Die Programmiersprache C++

Copyright 2004 by Christian Bartl
bartl.christian@aon.at

Index

1. Einführung in die objektorientierte Programmierung ... 4
2. Eigenschaften einer objektorientierten Programmiersprache ... 4
  2.1. Objekt ... 4
  2.2. Klasse ... 4
  2.3. Vererbung ... 4
  2.4. Funktions- und Operatorüberlagerung ... 4
  2.5. Strenges Typenkonzept ... 4
3. Aufbau einer Klasse ... 4
  3.1. Headerdatei der Klasse ... 4
  3.2. CPP-Datei der Klasse ... 5
  3.3. Erzeugen eines Objekts ... 5
4. Konstruktor und Destruktor ... 5
  4.1. Standardkonstruktor und Destruktor ... 5
  4.2. Allgemeinkonstruktor ... 6
  4.3. Copyconstructor (Kopierkonstruktor) ... 6
5. Statische Variablen ... 7
6. Statische Funktionen ... 7
7. Dynamische Speicherreservierung ... 8
  7.1. Operator „new“ ... 8
  7.2. Operator „delete“ ... 8
  7.3. Dynamische Objekte ... 8
8. Das Schlüsselwort „const“ ... 8
9. Die Referenz ... 9
  9.1. Parameterübergabe mit Referenzen ... 9
10. Der This-Zeiger ... 9
11. Überladen von Funktionen ... 9
12. Überladen von Operatoren ... 9
  12.1. Überladen des Inkrement-Operators ... 10
13. Typcast (Typenumwandlungen) ... 10
14. Namespace (Namensraum) ... 11
  14.1. Verschachtelte Namespaces ... 11
  14.2. Using-Klausel ... 12
  14.3. Namespace „std“ ... 12
15. Friend-Funktionen ... 12
16. Die Klasse string ... 13
  16.1. Allgemein ... 13
  16.2. Eingabe von Strings ... 13
  16.3. Ausgabe von Strings ... 14
  16.4. Länge eines Strings ... 14
  16.5. Durchlaufen aller Zeichen eines Strings ... 14
  16.6. Kopieren von Strings ... 15
  16.7. Verketten von Strings ... 16
  16.8. Vergleichen von Strings ... 17
  16.9. Zeichen suchen in Strings ... 19
  16.10. Strings umwandeln in andere Datentypen ... 20
17. Ein- und Ausgabe ... 21
  17.1. Grundlagen ... 21
  17.2. Klassenhierarchie ... 21
  17.3. Der Zeichensatz ... 21

17.4. Ausgabe ... 22
17.4.1. Ausgabe von Umlauten ... 23
17.5. Eingabe ... 24
17.6. Dateibehandlungen ... 24
17.6.1. Binärdateien ... 25
17.6.2. Textdateien ... 25
17.6.3. Beispiele ... 26
18. Klassenbeziehungen (Relationen) ... 30
18.1. Assoziation (kennt) ... 30
18.2. Aggregation (hat, existenzunabhängig) ... 30
18.3. Komposition (hat, existenzabhängig) ... 30
18.4. Benutzt (uses a) ... 30
18.5. Vererbung, Generalisierung (inheritance, ist) ... 30
18.6. Darstellung der Klassenbeziehungen ... 31
18.6.1. Statisches Klassendiagramm ... 31
18.6.2. Statisches Objektdiagramm ... 31
19. Vererbung ... 32
19.1. Grundlagen ... 32
19.1.1. Spezialisierung ... 32
19.1.2. Generalisierung ... 32
19.1.3. Zugriffsspezifizierer ... 32
19.1.4. Konstruktor und Destruktor ... 33
19.1.5. Polymorphismus ... 33
19.2. Mehrfachvererbung ... 33
19.3. Methoden redefinieren (Überschreiben von Methoden) ... 34
19.4. Virtuelle Methoden ... 34
19.5. Abstrakte Basisklassen ... 34
19.6. Virtuelle Basisklasse ... 34
20. Template-Funktionen ... 34
21. Template-Klassen ... 35
22. Fehlerbehandlung ... 36
23. Ausnahmebehandlung ... 37
23.1. Exeptionspezifikation und Deklaration ... 38
24. Exception-Hierarchie in C++ ... 38
24.1. Erkennen logischer Fehler ... 39
24.2. Durch Exeption verursachte Speicherfehler vermeiden ... 39
24.3. Speicherbeschaffung mit „new“ ... 39
25. Dynamische Datenstrukturen ... 39
25.1. Einfachverkettete Liste ... 40
25.2. Doppeltverkettete Liste ... 41
25.3. Binärer Baum ... 41
25.4. Bubblesort für eine lineare Liste ... 41
25.5. Stack, Stapel, Keller ... 42
25.6. Queue ... 42
26. Baumstruktur ... 42
26.1. Der Suchbaum ... 43
26.2. Bäume speichern (Traversierung) ... 44
26.2.1. Inorder Traversierung ... 44
26.2.2. Postorder Traversierung ... 44
26.2.3. Preorder Traversierung ... 44
27. Statische Bibliotheken erzeugen ... 45

1. Einführung in die objektorientierte Programmierung

C++ wurde 1983 von der Firma AT&T und dem Programmierer Bjorne Stroustrup entwickelt. Die ursprüngliche Version von C++ war ein modernes C mit Klassen. Funktionen werden in C++ Methoden und Variablen werden Attribute genannt.

2. Eigenschaften einer objektorientierten Programmiersprache

2.1. Objekt

Objekte stellen durch ihre Beziehungen einen Ausschnitt der realen Welt dar. Sie kommunizieren durch Nachrichten.

2.2. Klasse

Die Klasse ist eine Definition von benutzerspezifischen Datentypen mit den dazugehörigen Operationen und Zugriffsmethoden. Eine Klasse ist eine Art Vorlage für ein Objekt (Vergleichbar mit Stylesheets unter CSS und HTML). Das Ganze fällt in jedem Fall unter Datenkapselung. Unter Datenkapselung versteht man, dass Methoden nur auf Attribute ihres eigenen Objektes und nicht auf Attribute eines fremden Objekts zugreifen können.

2.3. Vererbung

Die Vererbung erlaubt die Beschreibung von Klassenbeziehungen im Sinne von Spezialisierung bzw. Abstraktion.

2.4. Funktions- und Operatorüberlagerung

Die Funktions- und Operatorüberladung erlaubt dem Programmierer existente Funktionen und Operatoren für andere Parametertypen als die bisherigen zu definieren. Bei der Überladung von Funktionen und Operatoren kommen die zwei Begriffe „Polymorphismus" und „spätes bzw. frühes Binden" vor.

2.5. Strenges Typenkonzept

Das strenge Typenkonzept von C++ verhindert ungewollte Nebeneffekte bei der impliziten Typenumwandlung, wie es in C möglich ist.

3. Aufbau einer Klasse

Die Definition einer Klasse entspricht der einer Struktur. Sie beginnt mit dem Schlüsselwort „class“, dem ein beliebiger Name folgt (Klassennamen beginnen mit Großbuchstaben). Die Definition einer Klasse ist in einer Headerdatei ausprogrammiert. Ausprogrammiert wird sie in einer CPP-Datei.

3.1. Headerdatei der Klasse

```
class Klassenname
{
private:
	Memberattribute
public:
	Konstruktor
	Destruktor
	Methoden
	Public-Attribute
};
```

Alle Attribute, die unter "private" stehen, können von außen nicht verändert werden. Alle Attribute unter „public" können auch von außen verändert werden. Alle Memberattribute haben nach der Ungarischen Notation „m_" vor dem Attributnamen.

3.2. CPP-Datei der Klasse

In die CPP-Datei muss zuerst die Headerdatei, in der sich die Klassendefinition befindet, eingebunden werden. Um eine Methode aufzurufen benötigt man folgenden Syntax:

```
Rückgabedatentyp Klassenname :: Methodenname (Übergabeparameter)
{
        Code
}
```

Der Operator „::“ wird Scope-Operator oder Gültigkeitsoperator genannt.

3.3. Erzeugen eines Objekts

Um ein Objekt zu erzeugen muss man die Headerdatei der Klasse in die CPP-Datei einbinden. Zum Erzeugen wird folgende Syntax verwendet:

```
Klassenname Objektname;
```

Wird ein Objekt so erzeugt werden alle Memberattribute automatisch auf 0 initialisiert. Über den Objektnamen und den „.“-Operator kann man jetzt die Attribute und Methoden des Objekts aufrufen. Dazu benötigt man folgende Syntax:

```
Objektname.Attributname();
Objektname.Methodenname();
```

4. Konstruktor und Destruktor

4.1. Standardkonstruktor und Destruktor

Objekte gehören initialisiert, damit diese keine unerlaubten Werte annehmen können. Um Klassenattribute initialisieren zu können, benötigt man spezielle Methoden mit dem Namen Konstruktor und Destruktor. Diese beiden Methoden haben die Aufgabe Objekte bei ihrer Erzeugung zu initialisieren und bei ihrer Zerstörung aufzuräumen. Das heißt, dass jedes Mal, wenn ein Objekt erzeugt wird, automatisch der Konstruktor aufgerufen wird und jedes Mal, wenn ein Objekt aufhört zu existieren, der Destruktor. Falls kein Konstruktor und Destruktor definiert werden, werden Standardfunktionen verwendet. Der Name des Konstruktors ist gleich dem Namen der Klasse und der Name des Destruktors ist gleich der Klassenname vor dem eine Tilde (~) steht. Im Gegensatz zu Konstruktoren können Destruktoren keine Parameter übernehmen. Man kann aber einen Standardkonstruktor auch selbst definieren, welches den Vorteil hat, dass die Fehleranfälligkeit geringer wird. Jede Klasse hat mindestens einen Konstruktor.

Definition des Standardkonstruktors:

```
Klassenname();
```

Programmierung des Standardkonstruktors:

```
Klassenname :: Klassenname ()
{
	Memberattribute
}
```

Definition des Standarddekonstruktors:

```
~Klassenname();
```

Programmierung des Standarddestruktors:

```
Klassenname :: ~Klassenname ()
{
}
```

4.2. Allgemeinkonstruktor

Allgemeine Konstruktoren können im Gegensatz zu Standardkonstruktoren Argumente haben und sie können überladen werden. Das heißt, dass es mehrere Allgemeinkonstruktoren mit unterschiedlichen Parametern geben kann.

Definition des Allgemeinkonstruktors:

```
Klassenname (Übergabeparameter);
```

Programmierung des Allgemeinkonstruktors:

```
Klassenname :: Klassenname (Übergabeparameter)
{
	Memberattribute = Übergabeparameter;
}
```

Erzeugen eines Objekts:

```
Klassenname Objektname;    //Alle Memberattribute werden auf 0 gesetzt
Klassenname Objektname (Übergabeparameter);
```

4.3. Copyconstructor (Kopierkonstruktor)

Der Kopierkonstruktor dient dazu, ein Objekt mit einem anderen Objekt derselben Klasse zu initialisieren. Das Argument eines Kopierkonstruktors ist eine Referenz auf ein Objekt derselben Klasse. Weil ein Objekt, das dem Kopierkonstruktor als Argument dient, nicht verändert werden soll, ist es sinnvoll, es als Referenz auf „const" zu übergeben. Falls kein Kopierkonstruktor vorgegeben, wird dieser für jede Klasse bei Bedarf vom System erzeugt.

