügt werden. Folgendes muss hinzugefügt werden:

kap11_03.cpp (komplettes Beispielprogramm)

```
05 class complex
06 {
         ... // siehe oben
15       complex operator- (const complex &a);
16 };

42 complex complex::operator- (const complex &a)
43 {
44     complex c;
45
46     c.re = re - a.re;
47     c.im = im - a.im;
48
49     return c;
50 }
```

Im folgenden werden noch der unäre Operator - für die Vorzeichenumkehr und der Operator += für die Klasse `complex` definiert. Bei einem unären Operator wird kein Parameter benötigt.

kap11_04.cpp (komplettes Beispielprogramm)

```
05 class complex
06 {
      ... // siehe oben
16       complex operator- ();
17       complex operator+= (const complex &a);
18 };
```

```
54 complex complex::operator- ()
55 {
56    complex c;
57
58    c.re = -re;
59    c.im = -im;
60
61    return c;
62 }
63
64 complex complex::operator+= (const complex &a)
65 {
66    re += a.re;
67    im += a.im;
68
69    return *this;
70 }
```

Da das Ergebnis des Operators += das Objekt selber ist, wird mit Hilfe des `this`-Zeigers das Objekt selber zurückgeliefert. Hier sehen Sie auch eine notwendige Verwendung des `this`-Zeigers.

Hinweis:

Die Operatoren << und >> können nicht als Operator-Methoden einer Klasse geschrieben werden, da das erste Argument dieser Operatoren kein Objekt der Klasse, sondern das entsprechende Datenstromobjekt ist.

11.3. Allgemeines

Werden die Operatoren ++ oder -- überladen, so kann oft nicht zwischen der Prefix- und Postfix-Notation unterschieden werden. Für Objekte eigener Klassen sind somit folgende Ausdrücke äquivalent (und bedeuten Prefix-Notation):

```
b = a++;
b = ++a;
```

Einige Compiler - so auch der GNU-C++-Compiler - ermöglichen allerdings die Definition eigener Postfix-Operatoren für das Inkrementieren und Dekrementieren. Dabei wird ein normalerweise mit 0 belegter und nicht verwendeter Parameter mit dem Typ `int` definiert. (Ob dies ein so geschickter Kunstgriff ist, sei dahin gestellt.) Im folgenden werden Prefix- und Postfix-Operatoren für die Klasse `complex` vorgestellt.

 *kap11_05.cpp* (komplettes Beispielprogramm)

```
005 class complex
006 {
      ... // siehe oben
018       complex operator++ ();    // Prefix-Operator
019       complex operator++ (int); // Postfix-Operator
020 };

074 complex complex::operator++ ()
075 {
076    ++re;
077    cout << "Prefix" << endl;
078
079   return *this;
080 }
081
082 complex complex::operator++ (int a)
083 {
```

```
084     re++;
085     cout << "Postfix" << endl;
086
087     return *this;
088 }
```

Da die Operatoren aber immer in der gleichen Reihenfolge abgearbeitet werden, machen die beiden Operatoren keinen Unterschied. So kommt bei den beiden folgenden Berechnungen immer das gleiche Ergebnis:

```
090 int main()
091 {
092     complex a(1, 1), b(2, 2), c, d;
093
094     d = a;
095     c = d++ + b;
096     cout << c << endl; // ergibt 4 + 3i
097     d = a;
098     c = ++d + b;
099     cout << c << endl; // ergibt ebenfalls 4 + 3i
100
101     return 0;
102 }
```

Das Überladen von den Operatoren `()`, `->`, `new` und `delete` stellen Spezialfälle dar, die in den nachfolgenden Abschnitten erläutert werden. Zuvor noch ein Abschnitt zum Thema Typumwandlung.

11.4. Typumwandlungs-Operatoren

In einem der vorigen Abschnitte wurde der Operator + für die Klasse `complex` überladen, so dass er auch zur Addition zweier komplexer Zahlen verwendet werden kann. Nach wie vor nicht besprochen ist, wie z.B. eine Addition einer komplexen Zahl mit einer `int`- oder `double`-Zahl vorgenommen wird.

```
complex a(1,1), b(2,2), c;

c = a + b;       // OK!
c = a + 17;      // Fehler! Operation ist nicht definiert!
c = a + 3.1415; // Fehler! Operation ist nicht definiert!
```

Um auch diese Formen der Addition zu ermöglichen, können grundsätzlich entsprechende Operator-Funktionen definiert werden. Die Klassendefinition von `complex` bräuchte dann 5 Operator-Funktionen allein für die Addition. Entsprechend kämen dann noch eine ganze Menge Operator-Funktionen für die anderen Rechenarten wie Subraktion, Multiplikation, Division, usw.

```
class complex
{  // siehe oben
   public:
      // für die Addition:
      friend complex operator+ (complex &a, complex &b);
      friend complex operator+ (complex &a, int b);
      friend complex operator+ (int a, complex &b);
      friend complex operator+ (complex &a, double b);
      friend complex operator+ (double a, complex &b);
};

// Die deklarierten Funktionen müssen natürlich
// auch noch definiert werden!
```

Wie Sie sehen, wären sehr viele Einzelfunktionen notwendig, um das Problem auf diese Art zu lösen. Wenn Sie sich jetzt vorstellen, dass auf einen abstrakten Datentyp wie der Klasse der komplexen Zahlen noch sehr

viel mehr Operatoren anwendbar sein müssten und evtl. auch andere Typen akzeptiert werden sollten, so erscheint der Aufwand dann zu hoch zu werden.

In C werden grundsätzlich Ausdrücke, bei denen verschiedene Typen gemischt vorkommen, mittels automatischer Typumwandlung übersetzt.

Beispiel:

Der folgende Ausdruck wird mittels automatischer (impliziter) Typumwandlung folgendermaßen ausgewertet:

```
10 + 3L - 123.456
```
(10 - `int`-Konstante, 3 - `long`-Konstante, 123.456 - `double`-Konstante)

1. Schritt: Umwandlung der `int`-Konstante `10` in einen `long`-Wert: `10L`

2. Schritt: `long`-Addition: `10L + 3L = 13L`

3. Schritt: Umwandlung des `long`-Werts `13L` in einen `double`-Wert: `13.0`

4. Schritt: `double`-Subtraktion: `13.0 - 123.456 = -110.456`

Es findet also eine automatische Typumwandlung vor der Durchführung der Operationen statt. Die Typumwandlung kann auch mittels expliziter Typumwandlung geschrieben werden:

```
(double) ((long) 10 + 3L) - 123.456
```

Während bei dem obigen Ausdruck die explizite Typumwandlung nicht notwendig ist, da die Typumwandlungen auch automatisch entsprechend richtig ausgeführt werden, sind im folgenden Beispiel explizite Typumwandlungen für die richtige Ausführung notwendig, da sonst ein Integer-Überlauf ein falsches Ergebnis liefern würde:

```
cout << "1000000 * 1000000 = ";
cout << (double) 1000000 * (double) 1000000 << endl;
```

In C++ können Typumwandlungen auch in einer anderen, den Funktionsaufrufen ähnlichen Schreibweise geschrieben werden, die aus Gründen der Einheitlichkeit und besseren Übersichtlichkeit vorzuziehen ist.

C:	`(int) d`	C++:	`int (d)`
	`(long) (a + b)`		`long (a + b)`
	`(char *) p`		`(char *) (p)`

Implizite Typumwandlungen sollten möglichst vermieden und stattdessen explizit vorgenommen werden, da hierdurch Möglichkeiten für viele schwer zu findende Fehler ausgeschaltet werden können.

Auf das Beispiel mit der Klasse `complex` angewendet heißt das, dass es sinnvoller wäre, anstatt vieler verschiedener Operator-Funktionen mit allen möglichen Kombinationen von Datentypen nur grundsätzlich eine Operator-Funktion pro Operator für alle Datentypen zu schreiben. Trotzdem soll auf den Mechanismus der automatischen oder expliziten Typumwandlung nicht verzichtet werden.

Doch wie können Variablen anderer Typen automatisch oder explizit in Objekte der Klasse `complex` umgewandelt werden?

Konstruktoren als Typumwandlungs-Operatoren

Grundsätzlich stellt jeder Konstruktor mit genau einem Argument eine Beschreibung dar, wie aus einer Variablen mit dem Typ des Arguments ein Objekt der entsprechenden Klasse erzeugt werden kann. Diese Beschreibung wird von C++ auch für die Typumwandlung verwendet.

Schließlich wird ja bei der Auswertung eines Teilausdrucks wie `complex(3)` ein temporäres Objekt erzeugt. Und beim Erzeugen eines Objekts wird der Konstruktor ausgeführt.

Für die Klasse `complex` bedeutet dies, dass nun der folgende Programmcode hinzugefügt werden kann.

kap11_06.cpp (komplettes Beispielprogramm)

```
005 class complex
006 {
        ... // siehe oben
010        // Konstruktoren:
010        complex(): re(0), im(0) {}
011        complex(double r, double i): re(r), im(i) {}
013        complex(int i);            // int -> complex
014        complex(double d);         // double -> complex

023 };
024
025 complex::complex(int i)
026 {
027    re = i;
028    im = 0;
029 }
030
031 complex::complex(double d)
032 {
033    re = d;
034    im = 0;
035 }
```

Nun lassen sich implizite und explizite Typumwandlungen durchführen:

```
105 int main()
106
107    complex a(1, 1), b(2, 2), c;
108
109    c = a + a;                      // -> complex + complex
110    c = complex(3);                 // -> complex(int)
111    c = 3;                          // -> complex(int)
112    c = complex(4.6);               // -> complex(double)
113    c = 4.6;                        // -> complex(double)
114    c = a + 3;                      // -> complex + complex(int)
115    c = a + 4.6;                    // -> complex + complex(double)
116    c = 3 - b;                      // -> complex(int) - complex
117    c = complex(3) + complex(4.6); // -> complex(int) + complex(double)
118    c = complex(3 + 4.6);           // -> complex(double(int) + double)
119
120    return 0;
121 }
```

Hinweis:

Die Zeile `c = 3 - b` funktioniert nur, wenn der `-`-Operator als `friend`-Funktion überladen wurde, da sonst die implizite Typumwandlung nicht funktioniert.

Typumwandlungsoperator-Methoden

Mittels Konstruktoren mit nur einem Parameter ist es möglich, beliebige andere Variablen in Objekte der eigenen Klasse umzuwandeln. Oft ist es aber auch wünschenswert, Objekte der eigenen Klasse in Variablen anderer Typen (z.B. `int`, `double`, usw.) umzuwandeln. Sogenannte Typumwandlungsoperator-Methoden erfüllen genau diese Aufgabe.

Eine Typumwandlungsoperator-Methode ist eine Methode, welche keine Argumente und keinen explizit angegebenen Ergebnistyp besitzt und deren Name aus dem Schlüsselwort `operator` und dem Namen des

Zieltyps besteht. Der Ergebnistyp solcher Methoden ist bereits durch den Namen vorgegeben. Der Code dieser Methode muss die Umwandlung des aktuellen Objekts in den geforderten Zieltyp beinhalten. Typumwandlungsoperator-Methoden sollten wie Konstruktoren immer `public` deklariert werden.

Im folgenden Beispiel wird die Klasse `complex` um eine Typumwandlungsoperator-Methode erweitert, die eine komplexe Zahl in eine `double`-Zahl umwandelt. Dabei soll der `double`-Wert gleich dem Realteil sein.

kap11_07.cpp (komplettes Beispielprogramm)

```
005 class complex
006 {
        ... // siehe oben
014        operator double() { return re; }

023 };

105 int main()
106 {
107    complex a(1, 1), b(2, 2);
108    double d;
109
110    d = double(a);              // -> double(complex)
111    d = a;                      // -> double(complex)
112    d = a + b;                  // -> double(complex + complex)
113    cout << double(a) << endl; // -> double(complex)
114
115    return 0;
116 }
```

Eine implizite Typumwandlung erfolgt immer bei der Parameterübergabe an Funktionen. Da Operationen in Ausdrücken eigentlich auch Funktionen sind, werden diese Typumwandlungen auch dort durchgeführt. Dabei versucht der Compiler, die Parametertypen immer so umzuwandeln, dass es eine Funktion dafür gibt. Allerdings kann es dabei auch zu Mehrdeutigkeiten kommen, wie das folgende Beispiel zeigt.

kap11_08.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 class Test
06 {
07    public:
08       Test(int i);
09       operator int();
10       Test operator+ (const Test &a);
11 };
12
13 int main()
14 {
15    Test a(2);
16
17    a = a + 3; // 2 Moeglichkeiten:
18               // Test::operator+ (a, Test(3))   oder
19               // int::operator+ (int(a), 3)
20
21    return 0;
22 }
```

Der Compiler muss an der Stelle `a = a + 3` eine Mehrdeutigkeit feststellen, da es verschiedene Möglichkeiten des Einsatzes von Operatoren gibt. Der Programmierer muss durch eine explizite Typumwandlung einen Parameter umwandeln, z.B. `a = a + Test(3)`.

11.5. Kopieren von Objekten

Zum Kopieren von Objekten wird grundsätzlich der Operator = verwendet. Dabei werden normalerweise alle Datenkomponenten des Quellobjekts im Verhältnis 1:1 in das (bereits existierende) Zielobjekt übertragen.

Verwendet ein Objekt auch dynamische Daten, welche z.B. im Konstruktor erzeugt und im Destruktor gelöscht werden, so werden die Daten selbst nicht kopiert, sondern nur die Zeiger bzw. Referenzen auf diese. Die alten, dynamischen Daten des Zielobjekts werden ferner auch nicht freigegeben.

Beispiel:

Die Klasse `Mitarbeiter` dient zur Verwaltung verschiedener Daten eines Angestellten. Dabei sollen auch evtl. Zweitwohnsitze mit ihrer genauen Adresse gespeichert werden können. Da nur wenige Mitarbeiter einen zweiten Wohnsitz besitzen, ist es sinnvoll, diese Daten dynamisch zu speichern, um nur bei vorhandenem zweiten Wohnsitz auch Speicherplatz zu belegen.

kap11_09.cpp (komplettes Beispielprogramm)

```
struct Wohnsitz
{  char PLZ[6];
   char Ort[46];
   char Strasse[50];
   int Nr;
};

class Mitarbeiter
{  char Name[50];
   Wohnsitz HauptWS;
   Wohnsitz *ZweitWS;
   // ...
};
```

Wird nun ein Mitarbeiter auf einen anderen Mitarbeiter kopiert (also zugewiesen), passiert folgendes:

```
Mitarbeiter M1, M2;

M2 = M1; // bei der Zuweisung werden die (statischen) Daten kopiert
```

Hierbei werden die statischen Daten kopiert, d.h. für den Zweitwohnsitz wird nur der Zeiger kopiert, nicht aber die dynamische Datenstruktur selbst. Es zeigen nun die Zeiger beider Mitarbeiter auf den gleichen Speicherbereich. Wird nun der Zweitwohnsitz von dem einen Mitarbeiter verändert, so verändert sich automatisch auch der vom anderen Mitarbeiter. Da der Zeiger vom zweiten Mitarbeiter überschrieben wurde, existiert nun kein Zeiger mehr auf den vorigen Zweitwohnsitz, also auf die dynamischen Daten, auf die er vorher zeigte. Daher können diese auch nicht mehr freigegeben werden (*memory leak*).

Um dieses im allgemeinen unerwünschte Verhalten zu umgehen und trotzdem Objekte mit dem Operator = auf einfache Weise kopieren zu können, ist es notwendig, diesen zu überladen und die notwendigen Schritte zum korrekten Kopieren explizit durchzuführen. Für die Klasse `Mitarbeiter` könnte das so aussehen:

```
class Mitarbeiter
{  // ... (siehe oben)
   public:
      Mitarbeiter &operator= (const Mitarbeiter &m);
};

Mitarbeiter &Mitarbeiter::operator= (const Mitarbeiter &m)
{  strcpy(Name, m.Name);
```

```
   HauptWS = m.HauptWS;
   if (ZweitWS) // ZweitWS != NULL ?
      delete ZweitWS;
   if (m.ZweitWS) // m.ZweitWS != NULL ?
   {  ZweitWS = new Wohnsitz;
      *ZweitWS = *m.ZweitWS;
   }
   else
      ZweitWS = NULL;
   return *this;
}
```

Zuerst (in den ersten beiden Programmzeilen der Operator-Methode) werden die statischen Daten kopiert. Als nächstes wird geprüft, ob im Ziel-Objekt bereits ein Zweitwohnsitz vorhanden ist, wenn ja, wird dieser dynamische Speicherbereich freigegeben. Dann wird geprüft, ob im Quell-Objekt ein Zweitwohnsitz eingetragen ist, wenn ja, wird entsprechend Speicherplatz reserviert und der Zweiwohnsitz kopiert (hier reicht eine einfache Zuweisung, da die Struktur selber keinen Zeiger enthält), wenn nein, wird ein `NULL`-Zeiger zugewiesen.

Mit Hilfe dieses zusätzlichen Operators können nun Mitarbeiter-Objekte korrekt kopiert werden. Es gibt aber immer noch einen Fall, in dem das Kopieren nicht wie zunächst erwartet funktioniert: Beim Aufruf einer Funktion bzw. Methode, die als Parameter ein Mitarbeiter-Objekt als Wert erhält, wird nicht der Zuweisungsoperator verwendet, sondern ein Kopier-Konstruktor (schließlich wird dabei auch ein neues Objekt erzeugt). Und der Standard-Kopier-Konstruktor kopiert wieder nur die statischen Daten. Also muss auch noch ein eigener Kopier-Konstruktor geschrieben werden.