Syntax:

```
Klassenname (const Klassenname & Name)
{
	Attribut = Name.Attribut;
}
```

Aufruf:

Um den Kopierkonstruktor aufrufen zu können, muss zumindestens ein Objekt bestehen, dass ein Objekt auf ein anderes kopiert werden kann.

```
Klassenname zweitesObjekt = erstesObjekt;
```

Der Kopierkonstruktor wird bei Zuweisungen oder Initialisierungen mit Allgemeinkonstruktoren nicht aufgerufen. Der Kopierkonstruktor wird nur bei der Deklaration aufgerufen.

Hier wird der Kopierkonstruktor nicht aufgerufen:

```
Klassenname Objekt1, Objekt2;
Objekt1 = Klassenname (1, 2);     // Initialisierung mit dem
                                     Allgemeinkonstruktor
Objekt2 = Objekt1;                // Zuweisung
```

5. Statische Variablen

In einer Klasse können Elemente statisch, mit dem Schlüsselwort „static“, deklariert sein. Wird eine Variable als static deklariert, so legt der Compiler nur eine Version dieser Variable an, gleichgültig wie viele Objekte erzeugt werden. Dies hat zur Folge, dass alle Objekte auf diesen Wert gemeinsam zugreifen können. Statische Variablen kann man benutzen, um Werte zwischen einzelnen Objekten auszutauschen. Da das static-Attribut nur einmal existiert, wird jede Änderung ihres Wertes sofort für alle anderen Objekte verfügbar. Neben diesem Datenaustausch besteht auch die Möglichkeit, die Anzahl der erzeugten Objekte zu überwachen.

Deklaration:

```
static Datentyp Attributname;
```

Initialisierung:

```
Datentyp Klassenname :: Attributname = Wert;
```

Statische Klassenattribute, die als public definiert werden, können auch direkt im Programm angesprochen werden, mittels „.“ oder „::“ Operator.

```
Objektname.Attributname = Wert;
Datentyp Klassenname :: Attributname = Wert;
```

6. Statische Funktionen

Auch Funktionen können statisch sein. Derartige Funktionen sind dann nicht mehr von der Existenz eines Objektes abhängig. Sie sind für die ganze Klasse gültig und können daher über den Klassennamen und den Scope-Operator aufgerufen werden. Statische Funktionen können allerdings nur mit statischen objektunabhängigen Attributen zusammenarbeiten.

Deklaration:

```
static Rückgabedatentyp Funktionsname (Übergabeparameter);
```

Definition:

```
Rückgabedatentyp Funktionsname (Übergabeparameter)
{
        Code
}
```

Aufruf:

```
Klassenname :: Funktionsname (Übergabeparameter);
```

7. Dynamische Speicherreservierung

7.1. Operator „new“

Die Anforderung von Speicher während der Programmausführung ist in C eine umständliche und fehleranfällige Sache. C++ hat hier mit dem „new“-Operator eine weitaus bessere Möglichkeit zu bieten. Speicher wird immer im Konstruktor alokiert. Um mit dem „new“-Operator Speicher zu reservieren, benötigt man einen Zeiger vom gleichen Datentyp, wie der, mit dem man das Array anlegen möchte.

Syntax:
```
Zeigername = new Datentyp [Feldgröße];
```

7.2. Operator „delete“

Als Gegenstück zum „new“-Operator kennt C++ den „delete“-Operator. Er wird benutzt um Speicher, der mit „new“ alokiert wurde, wieder freizugeben. Der „delete“-Operator wird im Destruktor angewandt.

Syntax:
```
delete [] Zeigername;
```

7.3. Dynamische Objekte

Um Objekte dynamisch anzulegen muss man zuerst einen Zeiger vom Typ Klasse anlegen. Das ganze ergibt dann ein Array von Objekten. Diese können wie normale Objekte benutzt werden. Allerdings hat man hier keinen Objektnamen mehr sondern Nummern. Achtung, es darf kein Wert in das 0.Feld geschrieben werden, weil der Compiler ansonsten einen Speicherfehler meldet.

Konstruktor:
```
Zeigername = new Klassenname [];
```

Destruktor:
```
delete [] Zeigername;
```

Beschreiben eines Memberattributs:
```
Zeigername [Position >0].Attribut = Wert;
```

Aufrufen einer Methode;
```
Zeigername [Position >0].Methodenname ();
```

8. Das Schlüsselwort „const“

Neben der Möglichkeit mit „#define“ Konstanten zu definieren, bietet C++ das Schlüsselwort „const“. Es sorgt dafür, dass eine Variable während des Programmablaufes nicht mehr geändert werden kann. Das Einsatzgebiet dieses Schlüsselwortes ist dasselbe wie bei „#define“. Es ist jedoch diesem vorzuziehen, da der Compiler hier die verwendeten Datentypen und Gültigkeitsbereiche besser kontrollieren kann, während es sich beim „#define“ nur um eine Textersetzung handelt, bei der keine Typprüfung vorgenommen wird. Konstanten müssen bei der Deklaration sofort initialisiert werden.

Syntax:
```
const Datentyp Name = Wert;
```

Auch Funktionen können als „const" deklariert werden. „const" steht dabei hinter der Parameterliste und muss auch später bei der Funktionsdefinition angegeben werden. Es bewirkt, dass die Funktion als nur lesend vom Compiler eingestuft wird. Sie kann somit weder selbst die Objektdaten (private) ändern noch kann sie Methoden aufrufen, die dies können.

9. Die Referenz

Eine Referenz ist ein Datentyp, der einen Verweis auf ein Objekt liefert. Referenzen werden in C++ auch zur Parameterübergabe verwendet. Eine Referenz bildet einen Alias-Namen für ein Objekt, über den es ansprechbar ist. Ein Objekt hat damit zwei Namen. Eine Referenz wird genau wie eine Variable benutzt, der Compiler erledigt den Rest der Arbeit. Referenzen müssen bei der Deklaration initialisiert werden. Eine Referenz ist besser lesbar als die gleichwertige Zeigerdarstellung.

Syntax:

```
Datentyp &Referenzname = Variable;
```

9.1. Parameterübergabe mit Referenzen

Um Referenzen zu übergeben schreibt man als Übergabeparameter einfach „Datentyp &Referenzname". Beim Funktionsaufruf werden einfach die Variablen übergeben. Innerhalb der Funktion wird dann auch einfach mit den Referenznamen gearbeitet.

10. Der This-Zeiger

Dem Zugriff einer Methode auf Datenelemente eines Objektes wird beim Aufruf implizit ein Zeiger auf das Objekt der Elementfunktion als zusätzliches aktuelles Argument übergeben. Das heißt, jeder Methode werden automatisch Referenzen auf die Memberattribute mit übergeben. Mit Hilfe des This-Zeigers kann man Attribute einer Methode ansprechen, wenn diese in einem Block durch lokale gleichnamige Attribute überdeckt werden. Innerhalb einer Methode bezeichnet „this" einen Zeiger auf das aktuelle Objekt und „*this" das Objekt selbst, für das die Methode aufgerufen wurde.

Syntax:

```
this->Attribut = Wert;
*this.Attribut = Wert;
```

11. Überladen von Funktionen

Hat man für mehrere Datentypen gleichartige Funktionen zu schreiben, so müssen diese in C verschiedene Namen tragen, damit der Compiler diese unterscheiden kann. In C++ besteht die Möglichkeit, für gleichartige Funktionen innerhalb eines Namensbereiches den gleichen Namen zu vergeben. Dies nennt man Überladen von Funktionen (functionoverloading). Anhand der verschiedenen Signaturen (Datentypen) kann der Compiler die zum Datentyp passende Funktion auswählen. Die Signatur besteht aus dem Datentyp für Rückgabewert und Übergabeparameter.

12. Überladen von Operatoren

In C++ können alle Operatoren überladen werden, mit Ausnahme des „."-Operators (Punkt-Operator), „*"-Operators (Stern-Operator), „::"-Operators (Scope-Operator), „?:"-Operators (if-Operator) und des sizeof-Operators da sie bereits für Operanden von Klassen vorbelegt sind. Die Operaoren „*", „+", „-", „/" können sowohl in ihrer unären

(einstelligen) als auch in ihrer binären (zweistelligen) Bedeutung überladen werden. Beim Überladen ist zu beachten, dass sich die Arität (Stelligkeit (unär, binär)) und die Assoziativität (a+b+c = (a+b)+c = a+(b+c)) des Operators nicht ändert. Außerdem sollte der Operator das tun, was der Leser des Programms von ihm erwartet. Es ist zum Beispiel nicht sinnvoll den „+"-Operator für eine Multiplikation zu überladen. Das Erscheinen desselben Operators für verschiedene Aufgaben ist Teil des Polymorphismus-Konzeptes von C++. Das Überladen eines Operators erfolgt in Form einer Funktion, das heißt mit Hilfe einer return-Anweisung wird der gesuchte Operatorwert zurückgegeben.

Syntax:

```
Rückgabedatentyp operatorOperatorsymbol (const Datentyp &Name)
{
        return Datentyp (….);
}
```

Beispiel:

Für die Klasse Bruch mit den Argumenten Zaehler und Nenner soll der „+"-Operator überladen werden.

```
Bruch operator+ (const Bruch &b)
{
        return Bruch (zaehler*b.nenner + b.zaehler*nenner, b.nenner*nenner);
}
```

Die Schreibweise „a+b" wird vom Compiler als „a.operator + (b)" interpretiert. Dabei wird der rechte Wert der Summe als konstante Referenz übergeben, damit er nicht kopiert werden muss. Der linke Wert ist implizit gegeben. Analog können die übrigen arithmetischen Operatoren überladen werden. Ein Beispiel für das Überladen von Operatoren, wo dieses benötigt wird, ist die Bruchrechnung. Das Überladen von Operatoren wird nur von wenigen mächtigen Programmiersprachen unterstützt und realisiert dabei einen Teil des Polymorphismus-Konzeptes.

12.1. Überladen des Inkrement-Operators

Beim Überladen des Inkrement-Operators würde sich hier eine Zweideutigkeit ergeben, da die Stellung (Präfix, Postfix) des Operators nicht erkennbar ist. Gemäß der Norm wird daher definiert:

```
Operator++ ()          // Präfix-Form (++i)
Operator++ (int)       // Postfix-Form (i++)
```

Die Postfix-Form wird durch ein formales int-Argument unterschieden.

13. Typcast (Typenumwandlungen)

Um Umwandlungen typsicher zu machen, wurde der alte cast-Operator der C++-Norm durch 4 neue Umwandlungs-Operatoren für verschiedene Anwendungsbereiche ersetzt. Diese Umwandlungs-Operatoren haben die Form eines Templates, das heißt, bei ihnen muss der jeweilige Datentyp in spitzen Klammern angegeben werden.

Variablenname = static_cast <Zieldatentyp> (Variable oder Wert);

Dient zur definierten Umwandlung von Datentypen. Der „static_cast"-Operator ist für alle Typumwandlungen gedacht, die der Compiler als gültig betrachtet. Achtung, bei

Umwandlungen kann es einen Verlust von Informationen gegeben. Beispiel: int i = static_cast <int> (1.234);

Variablenname = const_cast <Zieldatentyp> (Variable oder Wert);
Der „const_cast“-Operator dient hauptsächlich dazu, vorübergehend einen als „const“ deklarierten Parameter als nicht „const“ zu deklarieren. Der Einsatz des „const_cast“-Operators sollte nicht missbraucht werden. Es sollte in keinem Fall dazu führen, das Konzept der Konstanten in C++ zu unterlaufen und möglichst alle Konstanten veränderbar zu machen.