```
class Mitarbeiter
{  // ... (siehe oben)
   public:
      // Kopier-Konstruktor:
      Mitarbeiter (const Mitarbeiter &m);
};

// Standard-Kopier-Konstruktor:
Mitarbeiter::Mitarbeiter (const Mitarbeiter &m)
{  *this = m; // Kopieren der statischen Daten
};

// eigener Kopier-Konstruktor:
Mitarbeiter::Mitarbeiter (const Mitarbeiter &m)
{  // fast gleicher Inhalt wie bei Mitarbeiter::operator=
   strcpy(Name, m.Name);
   HauptWS = m.HauptWS;
   if (m.ZweitWS) // m.ZweitWS != NULL ?
   {  ZweitWS = new Wohnsitz;
      *ZweitWS = *m.ZweitWS;
   }
   else
      ZweitWS = NULL;
}
```

Hinweise:

- Das Kopieren von Objekten kann auch gänzlich unterbunden werden, indem ein Kopier-Konstruktor und ein entsprechender Zuweisungsoperator zwar deklariert, aber nicht definiert werden.
- Wird der Operator = überladen, so ist es nicht unbedingt notwendig, das Argument als Referenzparameter zu schreiben, jedoch meistens empfehlenswert. Beim Kopier-Konstruktor ist es aber Pflicht, das Argument als Referenzparameter zu schreiben. Sonst wäre eine Endlosrekursion die Folge, da bei der Übergabe des Parameters ein neues Objekt erzeugt und damit der Kopier-Konstruktor selber wieder aufgerufen werden würde.

- Wenn die Notwendigkeit des Überladens besteht, sind der Kopier-Konstruktor und der Zuweisungsoperator = in der Regel verschieden zu programmieren. Beim Konstruktor liegt ein uninitialisierter Speicherbereich vor, beim Zuweisungsoperator muss meist vorher geprüft werden, ob nicht ein dynamischer Speicherbereich erst freigegeben werden muss.
- Beim Überladen des Zuweisungsoperators = sollte auch darauf geachtet werden, dass eine Zuweisung der Form `a = a` nicht zum Absturz führt. Im oben angegebenen Beispiel wurde darauf keine Rücksicht genommen, so dass in diesem Fall der Speicherplatz freigegeben wird, der anschließend noch zum Kopieren nötig ist. Um dieses Problem zu umgeben, muss beim Zuweisungsoperator als erstes der Parameter `m` auf Gleichheit mit `this` geprüft werden.
  ```
  if (m == this)
     return *this; // nichts kopieren
  ```
- Leider wird in C++ bei der Initialisierung auch die alte Form noch zugelassen, um mit C kompatibel zu bleiben. Die meisten C++-Compiler führen bei einer Initialisierung der Form "`class complex a(2,3), b(a)`" wie erwartet den Kopierkonstruktor durch, bei einer Initialisierung der Form "`class complex a(2,3), b = a`" allerdings den Zuweisungsoperator. Da in der Zuweisung nicht damit gerechnet werden kann, dass `b` eine noch uninitialisierte Größe ist, führt das oft zu schwerwiegenden Speicherzugriffsfehlern. Initialisieren Sie also immer mit der neuen Schreibweise (Initialwert in Klammern).

11.6. Überladen des Funktionsoperators()

Mit Hilfe des Funktionsoperators kann ein Objekt auch ohne Angabe einer Methode - so als wäre es eine Funktion - aufgerufen werden, z.B. `Objekt(Par1, ...)`. Dadurch kann sich die Schreibarbeit für den Programmierer vereinfachen.

Im Gegensatz zu allen anderen Operatoren ist die Anzahl der Parameter beim Funktionsoperator sowie der Datentyp des Rückgabewertes bei der Definition frei wählbar. Dadurch entstehen hier noch mehr Möglichkeiten des Überladens als bei anderen Operatoren.

Neben dem Funktionsoperator `()` (also mit den runden Klammern) kann auch ein Funktionsoperator mit eckigen Klammern definiert werden (also `[]`).

Beispiel:

kap11_10.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 class FktOp
06 {
07       int Wert;
08    public:
09       FktOp()             { Wert = 0; }
10       FktOp(int W)        { Wert = W; }
11       void Ausgabe()      { cout << Wert << endl; }
12       void operator() () { Ausgabe(); }
13       int operator[] (int W)
14       {
15          int Temp = Wert; // alten Wert merken
16
17          Wert = W;         // neuen Wert setzen
18          Ausgabe();        // neuen Wert ausgeben
19          return Temp;      // alten Wert zurueckgeben
20       }
21 };
```

```
22
23 int main()
24 {
25    FktOp F(5);
26
27    cout << "Aufruf ueber Methode: ";
28    F.Ausgabe();
29    cout << "Aufruf ueber Funktionsoperator(): ";
30    F();
31    cout << "Aufruf ueber Funktionsoperator[]: ";
32    cout << "Ergebnis des Funktionsoperators[]: " << F[7] << endl;
33
34    return 0;
35 }
```

11.7. Überladen des Komponentenzugriffsoperator ->

Der Zugriff auf eine Komponente einer Klasse (oder einer Struktur) über einen Zeiger mit `->` (z.B. `Objekt->Komponente`) ist eine Operation, die aus zwei Teilen besteht. Zuerst wird der linke Teil verarbeitet (`Objekt->`): Hier wird der Zeiger auf die Klasse - oder besser gesagt: auf die Struktur innerhalb der Klasse - ermittelt. Dann wird mit diesem Zeiger auf die Komponente der Struktur zugegriffen.

Wird der Operator `->` überladen, so muss dieser Operator als Ergebnis den ersten Teil - also einen Zeiger auf die Struktur - zurückgeben. Wird also die normale Schreibweise `Objekt->Komponente` in die Operatorschreibweise umgesetzt, sieht es wie folgt aus: `(Objekt.operator->())->Komponente`. Wichtig: Das Ergebnis des selbstdefinierten `->` Operators muss als linke Seite des gesamten Operators verwendbar sein!

Durch das Überladen des `->` Operators erhält man die Möglichkeit, vor dem Zugriff auf die Komponente noch einzugreifen. Beispielsweise könnten damit "intelligente Zeiger" programmiert werden, die vor dem Komponentenzugriff überprüfen, ob dieser überhaupt sinnvoll oder möglich ist. Außerdem kann im Bedarfsfall eine neue linke Seite durch den Funktionswert angegeben werden.

Beispiel:

kap11_11.cpp

```
01 #include <iostream>
02 #include <string>
03
04 using namespace std;
05
06 class IntelligenterZeiger
07 {
08       // die eigentliche Struktur (private)
09       struct Struktur
10       {
11          int Nummer;
12          char Bezeichnung[100];
13          // Konstruktoren fuer die Struktur anlegen:
14          Struktur()
15          {
16             Nummer = 0;
17             strcpy(Bezeichnung, "Nicht vorhanden!");
18          }
19          Struktur(int Nr, char *Bez)
20          {
21             Nummer = Nr;
```

```
22              strcpy(Bezeichnung, Bez);
23           }
24        };
25
26        Struktur *Element;
27        bool Vorhanden;
28
29     public:
30        IntelligenterZeiger(): Vorhanden(false) {}
31        IntelligenterZeiger(int Nr, char *Bez)
32        {
33           Vorhanden = true;
34           Element = new Struktur(Nr, Bez);
35        }
36        ~IntelligenterZeiger()
37        {
38           if (Vorhanden)
39              delete Element;
40        }
41
42        // eigener Komponentenzugriffsoperator:
43        Struktur * operator->()
44        {
45           if (!Vorhanden)
46           {
47              // wenn nicht vorhanden, dann leere Struktur anlegen:
48              Element = new Struktur;
49              Vorhanden = true;
50           }
51           // private Struktur fuer Komponentenzugriff zurueckgeben:
52           return Element;
53        }
54  };
55
56  int main()
57  {
58     IntelligenterZeiger a(5, "Objekt"), b;
59
60     cout << "Nummer: " << a->Nummer << "; "
61          << "Bezeichnung: " << a->Bezeichnung << endl;
62     cout << "Nummer: " << b->Nummer << "; "
63           << "Bezeichnung: " << b->Bezeichnung << endl;
64
65     return 0;
66  }
```

Das Programm erzeugt folgende Ausgabe:

```
Nummer: 5; Bezeichnung: Objekt
Nummer: 0; Bezeichnung: Nicht vorhanden!
```

Hinweis:

Der Operator "`.`" kann nicht überladen werden!

11.8. Überladen des new und delete

Auch die Operatoren `new` und `delete` können überladen werden. Allerdings sind dabei einige besondere Regeln zu beachten:

- Der erste Parameter von `operator new` muss immer den Typ `size_t` haben. Dieser Typ ist je nach Compiler in `stddef.h` oder `types.h` definiert. Dieser Parameter wird immer mit der Größe des bei `new` angegebenen Typs belegt.
- Der Operator `new` kann auch weitere Parameter haben und damit beliebig überladen werden. Die Parameter (außer dem ersten, der ja automatisch belegt wird) werden dann folgendermaßen übergeben: `new (<Par2> P2, ...) <classname>`
- Da bei Aufruf von `operator new` noch kein Objekt existieren kann, ist `new` immer automatisch eine statische Funktion. `new` kennt also kein aktuelles Objekt.
- Hat eine Klasse einen Operator `new`, so wird dieser verwendet, wenn ein Objekt der Klasse dynamisch erzeugt wird. Ansonsten wird der globale Operator `new` verwendet. Wird allerdings ein Vektor dynamisch erzeugt, so wird immer der globale Operator `new` verwendet!
- Der Rückgabewert der Methode `operator new` muss immer `void *` sein.
- Wenn `new` überladen wird, kommt im Fehlerfall in der Regel der `newhandler` nicht zum Einsatz.
- Wird der Operator `delete` in einer Klasse überladen, so muss die Methode immer den Rückgabetyp `void` haben.
- Der Operator `delete` hat immer einen Parameter vom Typ `void *`. Er kann auch einen zweiten Parameter vom Typ `size_t` haben, der, falls vorhanden, mit der Größe des Objekts in Byte belegt ist.
- Pro Klasse kann es zwar mehrere `new`-Operatormethoden, aber immer nur eine `delete`-Operatormethode geben.
- Soll in einer Klasse, die einen überladenen Operator `new` oder `delete` besitzt, auf den globalen Operator zugegriffen werden, so muss der Bereichsoperator `::` verwendet werden (also `::new` bzw. `::delete`).

Mit Hilfe des Überladens der Operatoren `new` und `delete` kann beispielsweise für eine eigene Klasse eine selbstgeschriebene dynamische Speicherverwaltung implementiert werden.

Beispiel:

kap11_12.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 #define MAXANZAHL 100
06
07 class Klasse;
08
09 int AnzObjekte = 0;
10 Klasse *Objektliste[MAXANZAHL];
11
12 class Klasse
13 {
14       int Wert;
15    public:
16       Klasse(int W = 0): Wert(W) {}
17
18       static void *operator new (size_t Groesse)
19       {
20          Klasse *Zeiger = NULL;
21
22          if (AnzObjekte < MAXANZAHL)
23          {
```

```
24              Zeiger = ::new Klasse;
25              if (Zeiger != NULL)
26              {
27                 Objektliste[AnzObjekte] = Zeiger;
28                 AnzObjekte++;
29                 cout << "Objekt dynamisch erzeugt (Adresse "
30                      << Zeiger << ")\n" << endl;
31              }
32           }
33
34           return (void *) Zeiger;
35        }
36
37        static void operator delete (void *Zeiger)
38        {
39           int i = 0;
40
41           for ( ; i < AnzObjekte; i++)
42              if (Objektliste[i] == (Klasse *) Zeiger)
43              {
44                 ::delete (Klasse *) Zeiger;
45                 cout << i + 1 << ". Objekt wurde vernichtet (Adresse "
46                      << Objektliste[i] << ")\n" << endl;
47                 for ( ; i < AnzObjekte - 1; i++)
48                    Objektliste[i] = Objektliste[i + 1];
49                 Objektliste[i] = NULL;
50                 AnzObjekte--;
51              }
52        }
53
54        static void printObjektliste()
55        {
56           if (AnzObjekte == 0)
57              cout << "Keine Objekte dynamisch erzeugt!\n" << endl;
58           else
59           {
60              cout << "Dynamisch erzeugte Objekte:" << endl;
61              for (int i = 0; i < AnzObjekte; i++)
62                 cout << i + 1 << ".: Adresse " << Objektliste[i]
63                      << " (Wert = " << (Objektliste[i])->getWert() << ")"
64                      << endl;
65              cout << endl;
66           }
67        }
68
69        int getWert() { return Wert; }
70        void setWert(int W) { Wert = W; }
71  };
72
73  int main()
74  {
75     Klasse Objekt1(5);   // statisch erzeugtes Objekt!
76     Klasse::printObjektliste();
77
78     Klasse *Objekt2 = (Klasse *) new Klasse;
79     Klasse::printObjektliste();
80
81     Klasse *Objekt3 = (Klasse *) new Klasse(17);
```

```
82    Klasse::printObjektliste();
83
84    Klasse *Objekt4 = (Klasse *) new Klasse(3);
85    Klasse::printObjektliste();
86
87    delete Objekt2;
88    Klasse::printObjektliste();
89
90    delete Objekt4;
91    Klasse::printObjektliste();
92
93    delete Objekt3;
94    Klasse::printObjektliste();
95
96    delete &Objekt1; // Falsch, da Objekt1 nicht dynamisch erzeugt!
97                     // Ist aber kein Compiler-, Linker- o. Laufzeitfehler!
98    return 0;
99 }
```

Besonders problematisch wird es, wenn diese Klasse abgeleitet wird und der `new`-Operator in der abgeleiteten Klasse aufgerufen wird. Denn jetzt wird nur ein dynamisches Objekt der Basisklasse erzeugt. Daher ist es wichtig, dass beim Ableiten von Klassen mit einem eigenen `new`-Operator dieser in den abgeleiteten Klassen überschrieben wird.

Leider funktioniert das Überladen von `new` und `delete` nicht bei allen C++-Compilern gleich. Oft werden viele weitere Ausnahmeregeln definiert. So ist es beispielsweise manchmal möglich, auch die globalen Operatoren `new` und `delete` zu überladen, wie das folgende Beispiel zeigt.

Beispiel:

kap11_13.cpp

```
001 #include <iostream>
002
003 using namespace std;
004
005 #define MAXSPEICHER 100
006
007 typedef struct
008 {
009    void *Zeiger;
010    long Groesse;
011 } Eintrag;
012
013 int AnzReserviert = 0;
014 Eintrag Reserviert[MAXSPEICHER];
015
016 void printReservierteBereiche()
017 {
018    if (AnzReserviert == 0)
019       cout << "Kein Speicher reserviert!\n" << endl;
020    else
021    {
022       cout << "Reservierte Speicherbereiche:" << endl;
023       for (int i = 0 ; i < AnzReserviert; i++)
024          cout << "Adresse " << Reserviert[i].Zeiger
025               << " (" << Reserviert[i].Groesse << " Bytes)" << endl;
026       cout << endl;
027    }
028 }
```

```
029
030 void *operator new (size_t Groesse)
031 {
032    void *Zeiger = NULL;
033
034    if (AnzReserviert < MAXSPEICHER)
035    {
036       Zeiger = malloc(Groesse);
037       if (Zeiger)
038       {
039          Reserviert[AnzReserviert].Zeiger = Zeiger;
040          Reserviert[AnzReserviert].Groesse = long (Groesse);
041          AnzReserviert++;
042          cout << Groesse << " Bytes an Adresse "
043               << Zeiger << " reserviert\n" << endl;
044       }
045    }
046
047    return Zeiger;
048 }
049
050 void *operator new (size_t Groesse, int Anzahl)
051 {
052    void *Zeiger = NULL;
053
054    if (AnzReserviert < MAXSPEICHER)
055    {
056       Zeiger = malloc(Anzahl * Groesse);
057       if (Zeiger)
058       {
059          Reserviert[AnzReserviert].Zeiger = Zeiger;
060          Reserviert[AnzReserviert].Groesse = long (Anzahl * Groesse);
061          AnzReserviert++;
062          cout << Anzahl << " * " << Groesse << " Bytes an Adresse "
063               << Zeiger << " reserviert\n" << endl;
064       }
065    }
066
067    return Zeiger;
068 }
069
070 void operator delete (void *Zeiger)
071 {
072    int i = 0;
073
074    for ( ; i < AnzReserviert; i++)
075       if (Reserviert[i].Zeiger == Zeiger)
076       {
077          free(Zeiger);
078          cout << Reserviert[i].Groesse
079               << " Bytes wieder freigegeben\n" << endl;
080          for ( ; i < AnzReserviert - 1; i++) // Eintraege aufrutschen
081          {
082             Reserviert[i].Zeiger = Reserviert[i + 1].Zeiger;
083             Reserviert[i].Groesse = Reserviert[i + 1].Groesse;
084          }
085          Reserviert[i].Zeiger = NULL;
086          Reserviert[i].Groesse = 0;
```

```
087           AnzReserviert--;
088        }
089 }
090
091 int main()
092 {
093    printReservierteBereiche();
094
095    int *Zahlen = (int *) new(20) int;     // 20 * sizeof(int) Bytes
096    printReservierteBereiche();
097
098    char *Text = (char *) new(30) char;    // 30 * sizeof(char) Bytes
099    printReservierteBereiche();
100
101    double *Zahl = (double *) new double; // sizeof(double) Bytes
102    printReservierteBereiche();
103
104    delete Zahlen;
105    printReservierteBereiche();
106
107    delete Zahl;
108    printReservierteBereiche();
109
110    delete Text;
111    printReservierteBereiche();
112
113    return 0;
114 }
```

12. Templates

Oft tritt das Problem auf, dass der gleiche Algorithmus bzw. die gleiche Funktionalität einer Klasse für verschiedene Datentypen benötigt wird. Ohne Templates kann ein derart allgemeiner Algorithmus, der auf verschiedene Datentypen angewendet werden kann, nur unter Zuhilfenahme von Makros "programmiert" werden. Da ein Makro keine Angaben über den Typ des Elements besitzt, kann der Compiler auch keine Typ-Überprüfung durchführen, was zu einer höheren Fehleranfälligkeit des Programms führt. Die Implementation eines Algorithmus durch Makros, der sich nur in den verwendeten Datentypen unterscheidet, ist bei Erweiterungen schlechter wartbar und dadurch fehleranfälliger.

Mit dem Konzept der **Templates** (auch *Schablonen*, *generische* oder *parametrisierte Typen* genannt) ist es jedoch möglich, einen Algorithmus mit der entsprechenden Typüberprüfung durch den Compiler für verschiedene Datentypen zu entwickeln. Templates werden allerdings nicht von allen Compilern unterstützt bzw. sind in älteren Compilern nicht implementiert, da Templates noch nicht von Anfang an Bestandteil der Sprache C++ waren.

In C++ werden **Funktionen-Templates** und **Klassen-Templates** zur Verfügung gestellt. Dabei beschreibt ein Template eine nicht beschränkte Anzahl von Funktionen bzw. Klassen, in denen Typen verwendet werden, die im Template nicht genauer beschrieben werden. Bei der Verwendung der Templates werden diese fehlenden Typen schließlich bestimmt, und der Compiler generiert dann die nötigen konkreten Funktionen bzw. Klassen. Das heißt, die konkreten Funktionen und Klassen werden erst bei Verwendung eines Templates für jeden eingesetzten Datentyp generiert.

Jede Template-Vereinbarung beginnt dabei folgendermaßen:

```
template <class Par1, class Par2, ... > ...
```

In der Parameterliste stehen die Typen, die durch das Template ersetzt werden sollen, in der Form `class ParX`. Innerhalb der nachfolgend definierten Funktion oder Klasse können diese Typen schließlich verwendet werden. Natürlich müssen für die verwendeten Typen alle im Template vorkommenden Operationen existieren, damit die konkreten Funktionen oder Klassen auch generiert werden können. Findet der Compiler im Quelltext eine Funktion oder Klasse, die mit dem Namen und der Anzahl der Template-Parameter übereinstimmt, wird das Template eingesetzt.

Beispiel:

kap12_01.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 template <class T> T Minimum(T s, T t)
06 {
07    return (s < t) ? s : t;
08 }
09
10 int main()
11 {
12    int i    = Minimum(100, 200);
13    double f = Minimum(3.4, 7.2);
14
15    cout << "i = " << i << endl;
16    cout << "f = " << f << endl;
17
18    return 0;
19 }
```

In diesem Beispiel wird ein Template für eine Funktion `Minimum` erzeugt, die das Minimum zweier Werte eines Datentyps bestimmt. Die Operation < muss für den Datentyp definiert sein. Das `T` in der Template-Definition steht für den später einzusetzenden Datentyp. `class T` steht für einen zu ersetzenden Typ, es

können auch mehrere Ersetzungen vorgenommen werden. Die Funktion `Minimum` liefert einen Wert vom Typ `T` zurück und übernimmt zwei Parameter `s` und `t` vom Typ `T`.

12.1. Funktionen-Templates

Funktionen-Templates können Schablonen für alle Datentypen von C++ sowie auch selbstdefinierten Datentypen darstellen. Die echte Funktion, die auf dem Template beruht, wird erst erzeugt, wenn das Template für einen Datentyp benötigt wird. Wird keine Funktion verwendet, die dem Template entspricht, wird überhaupt nichts getan, also auch kein Programmcode generiert.

Syntax für Funktionen-Templates

Mit dem Schlüsselwort `template` wird die Template-Definition eingeleitet. Dahinter werden in spitzen Klammern dann die **formalen Template-Parameter** angegeben. Diesen wird das Schlüsselwort `class` vorangesetzt, was nicht einer Klasse im bisherigen Sinn entspricht, sondern eher einer Template-Klasse. Es können mehrere Template-Parameter angegeben werden. Diese werden dann später durch den echten Datentyp ersetzt. Es dürfen keine Default-Parameter verwendet werden. Die Parameterliste muss mindestens einen Parameter beinhalten.

Jetzt folgt im Prinzip eine typische Funktionsdeklaration. Zuerst wird der Rückgabetyp der Funktion angegeben. Dabei können auch die formalen Template-Parameter verwendet werden. Nun folgt der Funktionsname und anschließend die Parameterliste. In der Parameterliste müssen mindestens die formalen Template-Parameter vorkommen; zusätzlich können weitere beliebige Parameter angegeben werden.

```
template <class T1, class T2, ...> <Rückgabetyp> FktName(T1 Par1, T2 Par2, ...)
{
   // ...
}
```

Bei den Template-Parametern können außer den formalen Parametern auch noch andere Parameter angegeben werden. Diese müssen dann genauso in der eigentlichen Parameterliste der Funktion vorkommen.

Beispiel:

```
template <class T, int i> int Funktion(T t, int i)
{
   // ...
}
```

Hinweis:

Achten Sie beim Einsatz bestimmten Operatoren (z.B. +, -, *, /, <, >, usw.) darauf, dass diese für den betreffenden Datentyp überhaupt zur Verfügung stehen. Definieren Sie ansonsten die benötigten Operationen mittels Überladen von Operatoren.

Beispiel:

kap12_02.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 template <class T> double Prozent(T Von, T Proz)
06 {
07    // Groesse des Datentyps, der T ersetzt:
08    cout << "\nGroesse von T: " << sizeof(T) << endl;
09    return ((double) Proz * (double) Von / 100.0);
10 }
11
12 int main()
13 {
```

```
14     cout << "32% von 50 = " << Prozent(50, 32) << endl;
15     cout << "32.0% von 50.0 = " << Prozent(50.0, 32.0) << endl;
16     // ASCII 32: Leerzeichen; ASCII 50: '2'
17     cout << "Leerzeichen % von \'2\' = " << Prozent('2', ' ') << endl;
18
19     return 0;
20 }
```

Dieses Beispielprogramm liefert folgende Bildschirmausgabe. Dabei ist anhand der Größe des Datentyps sehr gut zu erkennen, dass tatsächlich 3 verschiedene Funktionen aus dem Template erzeugt wurden.

```
Größe von T: 4
32% von 50 = 16

Größe von T: 8
32.0% von 50.0 = 16

Größe von T: 1
Leerzeichen % von '2' = 16
```

Explizite Spezialisierung

Im letzten Beispiel wurde der formale Template-Parameter für zwei Parameter verwendet. Dieses funktioniert nur, solange beim Funktionsaufruf für beide Parameter immer der gleiche Datentyp eingesetzt wird. Probleme gibt es erst, wenn beim Funktionsaufruf für diese Parameter unterschiedliche Datentypen eingesetzt werden, z.B.

```
cout << Prozent(50.0, 32); << endl;
```

Der Compiler versucht, diesen Funktionsaufruf aufzulösen. Dabei kann er nur auf die bisher deklarierten Funktionen zugreifen. Da aber beim Template beide Parameter den gleichen Typen haben müssen (entweder zwei `int` oder `double`, aber nicht gemischt!) und für Template-Funktionen **keine** automatische Typumwandlung durchgeführt wird, erzeugt diese Zeile einen Fehler beim Compilieren.

Durch die explizite Angabe des gewünschten Datentyps in spitzen Klammern beim Aufruf der Template-Funktion wird eine explizite Spezialisierung des Templates vorgenommen. D.h. es wird ausdrücklich der Datentyp angegeben, auf den die Template-Funktion angewendet werden soll. Damit ist der Compiler dann gezwungen, die Parameter implizit in den angegebenen Datentyp umzuwandeln.

Beispiel:

kap12_03.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 template <class T> double Prozent(T Von, T Proz)
06 {
07     // Groesse des Datentyps, der T ersetzt:
08     cout << "\nGroesse von T: " << sizeof(T) << endl;
09
10     return ((double) Proz * (double) Von / 100.0);
11 }
12
13 int main()
14 {
15     cout << "32% von 50 = " << Prozent(50, 32) << endl;
16     cout << "32.0% von 50.0 = " << Prozent(50.0, 32.0) << endl;
17     // ASCII 32: Leerzeichen; ASCII 50: '2'
18     cout << "Leerzeichen von \'2\' = " << Prozent('2', ' ') << endl;
19
```

```
20     // nun funktioniert auch die folgende Zeile:
21     cout << "32% von 50.0 = " << Prozent<double>(50.0, 32) << endl;
22
23     return 0;
24 }
```

Dieses Programm liefert nun folgende Bildschirmausgabe:

```
Größe von T: 4
32% von 50 = 16

Größe von T: 8
32.0% von 50.0 = 16

Größe von T: 1
Leerzeichen von '2' = 16

Größe von T: 8
32% von 50.0 = 16
```

Bei den letzten beiden Ausgabezeilen wird deutlich, dass hier die Template-Funktion für den Datentyp `double` verwendet und der `int`-Parameter implizit in ein `double` umgewandelt wurde.

12.2. Klassen-Templates

Ein Klassen-Template ermöglicht die Definition einer generischen Klasse. Dabei wird eine Reihe von Klassen beschrieben, in der noch die Typen fehlen. Bei der Verwendung müssen diese Typen eingesetzt werden, so dass der Compiler daraus konkrete Klassen erzeugen kann. Klassen-Templates wurden bereits z.B. im Abschnitt *Komplexe Zahlen* im Kapitel *Datentypen in C++* verwendet.

Die Definition eines Klassen-Templates beginnt ebenfalls mit dem Schlüsselwort `template` und der Angabe der formalen Template-Parameter. Bei der Definition von Methoden einer Template-Klasse ist ebenfalls die Angabe von `template` notwendig. Außerdem muss hier auch darauf geachtet werden, dass mehrere konkrete Klassen von diesem Template gebildet werden, so dass der Klassenname erst mit der Angabe des Template-Parameters eindeutig wird.