Variablenname = dynamic_cast <Zieldatentyp> (Variable oder Wert);
Der „dynamic_cast“-Operator ist für die Umwandlung von polymorphen Objekten, wie sie bei Vererbung auftreten, gedacht.

Variablenname = reinterpreter_cast <Zieldatentyp> (Variable oder Wert);
Alle anderen nicht erwähnten Fälle sind Sache des „reinterpreter_cast“-Operators. Hier kann zum Beispiel ein Zeiger auf ein Objekt in einen Zeiger ganz anderer Art umgewandelt werden.

14. Namespace (Namensraum)

Namensräume sind Gültigkeitsbereiche, in denen beliebige Objekte wie Variablen, Methoden, Klassen und andere Namensräume deklariert werden können. Namensräume lösen folgendes Problem: Verwendet man umfangreiche eigene und fremde Bibliotheken, so ist die Wahrscheinlichkeit groß, dass irgendwann einmal zwei Methoden gleichen Namens auftreten. Deklariert man nun jede Bibliothek als eigenen Namensraum, so kann der Compiler die Funktionen gleichen Namens unterscheiden. Ein Namensraum hat in C++ folgende Eigenschaften:

- Alle in einem Namensraum enthaltenen Objekte können mittels Scope-Operator angesprochen werden.
- Für jeden Namensraum können Alias-Bezeichner festgelegt werden. Dies ist dann besonders nützlich, wenn der Name des Namensraums besonders lang ist.
- Wird die Verwendung eines bestimmten Namensraums mittels der using-Anweisung erklärt, so kann auf alle Objekte auch ohne Scope-Operator zugegriffen werden.

Deklaration und Definition:

```
namespace Name
{
        Code
}
```

Aufruf:

```
Name :: Variablenname;
Name :: Funktionsname();
```

14.1. Verschachtelte Namespaces

Um Variablen und Funktionen in einem Namensraum der sich in einem Namensraum befindet aufzurufen, muss man einfach wieder über die Namen und den Scope-Operator operieren.

```
Name1 :: Name2 :: Variable = Wert;
```

Für den verschachtelten Namensraum kann ein Alias eingeführt werden.

```
namespace NameGesamt = Name1 :: Name2;
```

14.2. Using-Klausel

Statt jede Variable oder Funktion mit der Namensraumangabe aufzurufen kann man mit der sogenannten Using-Klausel die Variablen und Funktionen von Namensräumen auch direkt verwenden. Diese Klausel überdeckt dann aber gleichnamige lokale Variablen.

Syntax:

```
using namespace Name;
```

Eine weitere Using-Klausel hebt die vorherige nicht auf, sondern ergänzt diese.

14.3. Namespace „std“

Alle Bibliotheken der „Standard Template Library“ (STL) sind im Namensraum „std“ zusammengefasst. Er wird erklärt durch die using-Anweisung „using namespace std“.

Syntax:

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

Alle Funktionsprototypen der C-include-Dateien, deren Dateiname auf „.h“ enden, gehören zur Programmiersprache C und zum globalen Namensraum. Dieselben Funktionen werden auch unter C++ zur Verfügung gestellt aber unter neuen Dateinamen. Diese Dateien unterscheiden sich im Namen durch ein vorangestelltes „c“ und das Fehlen der Dateiendung „.h“. Ausgenommen ist die Bibliothek iostream. Diese Funktionen sind im Namespace std definiert. Die Möglichkeit Funktionen mit der Dateiendung „.h“ einzubinden bleibt unberührt. Eine C-Headerdatei mit dem Namen „<name.h>“ heißt in C++ „<cname>“.

15. Friend-Funktionen

Ein Ziel der objektorientierten Programmierung (OOP) ist es die Daten zu verbergen. Dadurch, dass man den Zugriff auf die Daten nur durch Schnittstellenmethoden gestattet, kann man die Fehleranfälligkeit von Programmen verringern. Friend-Funktionen sind wieder ein Schritt in die andere Richtung. Mit friend-Funktionen kann man auch auf Daten zugreifen, die nicht zum eigenen Objekt gehören. Dies ist daher sparsam einzusetzen, da es eigentlich gegen das Prinzip der Datenkapselung verstößt. Mit dem Schlüsselwort „friend“ werden Klassen und Methoden bezeichnet, die die gleichen Rechte haben, wie die Schnittstellenmethoden selbst. Es ist gleichgültig ob friend-Funktionen im private- oder public-Bereich deklariert werden.

Syntax:

```
friend Rückgabedatentyp Funktionsname (Übergabeparameter);
```

Bei der Definition muss das Schlüsselwort „friend“ nicht angegeben werden.

16. Die Klasse string

16.1. Allgemein

Strings kann man in C++ entweder wie in C in char-Arrays oder in Zeigern auf char ablegen oder in Objekte der in C++ verfügbaren Klasse „string“. Diese erlaubt auf einfache Weise beliebig lange und dynamische Strings zu verwalten. Zudem sind viele Operationen deutlich einfacher und gefahrloser anzuwenden als in C. Zur Benutzung der Klasse string muss man die Headerdatei „string“ einbinden über den Befehl: #include <string>. Achtung nicht verwechseln mit cstring und string.h.
Ein Objekt der Klasse string wird durch Konstruktoren erzeugt.

- Default Konstruktor (leerer String)
  string Name;

- Typumwandlungskonstruktor von „const char *“ in „string“
  string Name („Hallo“);

- Typumwandlungskonstruktor
  string Name = „OK“;

- Kopierkonstruktor
  string Name (Name1);

- Kopierkonstruktor
  string Name = Name1;

- String enthält als x.Zeichen den Buchstaben ’u’
  string Name (x, ’u’);

- char-Array wird als „char *“ interpretiert und in dem string-Objekt Name abgelegt
  char Name [6] = “Hallo“;
  string str_Name (Name);

- char-Zeiger wird in string-Objekt Name abgelegt
  char * Name = “Wort“;
  string str_Name (Name);

Wichtig ist, dass zuerst immer ein leerer String erzeugt wird, der dann automatisch vergrößert bzw. wieder verkleinert wird. Eine Längenangabe sowie ein manuelles Vergrößern des Strings ist nicht notwendig.

16.2. Eingabe von Strings

Strings kann man entweder mit „cin>>“ eingeben oder mit der globalen Funktion „getline(…)“.

Da „cin>>“ führende Leerzeichen überliest und nach Beginn des Einlesens des 1.Zeichens stoppt, wenn es auf folgende Leerzeichen stößt, ist es gut geeignet um Strings getrennt einzugeben. Gibt man allerdings Wort1–Leerzeichen–Wort2 ein,

wird nur Wort gespeichert. Nicht zu vergessen ist das „cin>>" das abschließende Newline (also Enter) am Ende der Zeile im Tastaturpuffer lässt.

```
string Name;
cin>> Name;
```

Der Befehl „getline(…)" kombiniert mit „cin" überliest führende Leerzeichen nicht und stoppt die Eingabe, wenn im Eingabepuffer Newline steht (also die Enter-Taste gedrückt wurde). Daher ist der Befehl gut geeignet um Daten, die durch Leerzeichen getrennt sind, in einen String einzulesen.

```
string Name;
getline (cin, Name);
```

Um zwei oder mehr Werte mit einer Eingabe abzufragen aber in verschiedene Strings zu schreiben gibt es die Möglichkeit „getline(…)" ein Argument mitzugeben. Zum Beispiel sollen Vor- und Nachname, die durch ein Komma getrennt werden, über eine Abfrage in den String Vorname und Nachname geschrieben werden.

```
string Vorname, Nachname;
getline (cin, Vorname, ’,’);
getline (cin, Nachname, ’,’);
```

Dies bedeutet nichts anderes, als dass bis zum Komma eingelesen wird, dass Komma aber nicht in einen der beiden Strings geschrieben wird und dann wieder weiter eingelesen wird, bis ein Newline folgt.

16.3. Ausgabe von Strings

Die Ausgabe erfolgt über den Befehl „cout<<".

```
string Name;
cin>> Name;
cout<< Name;
```

16.4. Länge eines Strings

Die Länge eines Strings kann durch die Methode „size ()" ermittelt werden. Sie liefert die Länge als Wert des Datentyps „string :: size_type" zurück (Dies ist im Prinzip ein vorzeichenloser ganzzahliger Wert).

```
string :: size_type Laenge;
string Name;

Laenge = Name.size();                    //Laenge = 0

Name =  “Hallo“;
Laenge = Name.size();                    //Laenge =  5
```

16.5. Durchlaufen aller Zeichen eines Strings

Die Zeichen eines string-Objekts sind Variablen vom Datentyp char. Die einzelnen Zeichen können mit einer for-Schleife durchlaufen werden. Da ein string-Objekt

seine Länge gespeichert hat, benötigt man keine Variable die die Länge speichert. Um auf die Zeichen zugreifen zu können, verwendet man wie bei Arrays eckige Klammern „[]" (ohne Indexcheck) oder die Methode „at(…)" (mit Indexcheck).

Beispiel ohne Indexcheck

```
string Name = "Wert";
string :: size_type i;

for (i=0; i<Name.size(); i++)
  {
        cout<< Name [i];                //Ausgabe
        Name [i] = '0'                  //Eingabe
  }
```

Beispiel mit Indexcheck

```
string Name = "Wert";
string :: size_type i;

for (i=0; i<Name.size(); i++)
  {
        cout<< Name.at (i);             //Ausgabe
        Name.at (i) = '0'               //Eingabe
  }
```

Bei Verwendung der Methode „at(…)" können im Gegensatz zur Möglichkeit wie bei Arrays Zugriffsfehler in Form des Abbrechens des Programms erkannt werden.

16.6. Kopieren von Strings

Das Kopieren der string-Objekte mit der Wertzuweisung erzeugt eigenständige Kopien der Zeichenkette. Dies gilt auch bei Wertzuweisung con char-Arrays oder char-Zeigern.

```
string Name1= "Hallo", Name2;
Name2 = Name1;
        // Name2 ist ein von Name1 unabhängiges Objekt, das aber dieselbe
           Zeichenkette verwaltet

char * Name1 = "Wort";
string Name2;
Name2 = Name1;
        // Name2 ist ein string-Objekt, das die Zeichenkette "Wort" als String
           verwaltet.

string Name;
Name= 'a';
        // Name ist ein string-Objekt, das nur das Zeichen 'a' als String
           verwaltet.

string Name = 'a';
```

```
// Achtung FALSCH ! Die Initialisierung eines string-Objekts mit
einem char-Wert ist nicht erlaubt.
```

Sollen nicht alle sondern nur die Zeichen von x bis y kopiert werden, erfolgt dies durch die Methode „substr(…)“. Dabei wird als erster Parameter die Nummer des Anfangszeichens und als zweiter Parameter die Nummer des Endzeichens angegeben. Werden dabei zu große Werte angegeben, wird maximal bis zum letzten Zeichen kopiert. Die Klasse string liefert eine vorgefertigte Konstante „string :: npos“, die die Anzahl der Zeichen eines string-Objekts enthält.

```
string Name = “Wert Wert Wert“, Name2;

Name2 = Name1.substr(0, 3);
        // Kopiert die Zeichen von 0 bis 3

Name2 = Name1.substr(5, 7);
        // Kopiert die Zeichen von 5 bis 7

Name2 = Name1.substr(4,  string::npos);
        // Kopiert die Zeichen von 4 bis zum Ende des Strings

Name2 = Name1.substr(0,  0);
        // Kopiert ab dem 1.Zeichen nur ein Zeichen des Strings also nur “ “

Name2 = Name1.substr(0,  string::npos);
        // Kopiert alle Zeichen des Strings
```

Um Zeichen in einem String zu ersetzen gibt es die Methode “replace(…)”. Die Übergabeparameter sind wieder Anfangsnummer und Endnummer. Hinzu kommt jetzt der Wert oder der String, der eingesetzt werden soll.

```
string Name = “Wert Wert Wert“, Name2 = “Wert1“;

Name.replace(0, 3, “Hal“);
        // Ersetzt die Zeichen von 0 bis 3 durch “Hal“

Name.replace(0, 4, Name2);
        // Ersetzt die Zeichen von 0 bis 4 durch den längeren String Name2. Es
           steht nun in Name “Wert1 Wert Wert“
```

16.7. Verketten von Strings

Man kann string-Objekte auch verketten also zusammenhängen mit dem überladenen „+“ – Operator. Dabei ist es auch erlaubt char-Arrays, char-Zeiger sowie char-Werte zur Verkettung zu benutzen.

```
string Name1 = “Wert1“;
string Name2 = “Wert2“;
string Name3;

Name3 = Name1 + Name2;
```

```
char * Wort1 = "Wort1";
Name3 = Name1 + Wort1;

char Wort2 [5] = "Wort2";
Name3 = Name1 + Wort2;

Name3 = Name1 + "Wort3";

Name3 = Name1 + 'W';
```

Um nur eine bestimmte Anzahl von Zeichen eines string-Objekts anzuhängen gibt es die Methode „append(…)“. Als erstes Argument übergibt man den anzuhängenden Wert oder String. Als zweites Argument wird die Anzahl der zu kopierenden Zeichen x übergeben, das heißt, es werden die Zeichen 0 bis x angehängt. Übergibt man dieses Argument nicht, werden alle Zeichen kopiert.

```
string Name1 = "Wert1";
string Name2 = "Wert2";

Name1.append(", ");
        //Es wird ", " angehängt

Name1.append(Name2);
        //Es wird "Wert2" angehängt

Name1.append(Name2, 3);
        //Es werden nur 3 Zeichen angehängt also "Wer"
```

16.8. Vergleichen von Strings

String-Objekte kann man einfach mit dem „==“-Operator vergleichen. Es ist auch möglich char-Arrays, char-Zeiger aber nicht char-Werte beim Vergleich zu benutzen.

```
String Name1 = "Wert", Name2;
Name2 = Name1;

if (Name1 == Name2)
        cout<<"Strings sind gleich";

                // Vergleichsergebnis ist True

bool Vergleich;
        Vergleich = Name1 == Name2;

                // Die Bool'sche Variable enthält in diesem Fall den Wert True;

char * Wort = "Wert";
if (Name1 == Wort)
        cout<<"Strings sind gleich";

                // Vergleichsergebnis ist True
```

```
char Wort[5] = “Wert1“;
if (Name1 == Wort)
        cout<<“Strings sind gleich“;

                // Vergleichsergebnis ist False

Name1 = ’a’;
if ( Name1 == ’a’)
        cout<<“Strings sind gleich“;

                // Achtung FALSCH ! Vergleich von einzelnen char-Werten ist
                   nicht erlaubt
```

Es ist nicht nur möglich auf Gleichheit zu prüfen, sondern auch auf kleiner, kleiner gleich, größer und größer gleich. Dabei werden die Inhalte der beiden zu vergleichenden string-Objekte lexikographisch (bezüglich der Zeichensatznummern) verglichen.

Operationen für die zwei string-Objekte s1 und s2:

```
s1 > s2         // true, falls s1 lexikographisch größer als s2 ist
s1 > s2         // false, falls s1 nicht lexikographisch größer als s2 ist

s1 == s2        // true, falls s1 lexikographisch mit s2 übereinstimmt
s1 == s2        // false, falls s1 nicht lexikographisch mit s2 übereinstimmt

s1 < s2         // true, falls s1 lexikographisch kleiner als s2 ist
s1 < s2         // false, falls s1 nicht lexikographisch kleiner als s2 ist

s1 >= s2        // true, falls s1 lexikographisch größer als s2 ist oder mit s2
                   übereinstimmt
s1 >= s2        // false, falls s1 lexikographisch kleiner als s2 ist

s1 <= s2        // true, falls s1 lexikographisch kleiner als s2 ist oder mit s2
                   übereinstimmt
s1 <= s2        // false, falls s1 lexikographisch größer als s2 ist

s1 > s2         // true, falls s1 lexikographisch größer als s2 ist
s1 > s2         // false, falls s1 nicht lexikographisch größer als s2 ist
```

Das Vergleichen zweier string-Objekte ist auch über die Methode „compare(…)“ möglich. Als erstes Argument wird der zweite String übergeben. Als zweites und drittes Argument können optional die Nummer des Anfangszeichens und Endzeichens, die zu vergleichen sind, angegeben werden. Das Vergleichen von verschieden langen Strings ist möglich. Außerdem wird zwischen Groß- und Kleinschreibung unterschieden. Dabei gibt die Funktion folgende Werte zurück:

```
>0              //falls s1 lexikographisch größer als s2 ist
```

```
== 0            //falls s1 mit s2 übereinstimmt
<0              //falls s1 lexikographisch kleiner als s2 ist


string s1 = "Wert1";
string s2 = "wert1";

if  (s1.compare(s2) < 0)
        cout<<"s1 ist lexikographisch kleiner als s2";

                        // Vergleichsergebnis True

if  (s1.compare(s2, 1, 4) == 0)
        cout<<"s1 stimmt in den 1 bis 4 Zeichen mit s2 überein";

                        // Vergleichsergebnis True
```

16.9. Zeichen suchen in Strings

Um in einem String oder einem Teil eines Strings zu suchen verwendet man die Methode „find(…)“. Als erstes Argument wird der zu suchende Wert mit übergeben. Als zweites Argument wird die Position, an der zu suchen begonnen werden soll, mit übergeben. Wird dieses Argument nicht übergeben, wird automatisch bei Position 0 zu suchen begonnen. Außerdem benötigt man eine Variable vom Datentyp „string :: size_type“ die die Position des Zeichens speichert. Bei Nichtvorhandensein des Zeichens liefert die Funktion den Wert „size :: npos“ (no position) zurück.

Suche nach einem Zeichen von links beginnend:

```
string Name = „Wert";
string :: size_type Position;

Position = Name.find('e');
        // Suche nach dem Zeichen 'e' ab Position 0

Position = Name.find('e', 3);
        // Suche nach dem Zeichen 'e' ab Position 3
```

Suche nach einem Zeichen von rechts beginnend:

```
string Name = „Wert";
string :: size_type Position;

Position = Name.rfind('e');
        // Suche nach dem Zeichen 'e' ab Position Länge des Strings-1

Position = Name.rfind('e', 3);
        // Suche nach dem Zeichen 'e' von rechts her ab Position 3
```

Um nach einem von mehreren möglichen Zeichen, von links her beginnend, in einem String zu suchen gibt es die Methode „find_first_of(…)“. Dabei wird als

erstes Argument die zu suchenden Zeichen angegeben oder natürlich auch ein String. Als zweites Argument kann wieder die Position, an der zu suchen begonnen werden soll, angegeben werden. Bei Nichtvorhandensein des Zeichens liefert die Funktion den Wert „size :: npos“ (no position) zurück.

```
string Name = “Wert 1234“;
string Zahlen = “0123456789“;
string :: size_type Position;

Position = Name.find_first_of(Zahlen);
        // Position enthält die Position jenes Zeichens das als erstes mit einem
           der Zeichen vom String Zahlen übereinstimmt

Position = Name.find_first_of(„3456“, 2);
        // Es wird erst bei Position 2 zu suchen begonnen
```

Um nach einem Teilstring von links her beginnend zu suchen, wird wieder die Methode „find(…)“ benutzt. Um nach einem Teilstring, von rechts her beginnend, zu suchen, wird wieder die Methode „rfind(…)“ benutzt. Als Argument wird dabei ein ganzer String übergeben. Als zweites Argument kann wieder die Startposition angegeben werden. Bei Nichtvorhandensein des Strings liefert die Funktion den Wert „size :: npos“ (no position) zurück.

```
string Name = “Wert besteht aus einem Wort“;
string Wort = “aus“;
string :: size_type Position;

Position = Name.find(Wort);
        // Position enthält die Anfangsposition des Strings nach dem gesucht
           wurde

Position = Name.find (“besteht“, 3);
        // Es wird erst bei Position 3 zu suchen begonnen
```

16.10. Strings umwandeln in andere Datentypen

Man kann Strings durch verschiedene Methoden in char-Arrays oder C-Strings umwandeln.

Um eines Strings in einen C-String umzuwandeln, benötigt man einen C- String vom Typ „const char *“ und die Methode „c_str(void)“. Außerdem wird am Schluss die Binäre Null angehängt.

```
const char * Name;
string Wort = “Wert“;

Name = Wort.c_str();
```

Um einen Strings in ein char-Array umzuwandeln, benötigt man ein Feld vom Typ „const char *“ und die Methode „data(void)“. Es wird aber keine binäre Null in das Array geschrieben.

```
const char * Name [5];
string Wort = "Wert";

Name [5] = Wort.data();
```

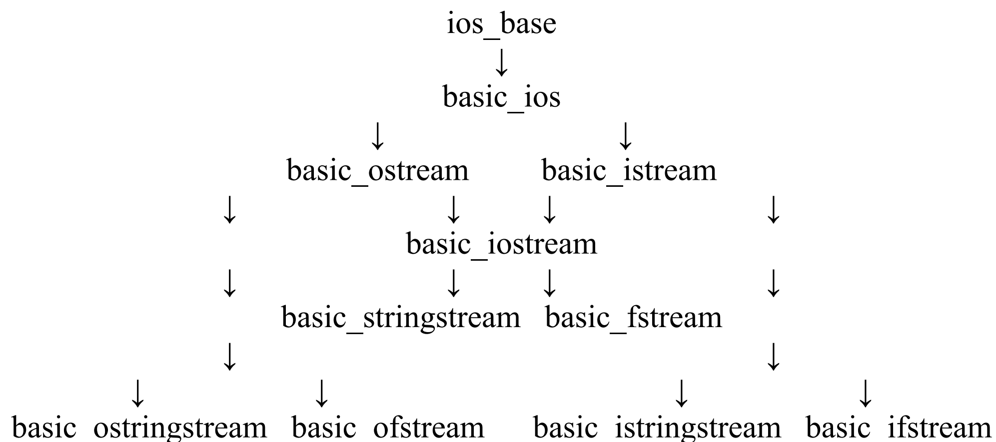
17. Ein- und Ausgabe

17.1. Grundlagen

Die Ein- und Ausgabe in C++ ist kein direkter Bestandteil der Programmiersprache selbst und wird durch entsprechende Klassenbibliotheken geregelt. Die Ein- und Ausgabe wird als Fluss (stream) eines Datenstroms oder Folge von Bytes aufgefasst. Das Byte ist die Dateneinheit des Stroms. Andere Datentypen wie int, struct, char*, … erhalten erst durch Bündelung und Interpretation von Bytesequenzen auf höherer Ebene ihre Bedeutung. Folgende Objekte sind im Header „iostream" als Klassen für Standardströme deklariert:

- Extern istream cin; // Standardeingabe
- Extern ostream cout; // Standardausgabe
- Extern ostream cerr; // Standardfehlerausgabe
- Extern ostream clog; // gepufferte Fehlerausgabe

17.2. Klassenhierarchie



17.3. Der Zeichensatz

C++ kennt zwei Arten von Zeichen: char (ASCII-Zeichen (Character)) und wchar_t (Multibyte-Zeichen (Wide Character). Multibyte-Zeichen sind die Zeichen des UNI-Codes (Universal-Code), der in Java und bei Internet-Browsern große Bedeutung gewonnen hat. Die ersten 256 Zeichen x0000 – x00FF, Latin1 bzw. BasicLatin, genannt, stimmen mit dem ASCII-Code überein. Insgesamt umfasst der UNI-Code 65536 (2^{16}) Zeichen von denen bisher 38885 festgelegt wurden. UNI-Code Zeichen werden durch ein großes „L" von den ASCII-Zeichen unterschieden. Ein Problem derzeit ist, dass der Zugriff auf diese Zeichensätze unter C++ noch nicht vollständig realisiert ist, da die meisten Rechner die asiatischen Zeichen nicht darstellen können. Dies ist bis jetzt nur in den asiatischen Windowsversionen und Linux integriert. Auch die meisten Internet-Browser sind nicht im Stande andere als europäische Sprachen darzustellen. Unter Windows ist es aber möglich bei der Installation asiatische Sprachenunterstützung mit zu installieren.