Syntax für Klassen-Templates

Das Schlüsselwort template leitet die Definition einer Template-Klasse ein. In spitzen Klammern werden dann die formalen Template-Parameter angegeben. Diesen wird jeweils das Schlüsselwort class vorangesetzt, was wieder einer Template-Klasse entspricht. Es können mehrere Template-Parameter angegeben werden. Diese werden dann später durch den echten Datentyp ersetzt. Es dürfen auch hier keine Default-Parameter verwendet werden. Die Parameterliste muss mindestens einen Parameter beinhalten. Danach folgt die übliche Klassendeklaration. In dieser können dann die formalen Template-Parameter verwendet werden.

```
// Deklaration der Klasse:
template <class T1, ...> class Klassenname
{
   // ...
};
```

Die Methoden der Klasse, die außerhalb der Klasse definiert werden, benötigen zusätzlich einen Hinweis, dass es sich um eine Methode einer Template-Klasse handelt. Dazu wird der Definition das Schlüsselwort `template` sowie die formale Parameterliste vorangestellt. Dem Klassennamen folgen dann noch wie üblich die Parameter für die Methode. Danach wird die Methode wie bei einer ganz normalen Klasse definiert. Die Angabe der formalen Template-Parameter ist hier notwendig, da diese Methode eventuell auch mit den Template-Parameter arbeiten muss.

```
// Definition der Methoden außerhalb der Klasse:
template <class T1, ...> Klassenname<T1, ...>::Methode()
{
   // ...
}
```

Im folgenden Beispielprogramm wird ein Klassen-Template für eine Klasse definiert, die ein Feld von einem bestimmten Datentyp mit einer übergebenen Größe erstellt. Es können Elemente in das Feld übernommen und einzelne Feldwerte ausgelesen werden.

kap12_04.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 // Template-Klasse mit generischen Datentyp T
06 template <class T> class Feld
07 {
08    // private:
09       int Len,         // Gesamtanzahl der Elemente (Laenge des Feldes)
10           Pos;         // aktuelle Anzahl der gespeicherten Elemente
11       T *Data;         // Zeiger auf generischen Datentyp
12
13    public:
14       Feld(int len);  // Konstruktor, legt das Feld dynamisch an
15       ~Feld();        // Destruktor, dynamisches Feld wieder freigeben
16       void Add(T t);  // fuegt dem Feld ein neues Element hinzu
17       T Get(int pos); // liefert Wert an der Position pos aus dem Feld
18 };
19
20 // ***************************************************************
21 // Konstruktor
22 // Beachten Sie den Aufbau dieser Definition: Zuerst muss angegeben
23 // werden, dass es sich um die Definition einer Template-Methode
24 // handelt, dann erst folgt die uebliche Definition der Methode.
25 // Im Konstruktor wird ein Feld mit der angegebenen Laenge len
26 // dynamisch erzeugt.
27 // ***************************************************************
28 template <class T> Feld<T>::Feld(int len)
29 {
30    Pos = 0;           // aktuelle Anzahl bzw. Position zuruecksetzen
31    Len = len;         // Laenge des Feldes merken
32    Data = new T[len]; // Feld dynamisch erzeugen
33 }
34
35 // ***************************************************************
36 // Destruktor
37 // Hier wird der Speicher, der im Konstruktor dynamisch fuer das
38 // Feld beschafft wurde, wieder freigegeben.
39 // ***************************************************************
40 template <class T> Feld<T>::~Feld()
41 {
42    delete [] Data; // Feld wieder dynamisch freigeben
43 }
44
45 // ***************************************************************
46 // Neues Element t dem Feld hinzufuegen. Es wird nicht ueberprueft,
47 // ob das Feld bereits voll ist.
```

```
48 // ********************************************************************
49 template <class T> void Feld<T>::Add(T t)
50 {
51    *(Data + Pos) = t; // Daten t an der naechsten
52                       // freien Stelle im Feld speichern
53    Pos++;             // Position um eins erhoehen
54 }
55
56 // ********************************************************************
57 // Wert an der Position pos aus dem Feld holen. Auch hier wird nicht
58 // geprueft, ob die angegebene Position pos korrekt ist.
59 // ********************************************************************
60 template <class T> T Feld<T>::Get(int pos)
61 {
62    return *(Data + pos - 1);
63 }
64
65 int main()
66 {
67    Feld<int> IntFeld(10);
68    Feld<double> DoubleFeld(10);
69
70    IntFeld.Add(5);
71    IntFeld.Add(17);
72    IntFeld.Add(39);
73    DoubleFeld.Add(3.1415);
74    DoubleFeld.Add(317.99);
75    DoubleFeld.Add(1024.573);
76
77    cout << "Der Inhalt des 2. Elements von IntFeld ist "
78         << IntFeld.Get(2) << endl;
79    cout << "Der Inhalt des 2. Elements von DoubleFeld ist "
80         << DoubleFeld.Get(2) << endl;
81
82    return 0;
83 }
```

Hinweis:

In neueren Compilern werden gleich vollständige Template-Klassen mitgeliefert. Diese implementieren typische Datenstrukturen wie Listen, Container, Stacks, Vektoren, u.a. Für die Implementation dieser Datenstrukturen wurde ein Standard geschaffen: die STL (Standard Template Library).

12.3. Member-Templates

Member-Templates sind Templates für einzelne Methoden (ähnlich der Funktionen-Templates), die innerhalb einer Klasse deklariert werden. Mit diesen Deklarationen ist es möglich, die Methode für ganz verschiedene Typen aufzurufen. Member-Templates werden in der C++-Standardbibliothek verwendet.

Beispiel:

kap12_05.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 class MeineKlasse
06 {
```

```
07    public:
08       template<class T> void Fkt(T &t) { t.Ausgabe(); }
09 };
10
11 class X
12 {
13       int XWert;
14    public:
15       X(int x): XWert(x) {};
16       void Ausgabe() { cout << "Klasse X, Wert: " << XWert << endl; }
17 };
18
19 class Y
20 {
21       char *YWert;
22    public:
23       Y(char *y): YWert(y) {};
24       void Ausgabe() { cout << "Klasse Y, Wert: " << YWert << endl; }
25 };
26
27 int main()
28 {
29    X einX(10);
30    Y einY("Hallo Welt");
31    MeineKlasse einObjekt;
32
33    einObjekt.Fkt(einX);
34    einObjekt.Fkt(einY);
35
36    return 0;
37 }
```

Dieses Beispielprogramm gibt folgendes auf dem Bildschirm aus:

```
Klasse X, Wert: 10
Klasse Y, Wert: Hallo Welt
```

12.4. Rekursive Templates

Zur Vertiefung des Verständnisses von Templates wird im folgenden gezeigt, wie der Compiler (!) zum Rechnen gebracht wird. Das folgende Beispielprogramm berechnet die Zweierpotenz $2^{11} = 2048$. Dabei wird die Klasse mit dem Schlüsselwort `struct` erzeugt, damit alle Elemente der Klasse standardmäßig `public` sind.

Beispiel:

kap12_06.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 template <int n> struct Zweihoch
06 {
07    enum{ Wert = 2 * Zweihoch<n - 1>::Wert };
08 };
09
10 template <> struct Zweihoch<0>
11 {
```

```
12     enum{ Wert = 1 };
13 };
14
15 int main()
16 {
17     cout << "2 ^ 11 = " << Zweihoch<11>::Wert << endl;
18
19     return 0;
20 }
```

Der Compiler versucht, `Zweihoch<11>::Wert` zu ermitteln. Die Aufzählungsvariable `Wert` hat für jeden Typ der Klasse, der von `n` abhängt, einen anderen Wert. Bei der Ermittlung stellt der Compiler fest, dass `Zweihoch<11>::Wert` dasselbe wie `2 * Zweihoch<10>::Wert` ist. `Zweihoch<10>::Wert` ist wiederum dasselbe wie `2 * Zweihoch<9>::Wert`, usw.

Die Rekursion bricht bei der Ermittlung von `Zweihoch<0>::Wert` ab, weil das Template für diesen Fall spezialisiert und der Wert mit `1` besetzt ist. Der Compiler erzeugt insgesamt 12 Datentypen (0 bis 11), die er zur Auswertung heranzieht. Weil er konstante Ausdrücke zur Compilationszeit kennt und berechnen kann, wird an die Stelle von `Zweihoch<11>::Wert` direkt das Ergebnis `2048` eingetragen, so dass zur Laufzeit des Programms keinerlei Rechnungen mehr nötig sind! Diese Methode zur Berechnung von Zweierpotenzen schlägt damit jede andere, was die Rechenzeit angeht. Dieses Verfahren wird mit gutem Erfolg erweitert auf andere Probleme wie zum Beispiel Berechnung der schnellen Fouriertransformation und Optimierung von Vektoroperationen, stellt aber hohe Anforderungen an die verwendeten Compiler.

13. Fehlerbehandlung

13.1. Einführung

Oft tritt ein Fehler in einer Funktion auf, der innerhalb dieser Funktion nicht behoben werden kann. Der Aufrufer der Funktion muss Kenntnis von dem aufgetretenen Fehler bekommen, damit er den Fehler abfangen oder weiter "nach oben" melden kann. Ein Programmabbruch in jedem Fehlerfall ist benutzerunfreundlich und auch nicht immer nötig. Eine allgemeine Fehlerbehebung ist nicht möglich, da jeder Fall einzelnd zu betrachten ist. Mit den bisher kennengelernten Mitteln lassen sich verschiedene Strategien zur Fehlerbehandlung verwenden.

- Das einfachste ist der sofortige Programmabbruch innerhalb der Funktion, in der der Fehler aufgetreten ist. Wenn keine entsprechende Meldung über die Art des Fehlers ausgegeben wird, ist die Fehlersuche sehr schwierig.
- Ein häufig verwendeter Mechanismus ist die Übergabe eines Parameters an den Aufrufer der Funktion. Dieser Parameter gibt Auskunft über Erfolg oder Misserfolg der Funktion. Der Aufrufer hat den Parameter nach jedem Funktionsaufruf auszuwerten und entsprechend zu reagieren. Der Vorteil liegt in der individuellen Art, wie der Aufrufer den Fehler behandeln kann, so dass eventuell gar kein Programmabbruch nötig ist. Der Nachteil besteht darin, dass der Programmcode durch viele eingestreute Abfragen schwerfällig wirkt und dass die Lesbarkeit dadurch leidet. Ein weiterer Nachteil ist, dass sich mancher Programmierer die Abfragen aus Schreibfaulheit spart in der Hoffnung, dass alles gut gehen wird.
- Es kann im Fehlerfall die globale Variable errno auf einen Fehlerwert gesetzt werden, die dann wie im vorigen Fall vom Aufrufer abgefragt werden muss (oder auch nicht). Globale Variablen sind jedoch grundsätzlich nicht gut geeignet, weil sie die Portabilität von Funktionen beeinträchtigen. Ferner sind die Werte der globalen Variablen beim Zusammenarbeiten mehrerer Programmteile möglicherweise nicht mehr eindeutig.
- Eine Funktion, die einen Fehler feststelt, kann eine andere Funktion zur Fehlermeldung und weiteren Bearbeitung aufrufen, die unter Umständen dann den Programmabbruch herbeiführt. Diese Variante erspart dem Aufrufer die vielen Abfragen nach Rückkehr aus der Funktion. Es ist in Abhängigkeit von der Fehlerart zu überlegen, ob dem Aufrufer die Information des Auftretens eines Fehlers mitgeteilt werden muss. Dies gilt sicher dann, wenn die Funktion die ihr zugedachte Aufgabe nicht erledigen konnte.
- Am einfachsten ist es, den Kopf in den Sand zu stecken, d.h. dem Aufrufer trotz eines Fehlers einen gültigen Wert zurückzuliefern und weiter nichts zu tun. Beispiel: Es kann der Index-Operator [] so programmiert werden, dass er bei einer Indexüberschreitung ohne Meldung immer den Wert an der Indexposition 0 zurückgibt. Diese Methode ist aber besonders tückisch, weil der Fehler sich möglicherweise durch die falschen Daten an einer ganz anderen Stelle bzw. erst sehr viel später im Programm bemerkbar macht. Solche Fehler ist nur sehr schwer zu finden. Weiteres Beispiel: Eingegebene Daten ohne Rückmeldung einfach verwerfen, wenn die Datei voll ist. usw. Diese Variante ist am uneffektivsten, wird aber leider noch an vielen Stellen angewendet.

Die Art der Fehlerbehandlung hängt auch davon ab, wie sicherheitskritisch der Einsatz der Software ist. Eine Fehlermeldung `Index-Fehler in Zeile 517; Index = 8530` ist in einem Textverarbeitungsprogramm vielleicht noch möglich (führt hoffentlich zu einem Fehlerbericht an den Programmierer). Ganz anders ist es aber, wenn diese Fehlermeldung in der Software eines in der Luft befindlichen Flugzeugs auftritt. Es reicht dann nicht aus, die Meldung einfach in eine LOG-Datei zu speichern und mit dem Programm weiterzumachen. Genauso wenig hilfreich ist es, dem Piloten diese Fehlermeldung anzuzeigen. Was soll der Pilot machen? Vielleicht eine aktuelle Firmware während des Fluges herunterladen?

Unter den oben angegebenen Varianten sind die zweite und noch mehr die vierte akzeptabel. C++ stellt zusätzlich die Ausnahmebehandlung zur Verfügung, mit der auftretende Fehler an spezielle Routinen übergeben werden können.

13.2. Ausnahmebehandlung

Die **Ausnahmebehandlung** (englisch *exception handling*) bietet den Vorteil, die Fehlerbehandlung von "normalen" Programmcode sauber zu trennen. Damit werden die Fehlerabfragen jeweils nach der Rückkehr aus einer Funktion überflüssig.

Zu unterscheiden ist zwischen der *Erkennung* von Fehlern (z.B. Division durch Null) und der *Behandlung* von Fehlern. Die Erkennung ist in der Regel einfach, die Behandlung ist dagegen häufig schwierig und manchmal unmöglich, so dass ein Programmabbruch unumgänglich ist. Die Fehlerbehandlung sollte einen von den zwei folgenden Wegen einschlagen:

- Den Fehler beheben und den Programmablauf fortsetzen oder, falls das nicht gelingt, das Programm mit einer aussagekräftigen Meldung abbrechen.
- Den Fehler an die aufrufende Funktion melden, die den Fehler dann ebenfalls auf eine dieser zwei Arten bearbeiten muss.

Für den Fall, dass Fehler in einer Funktion vom Aufrufer behandelt werden können, stellt C++ alternativ die Ausnahmebehandlung zur Verfügung. Der Ablauf läßt sich wie folgt skizzieren:

1. Eine Funktion versucht (englich *try*) die Erledigung einer Aufgabe.
2. Wenn diese Funktion einen Fehler feststellt, den sie nicht beheben kann, wirft (englisch *throw*) sie eine Ausnahme (englisch *exception*) aus.
3. Die Ausnahme wird von einer Fehlerbehandlungsroutine aufgefangen (englisch *catch*), die den Fehler behandelt.

Die Funktion, in der der Fehler auftritt, kann der Fehlerbehandlungsroutine ein Objekt eines beliebigen Datentyps zuwerfen, um Informationen zu übergeben, d.h. durch Zuwerfen unterschiedlicher Datentypen können unterschiedliche Fehlerbehandlungsroutinen aufgerufen werden. Nach der Behandlung des Fehlers wird das Programm mit dem Code, der den Fehlerbehandlungsroutinen folgt, fortgesetzt, d.h. die Funktion, in der der Fehler aufgetreten ist, wird nicht weiter abgearbeitet (oder noch anders ausgedrückt: Jedes `throw` führt zum Verlassen der Funktion). Wird durch den Aufruf einer Fehlerbehandlungsroutine aus einem Block herausgesprungen, werden die Destruktoren aller in diesem Block definierten Objekte aufgerufen.

Im folgenden wird die syntaktische Struktur gezeigt:

```
try
{
   func(); // falls func() einen Fehler entdeckt,
           // wirft sie eine Ausnahme aus
}

catch(Datentyp1)
{
   // Fehlerbehandlung für ausgeworfenes
   // Objekt vom Datentyp1
   // ...
}

catch(Datentyp2)
{
   // Fehlerbehandlung für ausgeworfenes
   // Objekt vom Datentyp2
   // ...
}

// gegebenenfalls weitere catch-Blöcke

// Fortsetzung des Programms nach
```

```
// Fehlerbehandlung an dieser Stelle!
// ...
```

Die Ausnahmebehandlung ist eigentlich zur Fehlerbehandlung gedacht. Davon kann aber auch abgewichen werden. Im folgenden Beispiel wird das Einlesen einer Zahl mit einer Ausnahmebehandlung versehen, so dass fehlerhafte Eingaben ignoriert werden. Dabei wird auch das Erreichen des Dateiendes (durch Drücken der Tastenkombination Strg+D oder Strg+Z) als Ausnahme behandelt, obwohl dies kein Fehler ist.