17.4. Ausgabe

Die abgeleitete Klasse „o_stream“ stellt den überladenen Operator „<<“ für jeden Standardtyp außer für void bereit.

```
basic_ostream & operator<< (const char*)
basic_ostream & operator<< (const int)
                    usw.
```

Jede Funktion des Operators „<<” liefert eine Referenz auf das Objekt basic_ostream zurück, für den er aufgerufen wurde, sodass auf ihn ein weiterer „<<“-Operator angewandt werden kann. In der Klasse ios_base sind eine Reihe von Formatierungsflags deklariert.

Der Zugriff auf die Flags erfolgt in dieser Form:

```
ios_base::Formatierungsflag
```

Zum Setzen der Flags dient die Funktion setf:

```
cout.setf(ios_base :: Formatierungsflag 1);
cout.setf(ios_base :: Formatierungsflag 1, ios_base :: Formatierungsflag 2);
```

Um einen Wert in einem bestimmten Zahlenformat auszugeben gibt es auch diese Möglichkeit:

```
cout<<dec<<Variablenname;
cout<<oct<<Variablenname;
cout<<hex<<Variablenname;
```

Für den ersten Übergabeparameter sind 19 Bitmasken definiert. Für den zweiten Übergabeparameter sind 3 Bitmasken definiert.

Formatierungsflags (Formatierungsflag 1):

boolalpha	Ein- und Ausgabe von bool als true und false
skipws	Übergehen von whitespace (Leerzeichen)
left	Ausrichtung linksbündig
right	Ausrichtung rechtsbündig
internal	Ausfüllen zwischen Vorzeichen und Zahl
dec	Dezimalsystem
oct	Oktalsystem
hex	Hexadezimalsystem
showbase	Präfix für Zahlensystem
showpoint	Dezimalpunkt und führende Nullen
showpos	„+“ für positive, ganze Zahlen
uppercase	große Hexziffer und Exponent „E“
scientific	Gleitkomma
fixed	Festkomma
adjustfield	Ausrichtung
basefield	Stellenwertsystem
floatfield	Fließkommaformat
unitbuf	Leerung des Ausgabepuffers nach cout

stdio Leerung des Ausgabepuffers nach Textzeichen

Parameter (Formatierungsflag 2):

adjustfield	left | right | internal
basefield	dec | oct | hex
floatfield	scientific | fixed

Da die Priorität der Verschiebungsoperatoren (<< , >>) sehr hoch ist, empfiehlt es sich, arithmetische und logische Ausdrücke im Ausgabestrom explizit zu klammern. Ansonsten kommt bei manchen Compilern eine Warnung, welche auf eine Zweideutigkeit hinweist. Für selbstdefinierte Datentypen wie Klassen muss der Ausgabeoperator vom Programmierer entsprechend überladen werden.

Funktionsdeklaration:

```
friend ostream & operator<< (ostream &, const Klassenname &);
```

Funktionsdefinition:

```
ostream & Klassenname :: operator<< (ostream &Name1, const Klassenname
&Name2)
{
        Name1<<Name2.Attribut;
        return Name1;
}
```

Außer der „setf“-Funktion existieren noch folgende Funktionen:

```
width()          // Ausgabebreite
preasion()       // Nachkommastellen
fill()           // für das Füllen mit Zeichen
```

Beispiel:

```
cout.width(5)          // Ausgabebreite von 5 Zeichen, der Rest
                          wird weggeschnitten
cout.preasion(3)       // 3 Nachkommastellen
cout.fill('c')         // füllt Zeile mit 'c'
```

Zur unformatierten Ausgabe existieren folgende Funktionen:

```
put()            // Schreiben eines einzelnen Zeichens
write()          // Schreibt n-Zeichen
flush()          // Ausgabe des Puffers
```

Beispiel:

```
cout.write('c', 4)            // gibt 4 'c' aus, also „cccc“
cout.write(„Beispiel“, 4)     // gibt „Beis“ aus
```

17.4.1. Ausgabe von Umlauten

Um ein Wort mit einem Umlaut auszugeben muss man folgendes tun:

```
// räumen
        cout<<“r“;
        cout.put(132);          // ASCII-Code
```

```
cout.flush();          // Gibt Puffer mit ASCII-Zeichen aus
cout<<"umen";
```

17.5. Eingabe

Zur Eingabe stellt die abgeleitete Klasse „basic_istream“ den überladenen Operator „>>“ für alle Standardtypen außer void bereit. Die Klasse ist ähnlich wie basic_ostream deklariert, es darf jedoch nicht das Referenzargument als const deklariert sein.

```
basic_istream & operator>> (bool &);
basic_istream & operator>> (int &);
```

Für selbstdefinierte Datentypen muss die Eingabe explizit programmiert werden.

Funktionsdeklaration:

```
friend istream & operator>> (istream &, Klassenname &);
```

Funktionsdefinition:

```
istream & Klassenname :: operator>> (istream &Name1, Klassenname
&Name2)
{
        Name1>>Name2.Attribut;
        return Name1;
}
```

Neben den Flags der Klasse ios_base verfügt die Klasse noch über andere Funktionen zum Zugriff auf Statusflags.

```
int eof() const                        // end of file (Programm wird mit
                                         „STRG+Z" beendet)
int clear() const                      // Löschen der Fehlerflags
int good() const                       // wahr bei Fehler, falsch bei EOF
bool fail() const                      // Wahr, wenn die Eingabe nicht geglückt
                                         ist.
iostate setstate (iostate Name) const  // setzt Status Name
iostate rdstate() const                // liefert den aktuellen Status
```

17.6. Dateibehandlungen

Die Ein- und Ausgabeoperationen sind auch bei Dateioperationen anwendbar. Die Funktionen get(), put(), open() und close() finden sich in der Headerdatei „fstream“. Diese Headerdatei enthält die Klassen „ifstream“, „ofstream“ und „fstream“. „ifstream“ bzw „ofstream“ sind parameterlose Konstruktoren, die einen nicht geöffneten Dateistrom erzeugen. Mit Parameterlisten ergibt sich folgendes Format:

```
ifstream Dateibezeichner (Quelldateiname, Parameter);
ofstream Dateibezeichner (Zieldateiname, Parameter);
```

Über den Dateibezeichner werden alle Dateioperationen ausgeführt (Lesen, Schreiben). Der Parameter hat einen der folgenden Aufzählungstypen aus der Klasse „basic_ios“ (bei VisualC++ „ios“):

- in // Lesen
- out // Schreiben
- binary // Binärdatei
- ate // Datei öffnen und Ende sochen
- app // Datei öffnen zum anhängen
- trunc // Datei beim Öffnen auf Länge 0 setzen
- nocreate // Datei öffnen, wenn existent
- noreplace // Datei öffnen, wenn nicht existent

Das Schließen einer geöffneten Datei erfolgt automatisch bei Programmende.

Neben den Flags der Klasse ios_base verfügt die Klasse noch über anderer Funktionen zum Zugriff auf Statusflags.

```
int eof() const                    // end of File (bei "STRG+Z" bricht das
                                      Programm ab)
int clear() const                  // Löschen der Fehlerflags
int good() const                   // wahr, wenn fail; eof falsch
bool fail() const                  // wahr, wenn Eingabe nicht geglückt
bool bad() const                   // wahr, wenn Eingabe gescheitert

iostate setstate(iostate s) const  // setzt Status s
iostate rdstate() const            // liefert den Status s
```

17.6.1. Binärdateien

Bei Binärdateien wird ein Strom von Bytes eingegeben bzw. ausgegeben. Diese Bytes werden nicht als Ganzzahlen oder Buchstaben interpretiert. Sie können nur mit einem Hexeditor gelesen werden. Das Schreiben in den stream führt die Funktion „write()“ aus und das Lesen die Funktion „read()“.

Schreiben:

```
Dateibezeichner.write((Datentyp *) &Variablenname,
            sizeof(Variablenname));
```

Lesen:

```
Dateibezeichner.read((Datentyp *)&Variablenname,
            sizeof(Variablenname));
```

17.6.2. Textdateien

Man unterscheidet beim Lesen zwischen sequentiellen und nicht sequentiellen Dateien. Textdateien haben den Vorteil mit jedem Editor lesbar zu sein. Sie sind jedoch nicht so kompakt wie Binärdateien, da sie einen Sepperator (Trennzeichen) zwischen den Elementen benötigen. Folgende Dateimodi kennt man bei Textdateien:

- in // nur Lesen
- out | trunc // nur Schreiben, existierende Dateien werden überschrieben, andere Dateien werden erzeugt
- out | app // nur Schreiben, anfügen an existierende Dateien, andere Dateien werden erzeugt

- in | out // Lesen und Schreiben, Datei muss schon existieren
- in | out | trunc // Lesen und Schreiben, existierende Dateien werden überschrieben, andere Dateien werden erzeugt
- in | out | app // Lesen und Schreiben, anfügen an existierende Dateien, andere Dateien werden erzeugt

Bei Textdateien wird zum Schreiben der Ausgabeoperator und zum Lesen der Eingabeoperator verwendet.

Schreiben:

```
Dateibezeichner<<x<<";";
```

Lesen:

```
Dateibezeichner>>x;
```

17.6.3. Beispiele

17.6.3.1. Kopieren einer Textdatei im Kommandozeilenmodus

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

using namespace std;

void error(char* string) {cerr<<string<<endl;}

int main (int argc, char* argv[])
{
        char ch;
        if(argc != 3)
        {
                error("Eingabeformat: Kopieren einer Textdatei datei1
                        datei2");
                exit(1);
        }

        ifstream quelle(argv[1]);
        if(!quelle)
        {
                error("Quelldatei kann nicht geoeffnet werden!");
        }
        ofstream ziel (argv[2]);
        if(!ziel)
        {
                error("Zieldatei kann nicht angelegt werden!");
        }
        while (quelle.get(ch))
        {
                ziel.put(ch);
        }
```

```
		if(!quelle.eof() || ziel.bad())
		{
			error("Fehler beim Dateikopieren aufgetreten!");
			return 1;
		}
		else
		{
			return 0;
		}
	}
```

17.6.3.2. Schreiben einer Binärdatei

```
#include<iostream>
#include<fstream>
#include<cstdlib>
#include<string>
#include<ctime>

using namespace std;

void startzufall()
{
	time_t now;
	srand((unsigned) time(&now));
}

int main ()
{
	string dateiname;
	cout<<"Welche Datei soll geschrieben werden: ";
	cin>>dateiname;

	ofstream zieldatei(dateiname.c_str(), ios::binary);
	if(!zieldatei)
	{
		cerr<<"Zieldatei kann nicht angelegt werden!" <<endl;
		exit(1);
	}
	int N;
	cout<<"Wie viele Wuerfelzahlen: ";
	cin>>N;
	startzufall();
	for(int i=0; i<N; i++)
	{
		int z= 1 + rand() % 6;
		zieldatei.write((char*)&z ,sizeof(z));
	}
	return 0;
}
```

17.6.3.3. Lesen einer Binärdatei

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

using namespace std;


int main ()
{
	string dateiname;
	cout<<"Welche Datei soll gelesen werden: ";
	cin>>dateiname;

	ifstream quelldatei(dateiname.c_str(), ios::nocreate);
	if(!quelldatei)
	{
		cerr<<"Quelldatei kann nicht geöffnet werden!" <<endl;
		exit(1);
	}
	int z, i = 0;
	while(quelldatei.read((char*)&z, sizeof(z)))
	{
		i++;
		cout<<z<<" ";
		if(i%20==0)
		{
			cout<<endl;
		}
	}
	cout<<"Es wurden " << i << " Zahlen eingelesen"<<endl;
	return 0;
}
```

17.6.3.4. Schreiben einer Textdatei

```
#include<iostream>
#include<fstream>
#include<cstdlib>
#include<string>
#include<ctime>

using namespace std;

void startzufall()
{
	time_t now;
	srand((unsigned) time(&now));
}

int main ()
{
```

```
        string dateiname;
        cout<<"Welche Datei soll geschrieben werden: ";
        cin>>dateiname;

        ofstream zieldatei(dateiname.c_str(), ios::out);
        if(!zieldatei)
        {
                cerr<<"Zieldatei kann nicht angelegt werden!" <<endl;
                exit(1);
        }
        int N;
        cout<<"Wie viele Wuerfelzahlen: ";
        cin>>N;
        startzufall();
        for(int i=0; i<N; i++)
        {
                int z= 1 + rand() % 6;
                zieldatei<<z<<"\n";
        }
        return 0;
}
```

17.6.3.5. Lesen einer Textdatei

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

using namespace std;

int main ()
{
        string dateiname;
        cout<<"Welche Datei soll gelesen werden: ";
        cin>>dateiname;

        ifstream quelldatei(dateiname.c_str(),ios::in | ios::nocreate);
        if(!quelldatei)
        {
                cerr<<"Quelldatei kann nicht geoeffnet werden!" <<endl;
                exit(1);
        }

        int z,i=0;
        while(quelldatei >> z)
        {
                i++;
                cout<<z<<" ";
                if(i%20==0)
                {
                        cout<<endl;
                }
```

```
        }
        cout<<“Es wurden “ << i <<“Zahlen eingelesen “<<endl;
        return 0;
}
```

18. Klassenbeziehungen (Relationen)

Klassen und Objekte existieren nicht isoliert. Sie können auf verschiedene Arten miteinander in Beziehung stehen. Man unterscheidet folgende Beziehungen:

18.1. Assoziation (kennt)

Eine Assoziation beschreibt eine Verbindung von einem Objekt zu einem oder mehreren anderen. Gemeinsamkeiten zwischen den Objekten können vorhanden sein, spielen aber keine Rolle. Solche Assoziationen können sein: eine Person arbeitet für ein Unternehmen, ein Unternehmen hat mehrere Filialen, ein Kunde erteilt einen Auftrag. Eine spezielle Art der Assoziation ist durch die sogenannte Kordinalität gegeben. Diese gibt die Anzahl der in Relation stehenden Objekte an. Vereinfacht ausgedrückt kann man sagen: Alle Objekte einer Klasse, welche im public-Bereich einer anderen Klasse vorkommen, bilden eine Assoziation.

18.2. Aggregation (hat, existenzunabhängig)

Unter Aggregation versteht man die Beziehung bei der das Ganze zu jedem Teil in einer „hat“ (has a) – Beziehung steht. Die einzelnen Teile stehen dann in einer „ist Teil von“ (is part of) – Beziehung zum Ganzen. Sie besagt, dass ein Objekt aus mehreren Teilen besteht, die wiederum aus Teilen bestehen können. Eine solche Aggregation kann sein: Eine Tastatur besteht aus einem Gehäuse, 102 Tasten, einem Kabel, usw. Vereinfacht ausgedrückt handelt, es sich dabei um einzelne Komponenten die auch einzeln für sich existieren können.

18.3. Komposition (hat, existenzabhängig)

Unter Komposition versteht man das Gleiche wie unter Aggregation, mit dem einen Unterschied, dass die einzelnen Teile nur im Ganzen einen Sinn ergeben. Es ist ein Zusammenwirken einzelner Komponenten, die nur als Einheit existieren können. Unter Aggregation und Komposition kann man sich vereinfacht vorstellen: Alle Objekte einer Klasse, welche im private-Bereich einer anderen Klasse vorkommen bilden eine Komposition und Aggregation.

18.4. Benutzt (uses a)

Bei der Benutzt-Beziehung verwendet eine Elementfunktion (Methode) einer Klasse ein Objekt einer anderen Klasse als Parameter. Die zweite Klasse ist jedoch nicht Teil der ersten Klasse. Vereinfacht ausgedrückt kann man sagen: Kommt ein Objekt einer Klasse in einer Methode einer anderen Klasse als lokaler Parameter vor, spricht man von der Benutzt-Beziehung.

18.5. Vererbung, Generalisierung (inheritance, ist)

Bei der Vererbung liegt eine „ist“ oder eine „ist eine Art von“ – Beziehung vor. Ein Objekt einer abgeleiteten Klasse kann damit stets an die Stelle eines Objekts der Basisklasse treten, da ja alle Methoden und Attribute der Oberklasse vorhanden sind, wenn auch möglicherweise überschrieben.

18.6. Darstellung der Klassenbeziehungen

Meist werden die Beziehungen in UML (Unified Modeling Language) dargestellt, welche dieselbe Funktion wie Struktogramme in C hat. Die UML ist ein von einem internationalen Gremium (OMG =Object Managment Group) standardisiertes System, welches die besten Analyse- und Entwurfsmethoden der Softwareentwicklung zusammenfasst. Dabei unterscheidet man folgende wichtige Methoden:

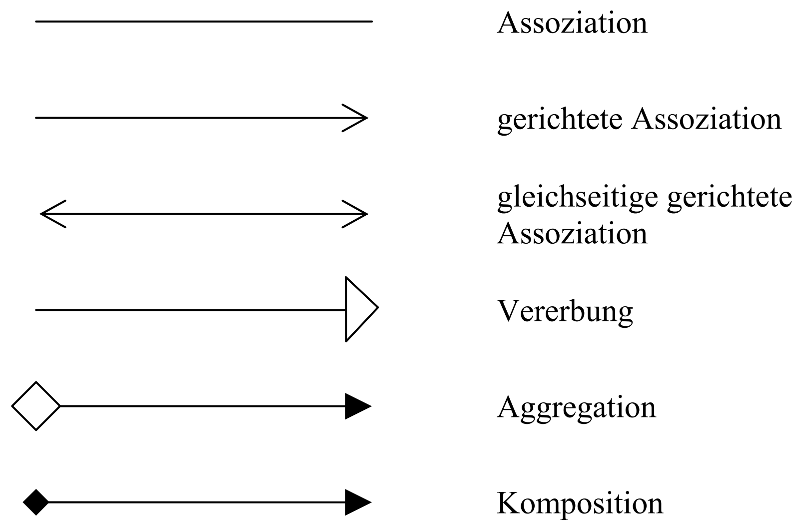
- Statische Klassen- / Objektdiagramme
- Interaktionsdiagramme (Sequenz- und Kollaberationsdiagramme)
- Aktivitätsdiagramme
- Anwendungsfalldiagramme (Use-Cases-Diagramme)

18.6.1. Statisches Klassendiagramm

Titel der Klasse
Private und Protectet-Attribute und Methoden - private-Attribute werden mit einem „–“ gekennzeichnet # protected-Attribute werden mit einem“#“ oder „°“ gekennzeichnet
Public-Attribute und Methoden + public-Attribute werden mit einem „+“ gekennzeichnet

18.6.2. Statisches Objektdiagramm

Die Objekte werden als Rechtecke dargestellt und durch Pfeile verbunden:



Zusatzangaben:

- Bedingungen
- Rollennamen
- Kordinalität
  Wird am Ende des Pfeiles angegeben.

1	genau 1 Objekt
*	viele Objekte, d.h. Null oder mehr
0 … 1	höchstens 1 Objekt
m … n	mindestens m aber höchstens n Objekte

19. Vererbung

19.1. Grundlagen

Die Vererbung ist ein aus der Biologie übernommener Begriff, der besagt, dass ein Nachkomme die genetischen Eigenschaften seiner Vorfahren erhält. Dieser Begriff wurde von der objektorientierten Programmierung übernommen für den Vorgang, dass eine abgeleitete Klasse Datenelemente und Elementmethoden von einer Basisklasse übernehmen kann. Die Vererbung ist von prinzipieller Bedeutung für die Wiederverwendbarkeit von Programmcodes, da wichtige Eigenschaften von Klassen abgeleitet werden können. Diese Vererbung kann fortgesetzt werden, sodass eine ganze Hierarchie von abgeleiteten Klassen entsteht. Diese können spezialisierte und zusätzliche Methoden gegenüber ihren vererbenden Klassen haben. Die gemeinsame Basisklasse enthält den Durchschnitt aller Klassen, d.h. die Eigenschaften, die allen abgeleiteten Klassen gemeinsam sind. Leitet sich jede Klasse direkt von genau einer vererbenden Klasse ab, so spricht man von einfacher Vererbung, sonst von mehrfacher Vererbung. Die Vererbung erfüllt zwei Zwecke in einer objektorientierten Anwendung: Spezialisierung und Generalisierung.

19.1.1. Spezialisierung

Unter Spezialisierung versteht man, dass ein Teil der bestehenden Funktionalität einer Klasse in die abgeleitete Klasse übernommen wird.

19.1.2. Generalisierung

Unter Generalisierung versteht man, dass bestehende Methoden der Basisklasse in der abgeleiteten Klasse mit weiteren Eigenschaften überladen werden.

19.1.3. Zugriffsspezifizierer

Im Sinne der Datenkapselung hat eine abgeleitete Klasse prinzipiell keinen Zugriff auf private-Elemente der Basisklasse. Zugang besteht nur für alle public- und protected-Elemente. Für die Basisklasse gilt: private entspricht gleich protected. Für die abgeleitete Klasse gilt: protected entspricht gleich public. Auf public-Datenelemente können sowohl Element- als auch Nicht-Element-Methoden beider Klassen zugreifen. Protected-Elemente sind nur für Elementmethoden der Basis- und abgeleiteten Klasse sichtbar. Die Art der Ableitung wird durch einen Zugriffsspezifizierer gekennzeichnet. Fehlt dieser, wird automatisch public als Zugriffspezifizierer angenommen.