```
#include <iostream>

using namespace std;

class DateiEnde {}; // Hilfsklasse

int liesZahl(istream& ein)
{
   int i;

   ein >> i;
   if (ein.eof())  // Dateiende erreicht?
      throw DateiEnde();
   if (ein.fail()) // falsche Zeichen eingegeben?
      throw "Syntaxfehler";
   if (ein.bad())  // nicht behebbarer Fehler?
      throw;
   return i;
} // liesZahl()

void Zahlen_lesen_und_ausgeben()
{
   int Zahl;
   bool Erfolgreich = true;

   while (true) // Endlosschleife
   {
      Erfolgreich = true;
      cout << "Zahl eingeben: ";
      try // Versuchsblock
      {
         // Zahl von Standardeingabe lesen
         Zahl = liesZahl(cin);
      }

      // Fehlerbehandlung:
      catch (DateiEnde)
      {
         cout << "Dateiende erreicht!" << endl;
         cin.clear(); // Fehlerbit zurücksetzen
         break;       // while-Schleife verlassen
      }

      catch(const char *Fehlermeldung)
      {
         cerr << Fehlermeldung << endl;
         Erfolgreich = false;
         cin.clear(); // Fehlerbit zurücksetzen
         while (cin.get() != '\n')
            ;         // falsche Zeichen entfernen
      }
```

```
      // Fortsetzung des Programms
      if (Erfolgreich)
         cout << "Zahl = " << Zahl << endl;
   } // while
} // Zahlen_lesen_und_ausgeben()

int main()
{
   Zahlen_lesen_und_ausgeben();
   return 0;
}
```

In der Funktion `liesZahl()` wird nach dem Einlesen der Zahl als erstes geprüft, ob das Dateiende erreicht ist (`eof()`). Wenn ja, wird ein Objekt der Klasse `DateiEnde`, dass durch den Aufruf des systemgenerierten Standard-Konstruktors erzeugt wird, ausgeworfen. Die Verwendung dieser Klasse dient nur der Lesbarkeit; anstelle dessen hätte auch eine beliebige Zahl ausgeworfen werden können, der `catch`-Block hätte dann ein `int` auffangen müssen.

Danach wird geprüft, ob falsche Zeichen eingegeben wurden (`fail()`). Wenn ja, wird eine konstante Zeichenkette ausgeworfen. Im dazugehörigen `catch`-Block müssen diese falschen Zeichen aus dem Tastaturpuffer entfernt werden. Im Beispiel werden alle Zeichen im Tastaturpuffer entfernt, alternativ könnten nur alle Nicht-Ziffern entfernt werden, bis eine Ziffer kommt oder der Tastaturpuffer leer ist.

Im `catch`-Block für Dateiende wird in den Klammern nur der Klassenname aber kein Objektname angegeben. Dies ist hier nicht notwendig, da auf das Objekt nicht zugegriffen wird. Dagegen wird im zweiten `catch`-Block das übergebene Objekt – wie in einer Funktionsdefinition – benannt und kann dadurch auf der Standardfehlerausgabe ausgegeben werden.

Für schwere und nicht behebbare Fehler (`bad()`) ist hier keine Fehlerbehandlungsroutine definiert. Diese Fehler werden deswegen solange an die nächsthöhere Ebene weitergereicht, bis sie auf eine geeignete Fehlerbehandlungsroutine treffen. Ist wie hier keine vorhanden, wird das Programm abgebrochen.

Die Fehlerbehandlungsroutinen werden der Reihe nach abgefragt, so dass eine, die auf alle Fehler passt, am Ende der Liste stehen muss, um die anderen nicht zu verdecken.

Das obige Programm könnte entsprechend erweitert werden. Die Folge von drei Punkten innerhalb der runden Klammern wird **Ellipse** genannt, was soviel wie Auslassung bedeutet. Nichtangabe eines Datentyps durch drei Punkte bedeutet "beliebige Anzahl beliebiger Datentypen".

```
catch (...) // für nicht spezifizierte Fehler
{
   cerr << "nicht behebbarer Fehler!" << endl;
   throw;   // Weitergabe an nächsthöhere Instanz
}
```

Der Vorteil der Trennung von Fehlererkennen und -behandlung wird durch einen Verlust an Lokalität erkauft, weil von der fehlerentdeckenden Stelle an eine ganz andere Stelle gesprungen wird, die auch Anlaufpunkt vieler anderer Stellen sein kann. Es lohnt sich daher, sich vor der Programmierung eine angemessene Strategie zur Fehlerbehandlung zu überlegen.

13.3. Exception-Sezifikationen

Um Ausnahmen, die eine Funktion auswerfen kann, in den Schnittstellen bekannt zu machen, können sie in der Funktionsdeklaration als sogenannt **Exception-Spezifikation** deklariert werden. Dabei sind drei Fälle zu unterscheiden:

1. `int liesZahl(istream &ein);`
   kann beliebige Ausnahmen auswerfen

2. `int liesZahl(istream &ein) throw();`
   verspricht, keine Ausnahmen auswerfen
3. `int liesZahl(istream &ein) throw(DateiEnde, const char*);`
   verspricht, nur die angegebenen Ausnahmen auswerfen

Der Benutzer einer Funktion kennt zwar ihren Prototyp, im allgemeinen aber nicht die Implementierung der Funktion. Die Angabe der möglichen Ausnahmen im Prototyp ist sinnvoll, damit der Benutzer der Funktion sich darauf einstellen und geeignete Maßnahmen treffen kann.

13.4. vordefinierte Exception-Klassen

Im Beispiel im Abschnitt 13.2 ist eine eigene Klasse definiert worden, deren Objekte mit `catch` aufgefangen werden. Eigene Klassen zur Ausnahmebehandlung können sinnvoll sein, C++ stellt aber auch eine Reihe vordefinierter Exception-Klassen zur Verfügung. Dabei erben alle speziellen Exception-Klassen von der Basisklasse `exception`, die folgende öffentliche Schnittstelle hat:

```
namespace std
{
   class exception
   {
      public:
         exception() throw();
         exception(const exception&) throw();
         exception& operator=(const exception&) throw();
         virtual ~exception() throw();
         virtual const char * what() const throw();
      private:
         // ...
   };
}
```

`throw()` nach den Deklarationen bedeutet, dass die Klasse selbst keine Exception wirft, weil es sonst eine unendliche Folge von Exceptions geben könnte.

Die Methode `what` gibt einen Zeiger auf `char` zurück, der auf eine Fehlermeldung verweist. Eigene Exeption-Klassen können durch Vererbung die Schnittstelle übernehmen, so wie die Klasse `logic_error`:

```
namespace std
{
   class logic_error: public exception
   {
      public:
         explicit logic_error(const string& argument);
   };
}
```

Dem Konstruktor kann ein `string`-Objekt mitgegeben werden, das eine Fehlerbeschreibung enthält, die im `catch`-Block ausgewertet werden kann. Mit `string` ist die Stringklasse der Standardbibliothek gemeint. Die von der Klasse `exception` geerbte Methode `what()` gibt einen Zeiger vom Typ `char *` anstelle eines `string`-Objekts zurück, weil die Erzeugung eines Strings dynamisch geschieht und ihrerseits eine Exception hervorrufen könnte. Der von `what()` zurückgegebene Zeiger verweist auf eine statisch abgelegte Fehlermeldung.

Im folgenden sind alle vordefinierten Exception-Klassen mit deren Bedeutung aufgelistet. Außerdem ist die Hierarchie der Klassen und die jeweils benötigte Headerdatei angegeben.

Klasse	`exception`
Unterklasse von	-
Bedeutung	Basisklasse
Headerdatei	`<exception>`
Klasse	`logic_error`
Unterklasse von	`exception`
Bedeutung	theoretisch vermeidbare Fehler, die noch vor der Ausführung des Programms entdeckt werden können, zum Beispiel Verletzung von logischen Vorbedingungen
Headerdatei	`<stdexcept>`
Klasse	`invalid_argument`
Unterklasse von	`logic_error`
Bedeutung	Fehler in Funktionen der Standard-C++-Bibliothek bei ungültigen Argumenten
Headerdatei	`<stdexcept>`
Klasse	`length_error`
Unterklasse von	`logic_error`
Bedeutung	Fehler in Funktionen der Standard-C++-Bibliothek, wenn ein Objekt erzeugt werden soll, das die maximal erlaubte Größe für dieses Objekt überschreitet
Headerdatei	`<stdexcept>`
Klasse	`out_of_range`
Unterklasse von	`logic_error`
Bedeutung	Bereichsüberschreitungsfehler in Funktionen der Standard-C++-Bibliothek
Headerdatei	`<stdexcept>`
Klasse	`domain_error`
Unterklasse von	`logic_error`
Bedeutung	anderer Fehler des Anwendungsbereichs
Headerdatei	`<stdexcept>`

Klasse	`runtime_error`
Unterklasse von	`exception`
Bedeutung	nicht vorhersehbare Fehler, deren Gründe außerhalb des Programms liegen, zum Beispiel datenabhängige Fehler
Headerdatei	`<stdexcept>`
Klasse	`range_error`
Unterklasse von	`runtime_error`
Bedeutung	Bereichsüberschreitung zur Laufzeit
Headerdatei	`<stdexcept>`
Klasse	`overflow_error`
Unterklasse von	`runtime_error`
Bedeutung	arithmetischer Überlauf
Headerdatei	`<stdexcept>`
Klasse	`undverflow_error`
Unterklasse von	`runtime_error`
Bedeutung	arithmetischer Unterlauf
Headerdatei	`<stdexcept>`
Klasse	`bad_alloc`
Unterklasse von	`exception`
Bedeutung	Speicherzuweisungsfehler
Headerdatei	`<new>`
Klasse	`bad_typeid`
Unterklasse von	`exception`
Bedeutung	falscher Objekttyp

Headerdatei	`<typeinfo>`
Klasse	`bad_cast`
Unterklasse von	`exception`
Bedeutung	Typumwandlungsfehler
Headerdatei	`<typeinfo>`
Klasse	`bad_exception`
Unterklasse von	`exception`
Bedeutung	Verletzung der Exception-Spezifikation
Headerdatei	`<exception>`

13.5. Besondere Fehlerbehandlungsfunktionen

Nun kann es passieren, dass während einer Fehlerbehandlung selbst wieder ein Fehler auftritt. Die in diesem Fall aufgerufenen Funktionen werden in diesem Abschnitt erläutert. Leider werden diese Funktionen nur bedingt von den verschiedenen Compilern unterstützt. Daher sollte in jedem Fall in der Dokumentation des Compilers nachgeschlagen werden.

Alle hier beschriebenen Funktionen sind in der Headerdatei `<exception>` deklariert.

- Methode `void terminate();`

  Beschreibung: Die Funktion `terminate()` beendet das Programm und wird (unter anderem) aufgerufen, wenn
  • der Exception-Mechanismus keine Möglichkeit zur Bearbeitung einer geworfenen Exception findet,
  • der vorgegebene `unexpected_handler()` gerufen wird,
  • ein Destruktor während des Aufräumens (englisch *stack unwinding*) eine Exception wirft oder wenn
  • ein statisches (nicht-lokales) Objekt während der Erzeugung oder Zerstörung eine Exception wirft.

- Methode `void unexpected();`

  Beschreibung: Die Funktion `unexpected()` wird aufgerufen, wenn eine Funktion ihre Versprechungen nicht einhält und eine Exception wird, die in der Exception-Spezifikation nicht aufgeführt wird. `unexpected()` kann nun selbst eine Exception auslösen, die der verletzten Exception-Spezifikation genügt, so dass die Suche nach dem geeigneten Exception-Handler weitergeht. Falls die ausgelöste Exception nicht der Exception-Spezifikation entspricht, sind zwei Fälle zu unterscheiden:
  1. Die Exception-Spezifikation führt `bad_exception` auf: Die geworfene Exception wird durch ein `bad-exception`-Objekt ersetzt.
  2. Die Exception-Spezifikation führt `bad_exception` nicht auf: Es wird die Funktion `terminate()` aufgerufen.

- Methode `bool uncaught_exception();`

  Beschreibung: Die Funktion `uncaught_exception()` gibt `true` nach der Auswertung einer Exception zurück, bis die Initialisierung im Exception-Handler abgeschlossen ist oder `unexpected()` wegen der Exception aufgerufen wurde. Ferner wird `true` zurückgegeben, wenn `terminate()` aus irgendeinem Grund außer dem direkten Aufruf begonnen wurde.

13.6. Benutzerdefinierte Fehlerbehandlungsfunktionen

Die im vorigen Abschnitt erläuterten Funktionen können bei Bedarf selbst definiert werden, um die vorgegebenen zu ersetzen. Dazu sind standardmäßig zwei Typen für Funktionszeiger definiert:

```
typedef void (*unexpected_handler)();
typedef void (*terminate_handler)();
```

Dazu passend gibt es zwei Funktionen, denen die Zeiger auf die selbstdefinierten Funktionen dieses Typs übergeben werden, um die vorgegebenen zu ersetzen:

```
unexpected_handler set_unexpected(unexpected_handler f) throw();
terminate_handler set_terminate(terminate_handler f) throw();
```

Übergeben werden Zeiger auf selbstdefinierte Funktionen, an die bestimmte Anforderungen gestellt werden:

Ein `unexpected_handler` soll

- `bad_exception` oder eine der Exception-Spezifikation genügende Exception werden oder
- `terminate()` aufrufen oder
- das Programm mit `abort()` oder `exit()` beenden. Der Unterschied zwischen den beiden liegt darin, dass `exit()` vor Programmende noch die mit `atexit(void (*f)())` registrierten Funktionen sowie alle Destruktoren statischer Objekte aufruft. Automatisch erzeugte (also Stack-) Objekte werden nicht zerstört.

Ein `terminate_handler` soll das Programm ohne Rückkehr an den Aufrufer beenden.

13.7. Zusicherungen

Wie können logische Fehler in einer Funktion oder allgemein in einem Programm gefunden werden, insbesondere während der Entwicklung? Dies ist möglich, indem in das Programm eine **Zusicherung** einer logischen Bedingung (englisch *assertion*) eingebaut wird. Das Makro `assert()` (siehe ANSI-C-Headerdatei `assert.h`) dient diesem Zweck. Das Argument des Makros nimmt eine logische Annahme auf. Ist die Annahme wahr, passiert nichts, ist sie falsch, wird das Programm mit einer Fehlermeldung abgebrochen. Sämtliche logischen Überprüfungen mit `assert`-Makros werden auf einmal abgeschaltet, wenn die Compilerdirektive `#define NDEBUG` vor dem `assert()`-Makro definiert wurde. Damit entfallen aufwendige Arbeitsvorgänge mit dem Editor für auszuliefernde Software, die keinen Debug-Code mehr enthalten soll.

Der Nachteil dieses Makros besteht in dem erzwungenen Programmabbruch. Wenn an die Stelle eines Abbruchs eine besondere Fehlerbehandlung (mit oder ohne Programmabbruch) treten soll, kann einfach ein eigenes Makro unter Benutzung der Ausnahmebehandlung geschrieben werden:

```
#ifndef myassert_h
   #define myassert_h myassert_h

   #ifdef NDEBUG
      // MyAssert durch nichts ersetzen
      #define MyAssert(Bedingung, Ausnahme)
   #else
```

```
      #define MyAssert(Bedingung, Ausnahme) \
         { if (!(Bedingung)) throw Ausnahme; }
   #endif
#endif
```

Das folgende Programm zeigt beispielhaft, wie dieses Makro benutzt wird. Wenn eine `1` eingegeben wird, endet das Programm normal. Wenn eine `0` eingegeben wird, gibt es zusätzlich eine Fehlermeldung. In allen anderen Fällen wird das Programm mit einer Fehlermeldung abgebrochen. Es wird gezeigt, wie Exception-Objekten Informationen mitgegeben werden können, die innerhalb des Objekts ausgewertet werden. In diesem Fall ist es nur die Entscheidung, ob die übergebene Zahl gleich 0 oder 1 ist.

```
#include <iostream>
#include <cstdlib>
#include <exception>
// Abschalten der Zusicherung mit NDEBUG
// #define NDEBUG
#include "myassert.h"

using namespace std;

// Exception-Klassen
class GleichNull
{
   public:
      const char* what() const
      {
         return "GleichNull entdeckt!";
      }
};

class UngleichEins
{
   public:
      UngleichEins(int i): Zahl(i) {}
      const char* what() const
      {
         return "UngleichEins entdeckt!";
      }
      int Wieviel() const { return Zahl; }
   private:
      int Zahl;
};

int main()
{
   int i;

   cout << "0 - GleichNull-Fehler\n"
        << "1 - normales Ende\n"
        << "!=1 - UngleichEins-Fehler\n\n"
        << "i = ";
   cin >> i;

   try
   {
      MyAssert(i, GleichNull());
      MyAssert(i == 1, UngleichEins());
   }
```

```
   catch(const GleichNull& FehlerObjekt)
   {
      cerr << FehlerObjekt.what() << endl
           << "keine weitere Fehlerbehandlung\n";
   }

   catch(const UngleichEins& FehlerObjekt)
   {
      cerr << FehlerObjekt.what() << endl
           << FehlerObjekt.Wieviel()
           << "!   ABBRUCH!" << endl;
      exit(1);
   }

   cout << "normales Programmende mit i = "
        << i << endl;
}
```

13.8. Speicherfehler durch Exceptions

In C++ kann ein Problem auftreten, wenn Speicher an einen in einem Block definierten Zeiger dynamisch zugewiesen wird und innerhalb dieses Blocks eine Exception ausgeworfen wird. Im folgenden Beispiel ist `Klasse` eine beliebige Klasse:

```
void func()
{
   Klasse Objekt; // lokales (Stack-)Objekt
   Klasse *Zeiger_auf_Objekt = new Klasse;
                  // dynamisches (Heap-)Objekt

   f(Objekt);     // irgendeine Berechnung
   f(*Zeiger_auf_Objekt);
   delete Zeiger_auf_Objekt;  // dynamisches Objekt freigeben
}
```

Wenn die Funktion `f()` eine Ausnahme auswirft, wird nur der Destruktor vom Objekt `Objekt` aufgerufen und damit auch nur von diesem Objekt der Speicher (auf dem Stack) wieder freigegeben. Der Speicher des Objektes, auf das der Zeiger `Zeiger_auf_Objekt` zeigt, wird dagegen nicht freigegeben, weil durch das Auswerfen der Exception die Funktion `func()` sofort verlassen wird und dadurch die Zeile mit dem `delete` nicht erreicht wird. Außerhalb der Funktion `func()` ist aber der Zeiger `Zeiger_auf_Objekt` nicht mehr bekannt.

```
int main()
{
   try
   {
      func();
   }

   catch(...)
   {
      delete Zeiger_auf_Objekt; // Fehler! Zeiger ist hier nicht bekannt!
   }
}
```

Aus diesem Grund sollten ausschließlich automatische (Stack-)Objekte verwendet werden, wenn Exceptions auftreten. Dies ist immer möglich, wenn man Beschaffung und Freigabe eines dynamischen Objektes

innerhalb eines Stack-Objekts versteckt. Eine Möglichkeit sind die "Intelligenten Zeiger" (siehe Kapitel 11.7 – erweitert als Template):

```
void func()
{
   Klasse Objekt;
   IntelligenterZeiger Zeiger_auf_Objekt(new Klasse);

   f(Objekt);
   f(*Zeiger_auf_Objekt);
}
```

Nun sind sowohl `Objekt` als auch `Zeiger_auf_Objekt` automatische Objekte. Das Problem beim Auswerfen von Exceptions ist behoben, weil der Destruktor des "Intelligenten Zeiger"-Objekts den beschafften Speicher wieder freigibt.

14. Iteratoren

14.1. Einführung

Bei Arrays kann auf die einzelnen Elemente der Reihe nach zugegriffen werden. Auch mit Zeigern kann mittels der Zeigerarithmetik auf hintereinander liegende Daten zugegriffen werden. In beiden Fällen funktioniert es nur dadurch, dass alle Elemente hintereinander im Arbeitsspeicher liegen und dass alle Elemente die gleiche Größe haben, d.h. es muss jeweils die gleiche Anzahl von Bytes weiter gesprungen werden, um zum nächsten Element zu gelangen.

Nun gibt es auch Klassen mit Datenstrukturen, bei denen die Daten im Arbeitsspeicher nicht hintereinander liegen (z.B. verkettete Listen) oder bei denen jedes Element eine andere Größe hat (z.B. ein Vektor von unterschiedlich langen Strings). In solchen Fällen werden für diese Klassen Methoden benötigt, um trotzdem die einzelnen Daten sequentiell aufrufen zu können. Solche Methoden verbergen gleichzeitig die innere Struktur der Daten (**Kontrollabstraktion**). Ein Prinzip zur Kontrollabstraktion dieser Art wird **Iterator** genannt.

Ein Iterator ist kein Typ und kein Objekt, sondern ein Konzept, eine Bezeichnung, die auf eine Menge von Klassen und Typen zutrifft, die bestimmten Anforderungen entsprechen. Ein Iterator kann auf ganz unterschiedlichen Weisen durch die Grunddatentypen oder Klassenobjekte realisiert werden. Weil ein Iterator zum Zugriff auf Elemente von Klassenobjekten dient, ist er eine Art verallgemeinerter Zeiger.

Beispiel:

```
#include <iostream>
#include <vector>

using namespace std;

typedef int * MeinIterator;
typedef void (*Funktion)(int &);
   // Der neue Typ Funktion ist ein Zeiger auf eine Funktion,
   // die eine Referenz auf int erhält und ein void zurückgibt.

void Drucken(int &x)
{
   cout.width(3);
   cout << x;
}

void anwenden(MeinIterator Anfang, MeinIterator Ende, Funktion f)
{
   while (Anfang != Ende)
      f(*Anfang++);
}

int main()
{
   int Zahlen[10];

   // Zahlenarray mit Werten füllen:
   for (int i = 0; i < 10; i++)
      Zahlen[i] = i * i;

   MeinIterator von = Zahlen, bis = von + 10;
   anwenden(von, bis, Drucken);
```

```
   cout << endl;
}
```

In diesem Beispiel ist `MeinIterator` vom Typ `int *`, also ein einfacher Zeiger. Der Iterator `von` zeigt auf den Anfang des Zahlen-Arrays und der Iterator `bis` zeigt auf die Position direkt nach dem letzten Array-Element. Die Funktion `anwenden()` bekommt die beiden Iteratoren sowie einen Zeiger auf eine Funktion als Argumente und ruft die Funktion für jedes Array-Element auf, wobei der Iterator `Anfang` bei jedem Schritt um eine Position weitergesetzt wird.

Die Funktion `anwenden()` wird nun im nächsten Beispiel zusätzlich mit einem `vector`-Objekt aufgerufen.

Beispiel:

```
#include <iostream>
#include <vector>

using namespace std;

typedef int * MeinIterator;
typedef void (*Funktion)(int &);
   //Der neue Typ Funktion ist ein Zeiger auf eine Funktion,
   // die eine Referenz auf int enthält und ein void zurückgibt.

void Drucken(int &x)
{
   cout.width(3);
   cout << x;
}

void anwenden(MeinIterator Anfang, MeinIterator Ende, Funktion f)
{
   while (Anfang != Ende)
      f(*Anfang++);
}

int main()
{
   int Zahlen[10];

   // Zahlenarray mit Werten füllen:
   for (int i = 0; i < 10; i++)
      Zahlen[i] = i * i;

   MeinIterator von = Zahlen, bis = von + 10;
   anwenden(von, bis, Drucken);

   vector <int> vec(8, 1); // 8 int-Elemente mit dem Wert 1
   anwenden(vec.begin(), vec.end(), Drucken);

   cout << endl;
}
```

Die Methoden `begin()` und `end()` liefern Iteratoren (in diesem Fall wieder Zeiger auf `int`) auf das erste Vektor-Element bzw. auf die erste Position nach dem Vektor `vec` (analog zu `von` und `bis`). Diese beiden sowie weitere Methoden stehen für alle Container-Klassen, die ein Iterator-Objekt zurückgeben, zur Verfügung (z.B. `vector`, `list`, `queue`, `deque`, `map`, `set`, `stack`).

Die Funktion `anwenden()` aus dem obigen Beispiel lässt sich als Template-Funktion verallgemeinern und ist im allgemeinen in folgender Fassung in der Headerdatei `<algorithm>` (gehört zur STL – Standard Template Library) enthalten:

```
Template <class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f)
{
   for (; first != last; ++first)
      f(*first);
   return f;
}
```

Durch die Wahl der Iteratoren und der Funktion ist es möglich, beliebige Container zu durchlaufen, ohne dass die Funktion `for_each()` jedesmal neu geschrieben werden muss.

Aus der Template-Funktion for_each() ist ersichtlich, welche Operatoren für die Iteratoren existieren müssen:

- Derefenzierungsoperator `*`
- Vergleichsoperator `!=`
- Inkrementoperator `++`

Um flexibel zu sein, sollte der Inkrementoperator in der Prä- und Postfix-Variante vorhanden sein. Zusätzlich sollten auch der Dekrementoperator `--` (auch Prä- und Postfix-Variante), die Subtraktionsoperatoren `-` und `-=` sowie der Vergleichsoperator `==` definiert sein, um eine möglichst gute Analogie zu der Zeigerarithmetik zu haben.

14.2. Iterator-Kategorien

Entsprechend der Funktionalität gibt es unterschiedliche **Iterator-Kategorien** in einer hierarchischen Anordnung.

Input-Iteratoren

Input-Iteratoren stellen das universelle Konzept zum lesenden Zugriff auf die Datenelemente dar. Hierbei ist **nicht** gefordert, das bei mehrmaligen Durchlaufen der Daten jedes Mal die gleichen Elemente geliefert werden – die Daten können damit auch vom Datenstrom `cin` kommen. Ein Zurückspringen an eine schon gelesene Stelle ist nicht möglich, da der Dekrementoperator `--` nicht definiert sein muss.

Output-Iteratoren

Mit Output-Iteratoren können Datenelemente geändert bzw. neue Elemente erzeugt werden. Dazu wird der Dereferenzierungsoperator `*` verwendet.

Forward-Iteratoren

Forward-Iteratoren haben die Funktionalität von Input- **und** Output-Iteratoren. Im Unterschied zu den vorigen Iteratoren können Werte des Forward-Iterators gespeichert werden, um ein Element der Datenstruktur wiederzufinden. Damit ist ein mehrfacher Durchlauf in Vorwärts-Richtung möglich (z.B. durch eine einfach verkettete Liste), wenn man sich den Anfang gemerkt hat.

Bidirectional-Iteratoren

Ein bidirektionaler Iterator ist ein Forward-Iterator, der die Datenelemente auch rückwärts durchlaufen kann (Dekrementoperator `--`). Die Iteratoren aller Standardcontainer sind mindestens bidirektionale Iteratoren.

Random-Access-Iteratoren

Random-Access-Iteratoren sind bidirektionale Iteratoren, welche zusätzlich wahlfreien Zugriff auf die Elemente bieten. Positionen eines Random-Access-Iterators lassen sich mit den Vergleichsoperatoren

vergleichen und durch Addition und Subtraktion von ganzzahligen Werten erhöhen bzw. erniedrigen (auch über den Speicherbereich der Daten hinaus – analog zu den Arrays). Ferner können zwei Positionen voneinander abgezogen werden, um die Anzahl der Elemente zu erhalten, die zwischen den beiden Positionen liegen. Die Standardcontainer `vector<T>` und `deque<T>` verfügen über Random-Access-Iteratoren.

In der folgenden Tabelle werden die Fähigkeiten der Iterator-Kategorien noch einmal zusammengestellt.

Operation	**Input**	**Output**	**Forward**	**Bidirectional**	**Random-Access**
`=`	•		•	•	•
`==`	•		•	•	•
`!=`	•		•	•	•
`*`	• (nur lesend)	• (nur schreibend)	•	•	•
`->`	•		•	•	•
`++`	•	•	•	•	•
`--`				•	•
`[ ]`					• [1]
`+ += - -=`					•
`< > <= >=`					•

[1] `I[n]` wird umgesetzt in `*(I + n)` für einen Interator `I`

14.3. Beispiel einer verketteten Liste mit Iteratoren

Im folgenden wird ein Beispiel für eine verkettete Liste mit einfachen Input-Iteratoren vorgestellt. Der Iterator wird als geschachtelte Klasse innerhalb der Listen-Klasse deklariert und definiert. Die einzige Eigenschaft der Iterator-Klasse ist ein Zeiger auf das aktuelle Listenelement.

```
#ifndef liste_h
#define liste_h liste_h
#include<cstddef>        // für NULL
#include<cassert>        // für assert

template<class T> class Liste
{
   private:
      struct Listenelement
      {
         T Daten;
         Listenelement *Next, *Prev;
         Listenelement(): Next(NULL), Prev(NULL) // Konstruktor
         { }
      };

      Listenelement *Ende, *Anfang;
      int Anzahl;

   public:
      Liste();                          // Konstruktor
      Liste(const Liste&);              // Kopierkonstruktor
      virtual ~Liste();                 // Destruktor
```

```
Liste& operator=(const Liste&);  // Zuweisungsoperator

bool empty() const { return Anzahl == 0; }
int size()   const { return Anzahl; }

// am Anfang bzw. Ende einfügen
void push_front(const T&);
void push_back(const T&);

// am Anfang bzw. Ende entnehmen
void pop_front();
void pop_back();

// am Anfang bzw. Ende lesen
T&       front();
const T& front() const;
T&       back();
const T& back() const;

// ============================================================
// Jetzt kommt die geschachtelte Klasse für den Iterator:
class Iterator
{
   private:
      Listenelement* aktuell;    // Zeiger auf aktuelles Element

   public:
      friend class Liste<T>;     // wg. erase-Zugriff auf aktuell
      Iterator(Listenelement* Init = NULL): aktuell(Init)
      { }

      Iterator(const Liste& L)
      {  *this = L.begin(); }

      const T& operator*() const // Dereferenzierung
      {  return aktuell->Daten; }

      T& operator*()             // Dereferenzierung
      {  return aktuell->Daten; }

      Iterator& operator++()     // prefix
      {
         if (aktuell)
            aktuell = aktuell->Next;
         return *this;
      }

      Iterator operator++(int)   // postfix
      {
         Iterator temp = *this;
         ++*this;
         return temp;
      }

      bool operator==(const Iterator& x) const
      {  return aktuell == x.aktuell; }

      bool operator!=(const Iterator& x) const
```

```
      {  return aktuell != x.aktuell; }
}; // Iterator
// Hier ist Deklaration und Definition des Iterators zu Ende!
// ========================================================

// Methoden der Klasse Liste, die die Klasse Iterator benutzen:

Iterator begin() const
{  return Iterator(Anfang); }

Iterator end() const
{  return Iterator(); }          // NULL-Iterator

void erase(Iterator& pos)        // Element löschen
{
   if (pos.aktuell == Anfang)
   {
      pop_front();
      pos.aktuell = Anfang;      // neuer Anfang
   }
   else
      if (pos.aktuell == Ende)
      {
         pop_back();
         pos.aktuell = Ende;
      }
      else  // zwischen zwei Elementen ausketten
      {
         pos.aktuell->Next->Prev = pos.aktuell->Prev;
         pos.aktuell->Prev->Next = pos.aktuell->Next;
         Listenelement *temp = pos.aktuell;
         pos.aktuell = pos.aktuell->Next;
         delete temp;
         Anzahl--;
      }
}

// Vor pos einfügen
Iterator insert(Iterator pos, const T& Wert)
{
   if (pos == begin())
   {
      push_front(Wert);
      return Iterator(Anfang);
   }
   if (pos == end())
   {
      push_back(Wert);
      return Iterator(Ende);
   }
   // zwischen 2 Elementen einketten
   Listenelement *temp = new Listenelement;
   temp->Daten = Wert;
   temp->Next = pos.aktuell;
   temp->Prev = pos.aktuell->Prev;
   pos.aktuell->Prev->Next=temp;
   pos.aktuell->Prev = temp;
   Anzahl++;
```

```
         return Iterator(temp);
      }
};


/* ====== Implementierung der Klasse Liste ===================*/
// soweit nicht schon geschehen

template<class T>
inline Liste<T>::Liste() :Ende(NULL), Anfang(NULL), Anzahl(0)
{ }                                              // Konstruktor

template<class T>
inline Liste<T>::Liste(const Liste<T>& L)        // Kopierkonstruktor
:Ende(NULL), Anfang(NULL), Anzahl(0)
{
   Listenelement *temp = L.Ende;
   while(temp)
   {
      push_front(temp->Daten);
      temp = temp->Prev;
   }
}

template<class T>
inline Liste<T>::~Liste()                        // Destruktor
{
   while(!empty())
      pop_front();
}

template<class T>                                // Zuweisungsoperator
inline Liste<T>& Liste<T>::operator=(const Liste<T>& L)
{
   if (this != &L)     // Selbstzuweisung ausschließen
   {
      while(!empty())
         pop_front();  //  alles löschen
      // ... und neu aufbauen
      Listenelement *temp = L.Ende;
      while(temp)
      {
         push_front(temp->Daten);
         temp = temp->Prev;
      }
   }
   return *this;
}

template<class T>
inline void Liste<T>::push_front(const T& Dat)
{
   Listenelement *temp = new Listenelement;
   temp->Daten = Dat;
   temp->Prev = NULL;
   temp->Next = Anfang;
   if (!Anfang)
      Ende=temp;        // einziges Element
```

```
   else
      Anfang->Prev = temp;
   Anfang = temp;
   Anzahl++;
}

template<class T>
inline void Liste<T>::push_back(const T& Dat)
{
   Listenelement *temp = new Listenelement;
   temp->Daten = Dat;
   temp->Prev = Ende;
   temp->Next = NULL;
   if (!Ende)
      Anfang=temp;        // einziges Element
   else
      temp->Prev->Next = temp;
   Ende = temp;
   Anzahl++;
}

template<class T>
inline void Liste<T>::pop_front()
{
   assert(!empty());
   Listenelement *temp = Anfang;
   Anfang = temp->Next;
   if (!Anfang)
      Ende = NULL;   // d.h. kein weiteres Element vorhanden
   else
      Anfang->Prev = NULL;
   delete temp;
   Anzahl--;
}

template<class T>
inline void Liste<T>::pop_back()
{
   assert(!empty());
   Listenelement *temp = Ende;
   Ende = temp->Prev;
   if (!Ende)
      Anfang = NULL;   // d.h. kein weiteres Element vorhanden
   else
      Ende->Next = NULL;
   delete temp;
   Anzahl--;
}

template<class T>
inline T& Liste<T>::front()
{
   assert(!empty());
   return Anfang->Daten;
}

template<class T>
inline const T& Liste<T>::front() const
```

```
{
   assert(!empty());
   return Anfang->Daten;
}

template<class T>
inline T& Liste<T>::back()
{
   assert(!empty());
   return Ende->Daten;
}

template<class T>
inline const T& Liste<T>::back() const
{
   assert(!empty());
   return Ende->Daten;
}

#endif
```

Zu dieser Beispiel-Klasse wird nun ein Beispiel-Hauptprogramm gezeigt. Dabei werden wieder die Funktionen `Drucken()` und `for_each()` aus dem Abschnitt *Einführung* dieses Kapitels verwendet. Im Hauptprogramm wird eine Liste mit `int`-Zahlen angelegt und mit den Quadratzahlen von 0 bis 9 gefüllt. Da die Quadratzahlen immer vorne eingefügt werden, stehen die Zahlen in der Liste in umgekehrter Reihenfolge. Dann wird ein Iterator angelegt und mit ihm das "Vorwärts-Gehen" in der verketteten Liste demonstriert. Dann wird die 36 aus der Liste wieder entfernt und abschließend alle Elemente der Liste ausgegeben.

```
#include "liste.h"
#include <iostream>

//using namespace std;

void Drucken(int& x)                    // aus dem Abschnitt Einführung
{
   cout.width(4);
   cout << x ;
}

template<class Iterator, class Funktion>       // Abschnitt Einführung
void for_each(Iterator Anfang, Iterator Ende, Funktion f)
{
   while(Anfang != Ende) f(*Anfang++);
}

int main()
{
   int i,x;
   Liste<int> L;                        // Liste mit int-Zahlen

   for (i = 0; i < 10; ++i)
   {
      x = i*i;
      L.push_front(x);                  // mit Quadratzahlen 0..81 füllen
   }

   Liste<int>::Iterator ListIter(L);
```

```
   cout << "*ListIter = " << *ListIter << endl;
   cout << "ListIter++;" << endl;
   ListIter++;
   cout << "*ListIter = " << *ListIter << endl;

   while(ListIter++ != L.end())        // 36 löschen, falls vorhanden
      if (*ListIter == 36)
      {
         cout << *ListIter << " wird geloescht\n";
         L.erase(ListIter);
         cout << *ListIter << " an aktueller Position\n";
         break;
      }

   // Ausgabe
   for_each(L.begin(), L.end(), Drucken);
   cout << endl;
}
```

Dieses Programm gibt folgendes auf dem Bildschirm aus:

```
*ListIter = 81
ListIter++;
*ListIter = 64
36 wird geloescht
25 an aktueller Position
  81  64  49  25  16   9   4   1   0
```

14.4. Spezielle Iteratoren

Neben den oben genannten Iteratoren gibt es noch eine Reihe weiterer spezieller Iteratoren.

Reverse-Iteratoren

Ein Reverse-Iterator ist bei einem bidirektionalen Iterator immer möglich. Ein Reverse-Iterator durchläuft die Datenelemente rückwärts mit dem Inkrementatoroperator ++. Beginn und Ende der Daten für Reverse-Iteratoren werden durch die Methoden `rbegin()` und `rend()` geliefert. Dabei verweist `rbegin()` auf das letzte Element und `rend()` auf die fiktive Position vor dem ersten Datenelement.

Insert-Iteratoren

Mit den bisherigen Iteratoren können nur vorhandene Werte in einer Datenstruktur überschrieben werden. Um Daten einzufügen werden Insert-Iteratoren benötigt. Je nach gewünschter Einfügeposition kann eine der drei folgenden Varianten verwendet werden.

1. `front_insert_iterator`
   Dieser Insert-Iterator fügt Daten am Anfang der Datenstruktur ein. Dazu muss die Klasse der Datenstruktur die Methode `push_front()` zur Verfügung stellen. Zusätzlich gibt es die Funktion `front_inserter()`, die einen `front_insert_iterator` zurückgibt.
2. `back_insert_iterator`
   Dieser Insert-Iterator fügt Daten am Ende der Datenstruktur ein. Dazu muss die Klasse der Datenstruktur die Methode `push_back()` zur Verfügung stellen. Zusätzlich gibt es die Funktion `back_inserter()`, die einen `back_insert_iterator` zurückgibt.
3. `insert_iterator`
   Dieser Insert-Iterator fügt Daten an einer ausgewählten Position ein. Dazu muss die Klasse der Datenstruktur die Methode `insert()` zur Verfügung stellen. Die Anwendung ist ähnlich wie bei den anderen beiden Insert-Iteratoren, nur dass dem Insert-Iterator die gewünschte Einfügeposition

mitgegeben werden muss. Zusätzlich gibt es die Funktion `inserter()`, die einen `insert_iterator` zurückgibt.

Stream-Iteratoren

Stream-Iteratoren dienen zum sequentiellen Lesen und Schreiben von Datenströmen mit den bekannten Operatoren << und >>. Der `istream`-Iterator ist ein Input-Iterator und der `ostream`-Iterator ein Output-Iterator.

Beispiel:

```
ifstream Quelle("Datei.txt");
istream_iterator<string> Pos(Quelle), Ende;
while (Pos != Ende)
   cout << *Pos << endl;
```

Die Dereferenzierung von `Pos` in der Schleife gibt nicht nur den gelesenen Wert zurück, sondern bewirkt auch das Weitergehen im Datenstrom zur nächsten Zeichenkette. Durch Erben von der Klasse `istream_iterator` und redefinieren einiger Methoden lassen sich eigene `istream`-Iteratoren mit besondern Eigenschaften schreiben. Dem Konstruktor eines `ostream`-Iterators kann wahlweise eine Zeichenkette zur Trennung von Elementen mitgegeben werden.

Beispiel:

```
// Kopiert die Datei Datei.txt in die Datei Ergebnis.txt.
// In der Ergebnis.txt wird in jeder Zeile ein * vorgesetzt.
ifstream Quelle("Datei.txt");
ofstream Ziel("Ergebnis.txt");

istream_iterator<string> iPos(Quelle), Ende;
ostream_iterator<string> oPos(Ziel, "*\n");

while (iPos != Ende)
   *oPos++ = *iPos++;
```

15. Der C++ 2011 – Standard

Nach der Verabschiedung des C++-Standards im Jahr 2003 hatte es ziemlich lange gedauert, bis ein neuer Standard veröffentlicht wurde. Man hatte sich ziemlich viel vorgenommen: Viele neue Features von anderen Programmiersprachen sollten auch in C++ verfügbar sein. Dadurch waren allerdings auch einige wesentliche Änderungen notwendig. Gleichzeitig musste auch immer auf die Kompatibilltät zu den vorigen Standards geachtet werden. Zuerst wurde davon ausgegangen, dass der neue Standard in 2008 oder 2009 fertig werden würde; daher wurde er als C++ 0x - Standard bezeichnet. Aber es dauerte noch bis zum 11.10.2011, bis der neue Standard unter der Bezeichnung ISO/IEC 14882:2011 (kurz C++ 11) verabschiedet wurde.

15.1. Doppelte >>

Die Verwendung von geschachtelten Templates, wie z.B. in der folgenden Zeile,

```
vector<vector<int>> Matrix;
```

verursachte vor dem 2011-Standard eine Fehlermeldung, da bisher durch das Erkennen des längst möglichen Tokens das >> als Schiebe- bzw. Einleseoperator erkannt wurde. Um das Problem zu umgehen, musste bisher ein Leerzeichen dazwischen eingefügt werden. Im 2011-Standard wird diese Zeile korrekt erkannt, da < und > jetzt als Klammerpaar erkannt wird und dieses eine höhere Priorität als der Schiebe- und Einleseoperator hat.

15.2. Neuer Nullzeiger

Es wurde ein neuer Nullzeiger eingeführt mit dem Namen `nullptr`. Dieser ersetzt den bisherigen `NULL`-Zeiger. Das Problem, das damit beseitigt wird, liegt in der impliziten Typumwandlung des `NULL`-Zeigers, da der `NULL`-Zeiger per `#define` als Zahl 0 definiert ist. Dies lässt sich mit dem folgenden Beispiel verdeutlichen:

```
void f(char *);
void f(int);

f(0);              // ruft f(int) auf
f(NULL);           // ruft auch f(int) auf
f((char *) NULL); // ruft jetzt f(char *) auf
f(nullptr);        // ruft im C++ 2011-Standard problemlos f(char *) auf
```

15.3. Verallgemeinerte Initialisierung

Bisher gab es mehrere Möglichkeiten, Objekte und Variablen zu initialisieren. Im Fall von

```
Typ(value)
```

wird je nach Definition mal ein Konstruktor aufgerufen und mal ein Typecasting durchgeführt. Gerade in Templates ist dies nicht immer eindeutig und damit sehr gefährlich. Im 2011-Standard wurden daher die **verallgemeinerten Initialisierungslisten** (Generalized Initializer Lists) eingeführt. Sie sind eine Liste von Elementen, die in geschweifte Klammern eingeschlossen werden.

```
Typ var1 = Typ{1, 2};
Typ var2 = {1, 2};
Typ var3{1, 2};
return {1, 2, 3};
```

Klassen können dann auch mit dem **Sequenz-Konstruktor** ausgestattet werden, der eine Initialisierungsliste als Argument vom Typ `std::initializer_list<T>` bekommt. Dabei muss die Größe der Liste nicht bekannt sein.

Beispiel:

kap15_01.cpp

```
01 #include <iostream>
02 #include <initializer_list>
03
04 using namespace std;
05
06 class MyVec
07 {
08       unsigned Len;
09       int *Array;
10    public:
11       MyVec(initializer_list<int> Seq)
12       : Len(Seq.size()), Array{new int[Len]}
13       {  copy(Seq.begin(), Seq.end(), Array);  }
14       ~MyVec()
15       {  delete [] Array;  }
16       friend ostream &operator<<(ostream &ostr, MyVec &v)
17       {
18          for (unsigned i = 0; i < v.Len; i++)
19             ostr << "Array[" << i << "] = " << *(v.Array + i) << endl;
20          return ostr;
21       }
22 };
23
24 int main()
25 {
26    MyVec Primzahlen{2, 3, 5, 7, 11, 13, 17};
27
28    cout << "Primzahlen:" << endl << Primzahlen;
29
30    return 0;
31 }
```

Bei der Initialisierung mit `{}` hat ein eindeutig passender Sequenz-Konstruktor Vorrang gegenüber dem Standard-Konstruktor, bevor das bisherige Verhalten greift.

15.4. Automatischer Datentyp `auto`

Bisher konnte mit `auto` bei der Deklaration von Variablen angegeben werden, dass sie lokal sind ("automatische" lokale Variablen; bei globalen Variablen durfte es nicht verwendet werden). Da dieses Schlüsselwort optional war und in der Praxis wohl niemand verwendet hat, wurde es für den 2011-Standard umdefiniert. Jetzt bestimmt es für die Definition von Variablen den Datentyp eines Ausdruckes. Dies funktioniert natürlich nur, wenn die Variable auch initialisiert wird.