```
class Name1 : Zugriffsspezifizierer Name2
{
};
```

19.1.3.1. Spezifizierer der Basisklasse

Art der Ableitung	**public**	**protected**	**private**
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Bei einer public-Ableitung bleibt der Status der Datenelemente erhalten.

19.1.4. Konstruktor und Destruktor

Benötigt man in einer abgeleiteten Klasse einen Konstruktor, wird, wenn die Basisklasse einen besitzt, der der Basisklasse verwendet. Zuerst wird der Konstruktor der Basisklasse, dann der der abgeleiteten Klasse aufgerufen. Bei den Destruktoren ist es genau umgekehrt. Hat die Basisklasse mehr Konstruktoren, so wird einer der Basisklasse anhand der Signatur ausgewählt. Über den Konstruktor der abgeleiteten Klasse wird mittels Doppelpunkt der Konstruktor der Basisklasse aufgerufen. Dabei werden die Argumente übergeben, die der Konstruktor der abgeleiteten Klasse einliest.

Beispiel:

```
Konstruktor_AbgeleiteteKlasse (Datentyp Parameter_BasisKlasse,
Datentyp Parmeter2_BasisKlasse, Datentyp
Parameter_AbgeleiteteKlasse) : Basisklassenname
(Parameter_BasisKlasse, Parmeter2_BasisKlasse)
{
        …
}
```

19.1.5. Polymorphismus

Polymorphismus bedeutet Vielgestaltigkeit. Man unterscheidet zwischen zwei Formen des Polymorphismus:

- Änderung der Parameter
  (Templates, Überladen von Funktionen)
- Änderung der Methoden
  (virtuelle Methoden beim Vererben, Überladen von Operatoren)

19.1.5.1. Statisches bzw. Frühes Binden (early binding)

Man unterscheidet 2 Formen des Linkens oder Bindens. Das Frühe Binden ist eine davon. Beim Frühen Binden kann bereits beim Kompelieren festgelegt werden, welche Funktionen aufgerufen werden.

19.1.5.2. Dynamisches bzw. Spätes Binden (late binding)

Beim Späten Binden kann erst zur Laufzeit bestimmt werden welche Funktionen aufgerufen werden. Diese Art des Bindens gibt es nur bei echten objektorientierten Programmiersprachen.

19.2. Mehrfachvererbung

Eine Klasse kann nicht nur aus einer Basisklasse sondern auch aus mehreren Basisklassen abgeleitet werden. Dabei werden die Zugriffspezifizierer einfach durch einen Beistrich getrennt.

```
class A: public B, private C
{
        …
};
```

19.3. Methoden redefinieren (Überschreiben von Methoden)

Ein ähnlicher Mechanismus wie das Vererben ist das Redefinieren (Überschreiben) von gleichnamigen Methoden mit unterschiedlichen Schnittstellen. Dabei wird die Methode der Basisklasse in den abgeleiteten Klassen neu ausprogrammiert.

19.4. Virtuelle Methoden

Virtuelle Methoden sind spezielle Elementfunktionen, die nicht zur Übersetzungszeit, sondern erst zur Laufzeit gebunden werden. Das heißt, erst beim Aufruf einer virtuellen Methode wird entschieden, welche Funktion tatsächlich ausgeführt wird. Eine Methode muss dann in der Basisklasse als virtuell deklariert werden, wenn diese mit gleichem Namen in den abgeleiteten Klassen vorkommt. Würde man dies nicht tun, wird beim Aufruf immer nur die Methode der Basisklasse aufgerufen, da diese vom Compiler statisch gebunden wurde. Wird diese jedoch als virtuell deklariert, wird sie erst zu Laufzeit gebunden und der Compiler kann die richtige Methode zur jeweiligen Klasse aufrufen.

```
virtual Rückgabedatentyp Methodenname(Übergabeparameter)
{
        …
}
```

19.5. Abstrakte Basisklassen

Die Einführung von virtuellen Methoden ermöglicht die Deklaration von einer Methode ohne Funktionsrumpf. Diese Methoden dienen nur zum Überladen (Redefinieren) und werden rein virtuell deklariert. Eine Klasse, die eine solche Methode enthält, nennt man Abstrakt. Von einer solchen Klasse kann keine Instanz (=Objekt) gebildet werden.

19.6. Virtuelle Basisklasse

Bei der Mehrfachvererbung kann eine Klasse mehrmals als indirekte Basisklasse auftreten. Dadurch enthalten Klassen die von anderen Klassen, welche zuvor von der Basisklasse geerbt haben, die Attribute und Methoden der Basisklasse doppelt oder sogar öfter. Um dies zu verhindern, müssen jene Klassen die direkt von der Basisklasse erben als „virtual" deklariert werden.

```
class Name: Zugriffsspezifizierer virtual Basisklasse
{
        …
}
```

Außerdem ist zu beachten, dass die Konstruktoren von virtuellen Basisklassen anders aufgerufen werden als jene von normalen Basisklassen. Sie können nur von jenen Klassenkonstruktoren aufgerufen werden, die ihre Klasse auch als direkte Basisklasse benutzen. Bei allen anderen Klassen werden sie vor allen anderen Klassenkonstruktoren aufgerufen und zwar nur einmal.

20. Template-Funktionen

C++ bietet die Möglichkeit Template-Funktionen zu definieren. Diese Funktionen werden auch „generisch" genannt. Diese generischen Funktion verwendet man dann, wenn man eine Funktion benötigt, aber im vorhinein den Datentyp der Signatur (Rückgabedatentyp,

Datentyp der Übergabeparameter) noch nicht kennt. Dabei wird bei der Deklaration „template <class T>“ der Funktion vorangestellt, danach wird die Funktion ausprogrammiert, anstatt des Datentyps der Signatur wird hier aber der formale Parameter „T“ eingefügt.

Beispiel:

```
template <class T>
T Funktionsname (T Parametername1, T Parametername2)
{
        return(Parametername1 + Parametername2);
}

void main
{
        int i =13;
        double d=2.4;
        string s ='p';

        cout<<Funktionsname(i, 2);
        cout<<Funktionsname(d, 6.);
        cout<<Funktionsname(s, 'y');
}
```

Benötigt man mehrere formale Parameter, gibt man diese bei der Deklaration mit dem Schlüsselwort „template“ an.

```
template<class T,class S, …>
```

21. Template-Klassen

Template-Klassen werden dann eingesetzt, wenn eine Klasse mit verschiedenen Datentypen verwendet werden soll. Damit stellen Template-Klassen gemeinsam mit den Template-Funktionen ein wichtiges Element der Wiederverwendbarkeit dar. Dies gilt vor allem für die Standard-Template-Library (STL). Diese enthält die wichtigsten Datenstrukturen und grundlegende Algorithmen in Template-Form. Keine Daten für Templates sind jene Klassen, die als Behälter für Datentypen fungieren und daher als Container-Klassen bezeichnet werden. Eine Template-Klasse wird spezifiziert durch das Schlüsselwort „template“.

```
template <class T>

class A
{
        T* a;
};
```

Der in Klasse A auftretende Datentyp muss spezifiziert werden:

```
A <int> a;              // Template-Attribut wird mit int belegt
```

Nicht Inline-Methoden sind wie folgt zu definieren:

```
template <class T>
void A<T>  :: Methodenname (T a) { … }
```

Ein Konstruktor hat folgende Form :

```
template <class T>
 A<T> :: A (T a) { … }
```

22. Fehlerbehandlung

Es werden folgende Fehlerarten unterschieden: Compilerfehler, Linkerfehler, logische Fehler und Laufzeitfehler. Oft tritt ein Fehler in einer Funktion auf, der innerhalb der Funktion selbst nicht behoben werden kann. Der Aufrufer einer Funktion muss Kenntnis von aufgetretenen Fehlern bekommen, damit er den Fehler abfangen oder noch weiter nach oben melden kann. Ein Programmabbruch in jedem Fall ist benutzerunfreundlich und nicht immer nötig. Eine generelle Fehlerbehebung ist nicht möglich, jeder Einzelfall ist gesondert zu überlegen. Mit den bisher behandelten Mitteln lassen sich verschiedene Strategien zu Fehlerbehebung realisieren. In der nachfolgenden Aufzählung wird angenommen, dass der Fehler in einer Funktion entstanden ist.

1) Das programmtechnisch einfachste ist der sofortige Programmabbruch innerhalb der Funktion, die einen Fehler verursacht. Wenn keine erläuterten Meldungen über den Abbruch ausgegeben werden ist dadurch die Fehlerdiagnose, erschwert bzw. unmöglich.
2) Ein üblicher Mechanismus zur Fehlerbehandlung ist die Übergabe eines Parameters an den Aufrufer einer Funktion, der Auskunft über Erfolg oder Misserfolg gibt. Diese hat den Parameter auszuwerten und entsprechend zu reagieren. Der Parameter ist nach jedem Funktionsaufruf auszuwerten.

   ```
   Ergebnis = funktion(par1, par2, &fehler);