```
auto x = 3.1415;
auto Erg = funktion();
```

Gerade bei nachträglichen Typänderungen erleichtert sich damit die Wartbarkeit der Programme. Aber auch für die Range-For-Schleife (siehe nächsten Abschnitt) kann dies hervorragend verwendet werden.

15.5. Range-For-Schleife

Bisher musste z.B. für die Ausgabe aller `vector`-Elemente etwas umständlich der Iterator verwendet werden. Mit dem neuen Range-For (`for`-Schleife mit nur einem Parameter) kann diese Schleife viel

übersichtlicher geschrieben werden. Durch die Verwendung der neuen Definition von `auto` wird der Typ automatisch ermittelt. Die Range-For-Schleife ermittelt dann automatisch den Bereich vom ersten bis zum letzten Element. Dies kann dann auch auf gewöhnliche Arrays sowie auf verallgemeinerte Initialisierungslisten angewendet werden.

Dabei ist zu beachten, dass der Iterator der Range-For-Schleife nicht mehr ein Zeiger auf das Datenelement sondern das Datenelement selber ist. D.h. beim Zugriff auf das Datenelement mit dem Iterator darf nicht mehr der Variablenoperator `*` verwendet werden. Ferner kann dadurch der Iterator in der Range-For-Schleife auch als Referenzvariable definiert werden.

Beispiel:

kap15_02.cpp

```
01 #include <iostream>
02 #include <vector>
03
04 using namespace std;
05
06 int main()
07 {
08    vector<int> Vektor {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
09    int Array[10] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};
10
11    for (vector<int>::iterator it = Vektor.begin(); it != Vektor.end(); it++)
12       cout << *it << endl;
13
14    for (auto &it: Vektor) // uebersichtlicher mit auto
15       cout << it << endl;
16
17    for (auto &it: Array) // auch auf Arrays anwendbar
18       cout << it << endl;
19
20    // auch auf verallgemeinerte Initialisierungslisten anwendbar:
21    for (auto Prim: {2, 3, 5, 7, 11, 13, 17} )
22       cout << Prim << endl;
23
24    return 0;
25 }
```

Ein großer Vorteil ist dabei, dass es gesichert ist, dass alle Elemente der Datenmenge durchlaufen werden und dass auch nicht das Ende der Datenmenge ausversehens überschritten wird.

15.6. R-Wert-Referenzen

Normale Referenzen können nur auf L-Werte verweisen, also auf Ausdrücke, deren Werte eine Speicheradresse haben. Nur mit dem `const`-Zusatz war es möglich, dass eine Referenz auf einen R-Wert verweist. Dazu wird ein temporärer Speicherplatz belegt, in dem der R-Wert gespeichert wird - d.h. die Referenz verweist intern doch auf eine Speicheradresse, die aber keinen Namen hat. Durch das `const` kann jedoch der Wert in dieser Speicheradresse nicht mehr geändert werden (siehe Beispielprogramm).

Mit dem C++ 2011 - Standard wurden nun **R-Wert-Referenzen** (**rvalue references**) eingeführt, also Referenzen, die auf Ausdrücke verweisen, die keine Speicheradresse haben. Bei genauerer Betrachtung wird aber auch hier ein temporärer Speicherplatz für den R-Wert belegt, aber der Wert in diesem Speicherplatz ist variabel. Bei einer R-Wert-Referenz werden zwei kaufmännische Unds (`&&`) statt nur einem verwendet. R-Wert-Referenzen können nicht auf L-Werte verweisen; dafür gibt es schließlich die normalen Referenzen.

Beispiel:

kap15_03.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 int main()
06 {
07    int i = 5 + 6;
08    int &lr1 = i;
09    const int &lr2 = 4 + 5; // lvalue reference; nur moeglich wegen const
10    // int &lr3 = 3 + 4;    // Fehler: invalid initialization of non-const reference
11                            //         of type 'int&' from an rvalue of type 'int'
12    int &&rr1 = 3 + 4;      // rvalue reference
13    // int &&rr2 = i;       // Fehler: cannot bind 'int' lvalue to 'int&&'
14
15    cout << "Wert der lvalue reference: " << lr1 << endl;
16    cout << "Wert der const lvalue reference: " << lr2 << endl;
17    cout << "Wert der rvalue reference: " << rr1 << endl; << endl;
18
19    cout << "Adresse der Variable i: " << &i << endl;
20    cout << "Adresse der lvalue reference: " << &lr1 << endl;
21    cout << "Adresse der const lvalue reference: " << &lr2 << endl;
22    cout << "Adresse der rvalue reference: " << &rr1 << endl; << endl;
23
24    // lr2 = 5 + 5; // Fehler: assignment of read-only reference 'lr2'
25    rr1 = 4 + 4;
26
27    cout << "Wert der const lvalue reference: " << lr2 << endl;
28    cout << "Wert der rvalue reference: " << rr1 << endl;
29    cout << "Adresse der const lvalue reference: " << &lr2 << endl;
30    cout << "Adresse der rvalue reference: " << &rr1 << endl;
31
32    return 0;
33 }
```

Die folgende Ausgabe dieses Programmes zeigt, dass der Speicher für drei `int`-Werte belegt wurde. Einmal für die Variable `i` (die Referenz `lr1` hat die gleiche Adresse wie die Variable `i`) sowie für die beiden temporären Werte, auf die die Referenzen `lr2` und `rr1` verweisen.

```
Wert der lvalue reference: 11
Wert der const lvalue reference: 9
Wert der rvalue reference: 7

Adresse der Variable i: 0x6efed8
Adresse der lvalue reference: 0x6efed8
Adresse der const lvalue reference: 0x6efedc
Adresse der rvalue reference: 0x6efee0

Wert der const lvalue reference: 9
Wert der rvalue reference: 8
Adresse der const lvalue reference: 0x6efedc
Adresse der rvalue reference: 0x6efee0
```

Zusammen mit dem Überladen von Funktionen kann unterschieden werden, ob ein Ausdruck einen L-Wert oder nur einen R-Wert hat. Dazu werden zwei überladene Funktionen definiert; die erste Funktion erhält eine L-Wert-Referenz, die zweite Funktion erhält eine R-Wert-Referenz. Der Compiler entscheidet damit eigenständig, welche der beiden Funktionen aufgerufen werden muss. Dabei kann die erste Funktion auch mit einer `const`-Referenz angegeben werden, denn die zweite Funktion mit der expliziten R-Wert-Referenz hat bei einem Aufruf mit einem R-Wert Vorrang.

Beispiel:

kap15_04.cpp

```
01 #include <iostream>
02
03 using namespace std;
04
05 void printReference (const int &value)
06 {
07    cout << "lvalue: value = " << value << " / Adresse: " << &value << endl;
08 }
09
10 void printReference (int &&value)
11 {
12    cout << "rvalue: value = " << value << " / Adresse: " << &value << endl;
13 }
14
15 int getValue ()
16 {
17    int ii = 10;
18    cout << "lok. Var.: ii = " << ii << " / Adresse: " << &ii << endl;
19    return ii;
20 }
21
22 int main()
23 {
24    int i = 11;
25
26    cout << "Variable i: " << i << " / Adresse: " << &i << endl;
27    printReference(i);
28    printReference(getValue());
29
30    return 0;
31 }
```

Beim ersten Aufruf von `printReference` wird die Funktion mit der L-Wert-Referenz aufgerufen. Anhand der Adressen der Variablen ist zu sehen, die lokale Variable `value` auf die gleiche Stelle zugreift wie die Variable `i` im Hauptprogramm. Dagegen unterscheiden sich die Adressen der lokalen Variablen `value` und `ii` beim zweiten Aufruf von `printReference`. Denn die Funktion `getValue` gibt nur einen temporären Wert zurück, der als R-Wert-Referenz weitergegeben wird; folglich wird die zweite Funktion `printReference` aufgerufen, denn die Variable `ii` ist beim Aufruf von `printReference` nicht mehr gültig.

```
Variable i: 11 / Adresse: 0x6efed4
lvalue: value = 11 / Adresse: 0x6efed4
lok. Var.: ii = 10 / Adresse: 0x6efeac
rvalue: value = 10 / Adresse: 0x6efee0
```

15.7. Verschieben geht schneller als Kopieren

Die Idee vom Überladen von Kopierkonstruktor und Zuweisungsoperator ist u.a. das Kopieren von dynamisch reservierten Speicherbereichen (siehe Kapitel 11.5). Wird dann beim Erzeugen eines neuen Objektes bzw. bei der Zuweisung eines Objektes ein temporäres Objekt angegeben, so werden erst neue Speicherbereiche reserviert und die dynamischen Speicherbereiche dorthin kopiert, nur um dann bei dem temporären Objekt die reservierten Speicherbereiche wieder freizugeben. Deutlich schneller geht es, wenn die Zeiger auf die reservierten Speicherbereiche verschoben werden, d.h. der Zeiger auf den reservierten Speicherbereich wird in das Zielobjekt kopiert und anschließend der Zeiger im Quellobjekt auf `nullptr` gesetzt.

Zur Erkennung eines temporären Objektes können die R-Wert-Referenzen verwendet werden (siehe voriger Abschnitt). Damit lassen sich nun **Verschiebe-Konstruktoren** (**move constructor**) und **Verschiebe-Zuweisungsoperatoren** (**move assignments**) definieren.

Beispiel:

kap15_05.cpp

```
001 #include <iostream>
002
003 using namespace std;
004
005 class Array
006 {
007    int size;
008    int *ptr;
009    string name;
010 public:
011    // Standardkonstruktor
012    Array()
013    : size(0), ptr(nullptr), name("leeres Array")
014    {  cout << "Standardkonstruktor     Name: " << name << endl;  }
015
016    // Konstruktor
017    Array(int s, string n)
018    : size(s), ptr(new int[s]), name(n)
019    {  cout << "Konstruktor             Name: " << name << endl;  }
020
021    // Kopierkonstruktor
022    Array(const Array& arr)
023    : size(arr.size), ptr(new int[arr.size]), name(arr.name + " - kopiert")
024    {
025       for (int i = 0; i < size; i++)
026          ptr[i] = arr.ptr[i];
027       cout << "Kopierkonstruktor       Name: " << name << endl;
028    }
029
030    // Verschiebekonstruktor
031    Array(Array&& arr)
032    : size(arr.size), ptr(arr.ptr), name(arr.name + " - verschoben")
033    {
034       arr.size = 0;
035       arr.ptr  = nullptr;
036       arr.name = "verschobenes Array";
037       cout << "Verschiebekonstruktor   Name: " << name << endl;
038    }
039
040    // Zuweisungsoperator
041    Array &operator=(const Array &arr)
042    {
043       delete [] ptr;
044
045       size = arr.size;
046       ptr  = new int[arr.size];
047       name = arr.name + " - kopiert (Zuweisung)";
048       for (int i = 0; i < size; i++)
049          ptr[i] = arr.ptr[i];
050       cout << "Zuweisungsoperator      Name: " << name << endl;
051       return *this;
```

```
052     }
053
054     // Verschiebezuweisungsoperator
055     Array &operator=(Array &&arr)
056     {
057        size = arr.size;
058        ptr = arr.ptr;
059        name = arr.name + " - verschoben (Zuweisung)";
060        arr.size = 0;
061        arr.ptr = nullptr;
062        arr.name = "verschobenes Array (Zuweisung)";
063        cout << "Verschiebezuweisungsop. Name: " << name << endl;
064        return *this;
065     }
066
067     // Destruktor
068     ~Array()
069     {
070        delete [] ptr;
071        cout << "Destruktor              Name: " << name << endl;
072     }
073
074     friend ostream &operator<<(ostream &ostr, Array &arr)
075     {
076        ostr << "Objekt " << arr.name;
077        return ostr;
078     }
079 };
080
081 // Diese Funktion gibt ein Objekt per Wert
082 // - d.h. ein temporaeres Objekt - zurueck
083 Array func_temporary_array_object(int sz)
084 {
085     Array m(Array(sz, "lokales Array")); // Konstruktor
086     return m;
087 } // Destruktor vom lokalen Objekt m
088
089 int main()
090 {
091     Array VArr1(3, "main Array");                // Konstruktor
092     Array VArr2(VArr1);                          // Kopierkonstruktor
093     Array VArr3(func_temporary_array_object(3)); // Verschiebekonstruktor
094     Array VArr4, VArr5;                          // Standardkonstruktor
095     VArr4 = VArr2;                               // Zuweisungsoperator
096     VArr5 = func_temporary_array_object(5);      // Verschiebezuweisungsop.
097
098     cout << "Objekte:" << endl;
099     cout << VArr1 << endl;
100     cout << VArr2 << endl;
101     cout << VArr3 << endl;
102     cout << VArr4 << endl;
103     cout << VArr5 << endl;
104
105     return 0;
106 } // Destruktoren von VArr5 bis VArr1
```

Bei Betrachtung der Ausgabe kommt allerdings etwas Verwunderung auf, denn die Bildschirmausgaben des Verschiebekonstruktors sind nicht zu finden. Wird der Verschiebekonstruktor gelöscht, funktioniert das

Programm immer noch. Der Kopierkonstruktor übernimmt nun die Funktion, da hier der Referenzparameter per `const` übergeben wird; damit können auch Referenzen auf R-Werte übergeben werden (siehe voriger Abschnitt). Aber auch nun ist die erwartete Bildschirmausgabe des Kopierkonstruktors nicht zu entdecken. Wird noch zusätzlich das `const` im Kopierkonstruktor entfernt, lässt sich das Programm nicht mehr compilieren.

Ausgabe:

```
Konstruktor             Name: main Array
Kopierkonstruktor       Name: main Array – kopiert
Konstruktor             Name: lokales Array
Standardkonstruktor     Name: leeres Array
Standardkonstruktor     Name: leeres Array
Zuweisungsoperator      Name: main Array - kopiert - kopiert (Zuweisung)
Konstruktor             Name: lokales Array
Verschiebezuweisungsop. Name: lokales Array - verschoben (Zuweisung)
Destruktor              Name: verschobenes Array (Zuweisung)
Objekte:
Objekt main Array
Objekt main Array – kopiert
Objekt lokales Array
Objekt main Array - kopiert - kopiert (Zuweisung)
Objekt lokales Array - verschoben (Zuweisung)
Destruktor              Name: lokales Array - verschoben (Zuweisung)
Destruktor              Name: main Array - kopiert - kopiert (Zuweisung)
Destruktor              Name: lokales Array
Destruktor              Name: main Array – kopiert
Destruktor              Name: main Array
```

Verantwortlich dafür ist das Auslassen von Kopier- bzw. Verschiebekonstruktor (*copy elision*), indem das Objekt gleich in dem Speicherbereich erzeugt wird, in den das Objekt kopiert bzw. verschoben werden soll, um damit unnötige Rechnerzeit zu vermeiden. Dies soll laut Standard auch dann vom Compiler durchgeführt werden, wenn dabei sichtbare Seiteneffekte (wie hier die Bildschirmausgaben) wegfallen. Trotzdem müssen Kopier- bzw. Verschiebekonstruktor definiert werden für den Fall, dass sich diese Optimierung nicht durchführen lässt.

Hinweis: Nicht jeder Compiler hält sich an diesen Standard, so dass das Ergebnis variieren kann. Daher sollte dringend vermieden werden, in den Kopier- und Verschiebekonstruktor zusätzliche Seiteneffekte zu integrieren.

15.8. Lambda-Ausdrücke

15.9. Variadic Templates

15.10. Aufrufen und Erben von Konstruktoren

15.11. Noch konstanter: `constexpr`

15.12. `default` und `delete` für automatisch erzeugte Konstruktoren und Operatoren

15.13. Datentyp ermitteln mit `decltype`

15.14. Multi-Threading

15.15. Neue Klassen in der STL

15.16. Timer

15.17. Zufallszahlen

16. Der C++ 2014 – Standard

17. Der C++ 2017 – Standard

18. Der C++ 2020 – Standard

Anhang 1: Schlüsselwörter

Die folgenden, alphabetisch sortierten Schlüsselwörter sind in C++ reserviert und dürfen anderweitig nicht verwendet werden!

and	and_eq	asm
auto	bitand	bitor
bool	break	case
catch	char	class
compl	const	const_cast
continue	default	delete
do	double	dynamic_cast
else	enum	explicit
extern	false	float
for	friend	goto
if	inline	int
long	mutable	namespace
new	not	not_eq
operator	or	or_eq
private	protected	public
register	reinterpret_cast	return
short	signed	sizeof
static	static_cast	struct
switch	template	this
throw	true	try
typedef	typeid	typename
union	unsigned	using
virtual	void	volatile
wchar_t	while	xor
xor_eq		