   if(fehler)
   {
       switch(fehler)
       {
               case 1: ergebnis = -1; break;
               case 2: ergebnis = 0; break;
               default: cout<<“Nicht behebbarer Fehler!“<<endl; exit(-2);
       }
   }
   ```

   Der Vorteil dieser Methode liegt in der selektiven Art und Weise wie der Aufrufer einer Funktion Fehler an der Stelle des Auftretens behandeln kann, sodass sogar ein Abbruch nicht nötig ist. Der Nachteil ist, dass der Programmcode schwerfällig wirkt und schwer leserlich ist, auf Grund der vielen Fehlerprüfungen.
3) Eine Funktion kann im Fehlerfall die in der C-Welt übliche globale Variable „errno“ setzen, die dann wie im vorherigen Fall abgefragt wird. Globale Variablen sind nicht gut geeignet, weil sie die Portabilität von Funktionen beeinträchtigen und weil beim Zusammenwirken mehrerer Programmteile Werte der Variablen möglicherweise nicht mehr eindeutig sind.

4) Eine Funktion, die einen Fehler feststellt, kann eine andere Funktion zur Fehlermeldung und Fehlerbehandlung aufrufen. Diese Funktion kann auch den Programmabbruch herbeiführen. Bei dieser Methode muss überlegt werden, ob dem Aufrufer die Information des Fehlschlages mitgeteilt werden muss. Dies gilt sicher dann, wenn die Funktion die ihr zugedachte Arbeit nicht erledigen kann.
5) Die noch immer anzutreffende Methode ist die, dem Aufrufer trotz eines Fehlers einen gültigen Wert zurückzugeben und nichts zu tun. Dadurch können aber Folgefehler entstehen. Ein Beispiel ist, dass man einen Indexoperator eines Feldes so programmiert, dass bei einer Indexüberschreitung immer der Wert 0, ohne Meldung, zurückgegeben wird. Dies ist besonders tückisch, weil Fehler sich an einer ganz anderen Stelle und sehr viel später bemerkbar machen und schwer zu finden sind.

Die Art der Fehlermeldung hängt ganz von der Anwendung ab und wo diese eingesetzt wird. Unter den oben genannten Methoden sind die 2te und die 4te akzeptabel. C++ stellt zusätzlich die Ausnahmebehandlung bereit, mit der Fehler an spezielle Funktionen des aufrufenden Kontexts übergeben werden.

23. Ausnahmebehandlung

Die 4-te Fehlerbehandlungsstrategie, aus dem vorherigen Punkt, hat den Vorteil, dass der Programmcode, der die eigentliche Aufgabe erledigen soll, von vielen eingestreuten Fehlerprüfungen entlastet wird. In C++ bietet dies die Ausnahmebehandlung (=exeptionhandling) an, welche ebenfalls den Fehlerbehandlungscode vom Programm sauber trennt. Zu unterscheiden ist zwischen der Kennung von Fehlern, wie z.B. Division durch 0, Bereichsüberschreitung eines Arrays, Syntaxfehler bei Eingaben, Zugriff auf eine nicht geöffnete Datei, Fehlschlag bei der Speicherbeschaffung und Nichteinhaltung der Vorbedingungen einer Methode, und der Fehlerbehandlung. Die Fehlererkennung ist in der Regel einfach, die Behandlung schwierig und häufig unmöglich.

Ablauf der Fehlerbehandlung:
1) Eine Funktion versucht die Erledigung einer Aufgabe. (englisch = try)
2) Wenn die Funktion einen Fehler feststellt, den sie nicht beheben kann, wirft sie eine Ausnahme. (englisch = throw)
3) Die Ausnahme wird von einer Fehlerbehandlungsroutine aufgefangen, die den Fehler bearbeitet. (englisch = catch)

Die Funktion kann der Fehlerbehandlungsroutine ein Objekt eines beliebigen Datentyps zuwerfen um Informationen zu übergeben. Im Unterschied zu der im vorherigen Punkt beschriebenen 4-te Strategie wird nach der Fehlerbehandlung nicht in die Funktion zurückgesprungen. Das Programm wird vielmehr mit dem der Fehlerbehandlung folgenden Code fortgesetzt. Wenn aus einem Block herausgesprungen wird, werden die Destruktoren aller in diesem Block automatisch definierten Objkete aufgerufen.

Syntaktisches Schema (Ablauf):

```
try
{
        func();
}
```

// Falls die Funktion „func()“ einen Fehler entdeckt, wirft sie eine Ausnahme aus, wobei ein Objekt übergeben werden kann, um die Fehlerbehandlung anzustoßen.

```
catch (Datentyp 1)
{
        …
}
```

// Durch ausgeworfenes Objekt des Datentyps 1 ausgewählte Fehlerbehandlung

```
catch (Datentyp 2)
{
        …
}
```

// Durch ausgeworfenes Objekt des Datentyps 2 ausgewählte Fehlerbehandlung

```
catch(…)
{
        …
}
```

// Die sogenannte Ellipse behandelt einen nicht spezifizierten Fehler

23.1. Exeptionspezifikation und Deklaration

Um Ausnahmen, die eine Funktion auswerfen kann, in den Schnittstellen bekannt zu machen, können sie im Funktionsprototyp als sogenannte Exeptionspezifikation deklariert werden. Dabei unterscheidet man 3 Fälle:

1) void Funktionsname();
   Es können beliebige Ausnahmen geworfen werden.
2) void Funktionsname() throw();
   Verspricht keine Ausnahmen auszuwerfen.
3) void Funktionsname() throw(DateiEnde, const char*);
   Verspricht nur die deklarierten Ausnahmen auszuwerfen.

Der Benutzer einer Funktion kennt zwar ihren Prototyp, im Allgemeinen aber, nicht die Implementierung der Funktion. Die Angabe der möglichen Ausnahmen im Prototyp is daher sinnvoll, damit der Benutzer der Funktion sich darauf einstellen und geeignete Maßnahmen treffen kann.

24. Exception-Hierarchie in C++

Bisher haben wir immer eigene Klassen definiert um Ausnahmen aufzufangen. In C++ gibt es eine Reihe von vordefinierten Klassen zur Ausnahmebehandlung. Dabei erben alle speziellen Exception-Klassen von der Basisklasse „Exeption“.

```
exeption   → logic_error          → invalid_argument
                                  → length_error
                                  → out_of_range
                                  → domain_error
```

→ runtime_error
- → overflow_error
- → underflow_error
- → bad_alloc
- → bad_typeid
- → bad_cast
- → bad exeption

In der Klasse „Exeption“ ist die Methode „what()“ implementiert, welche einen Zeiger auf char zurückgibt. Diese Zeichenkettenkonstante verweist auf eine Fehlermeldung. Eigene Klassen sollen in diese Vererbungsstruktur eingebunden werden. Dadurch können sie vorgefertigte Eigenschaften und Methoden übernehmen.

24.1. Erkennen logischer Fehler

Die Frage, die sich einem Entwickler stellt ist: „Wie können während der Entwicklung logische Fehler erkannt werden?“ Dies ist möglich in dem man in das Programm eine Zusicherung einer logischen Bedingung einbaut (assertion). In C und C++ dient dazu das Makro „assert()“, das man in der Bibliothek „assert.h“ findet. 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 Überprüfungen mit „assert()“ werden abgebrochen, wenn das Programm die Präprozessor diriktive „#define NDEBUG“ findet.

Beispiel: assert (x>5);

24.2. Durch Exeption verursachte Speicherfehler vermeiden

Um Speicherfehler bei der Exception-Behandlung zu vermeiden, sollten dabei nur Stack-Objekte (automatische Objekte) verwendet werden und keine Heap-Objekte (dynamische Objekte).

24.3. Speicherbeschaffung mit „new“

In C wurde bei der Speicherbeschaffung der NULL-Zeiger abgefragt. In C++ wirft die Methode „new“ eine Ausnahme, vom Typ „bad_alloc“, wenn etwas schief geht.

Beispiel:

```
try
{
        int* vielSpeicher = new int [30 000];
}

catch (bad_alloc)
{
        cerr<<“Kein Speicher vorhanden !“<<endl;
        exit(1);
}
```

25. Dynamische Datenstrukturen

Dynamische Datenstrukturen ändern ihre Struktur und den von ihnen belegten Speicherplatz während der Programmausführung. Sie sind aus einzelnen Elementen, den

einzelnen Knoten, aufgebaut, zwischen denen üblicherweise eine gewisse Nachbarschaftsbeziehung (Verweise) besteht. Die Dynamik liegt beim Einfügen neuer Knoten, Entfernen vorhandener Knoten und in der Änderung der Nachbarschaftsbeziehungen. Der Speicherplatz für einen Knoten wird erst bei Bedarf zur Programmlaufzeit allokiert. Die Beziehung zwischen den einzelnen Knoten wird über Zeiger hergestellt, die jeweils auf den Speicherort des logischen Nachbarns zeigen. Jeder Knoten wird daher neben den zu speichernden Nutzdaten mindestens einen Zeiger auf die jeweiligen Nachbarknoten enthalten. Man nennt solche Strukturen daher auch verkettete oder rekursive Datenstrukturen. Die wichtigsten dieser Datenstrukturen sind:

- Lineare Liste
  - Einfachverkettete Liste
  - Doppeltverkettete Liste
  - Ring (Ringspeicher)
  - Queue(Pufferspeicher, First in First out)
  - Stack (Kellerspeicher, Last in First out)
- Bäume
  - Binärbaum (zwei Nachfolger)
  - Vielwegbaum (viele Nachfolger)
- Allgemeine Graphen

Die wichtigsten Operationen mit dynamischen Datenstrukturen sind:

- Erzeugen eines neuen Elementes
- Einfügen eines Elementes
- Entfernen eines Elementes

25.1. Einfachverkettete Liste

Bei der einfachverketteten Liste enthält jedes Element einen Datensatz und einen Zeiger auf das nächste Element in der Liste. Der Zeiger des ersten Elementes ist der root-Zeiger von dem aus jede Operation in der Liste begonnen wird. Der Zeiger des letzten Elements zeigt immer auf NULL, so kann das Ende der Liste durch Abfrage auf NULL in einer Schleife gefunden werden.

Syntax:

```
struct Listenname
{
        // Daten der Liste;

        Listenname * next;
};
```

Über den Befehl „typedef“ kann man der Liste einen Aliasnamen geben.

```
typedef struct Listenname
{
        // Daten der Liste

        Listenname * next;
}Neuer Name der Liste;
```

25.2. Doppeltverkettete Liste

Bei der doppeltverketteten Liste enthält jedes Element einen Datensatz und einen Zeiger auf das nächste Element, sowie einen Zeiger auf das vorherige Element in der Liste. Der Next-Zeiger des ersten Elementes ist der root-Zeiger, von dem aus jede Operation in der Liste begonnen wird. Der Prev-Zeiger des ersten Elements zeigt auf NULL; Der Next-Zeiger des letzten Elements zeigt immer auf NULL, so kann das Ende der Liste durch Abfrage auf NULL in einer Schleife gefunden werden.

Syntax:

```
struct Listenname
{
        // Daten der Liste

        Listenname * next;
        Listenname * prev;
};
```

25.3. Binärer Baum

Jedes Element eines binären Baumes enthält einen Datensatz sowie einen Zeiger auf das links darunterliegende Element und einen auf das rechts darunterliegende Element. Der root-Zeiger zeigt auf das oberste Element, von diesem ausgehend werden alle Operationen innerhalb des Baumes gestartet. Jeder Zeiger, der auf kein Element zeigt, wird auf NULL gesetzt.

Syntax:

```
struct Baum
{
        // Daten der Liste

        Baum * left;
        Baum * right;
};
```

25.4. Bubblesort für eine lineare Liste

```
LinListe* Bubblesort (LinListe * root)
{
    LinListe * Rest, * Akt;
    char Temp [10];
    Akt = root;

    if((root==NULL) || (root-> next==NULL)) return 0;

    do
    {
            Rest=Akt;
            do
            {
```

```
                Rest=Rest->next;
                if(strcmp(Rest->Inhalt, Akt->Inhalt)<0)
                {
                        strcpy(Temp, Rest->Inhalt);
                        strcpy(Rest->Inhalt, Akt->Inhalt);
                        strcpy(Akt->Inhalt, Temp);
                }
            }
            while(Rest->next != NULL);
            Akt=Akt->next;
        }
        while(Akt->next != NULL)
        return 0;
    }
```

25.5. Stack, Stapel, Keller

Ein Stack ist eine besondere lineare Liste, weil die Daten eine Folge bilden, die nur durch die Ein- und Ausgabefunktionen bestimmt wird. Das Prinzip eines Kellers besteht darin, dass stets das zuletzt eingefügte Element als erstes entfernt werden muss. (LiFo = Last in First out)

Die Standardfunktionen sind:

- Init() // Initialisiert einen neuen Stack
- Push() // Legt ein Element an
- Pop() // Holt ein Element
- Empty() // Prüft, ob der Stack leer ist
- Top() // Auslesen des ersten Elementes
- Bottom() // Auslesen des letzten Elementes

25.6. Queue

Ein Queue ist eine besondere lineare Liste, weil die Daten eine Folge bilden, die nur durch die Ein- und Ausgabefunktionen bestimmt wird. Das Prinzip eines Kellers besteht darin, dass immer das erste Element entfernt werden muss. (FiFo = First in First out)

Die Standardfunktionen sind:

- Init() // Initialisiert einen neuen Queue
- Push() // Legt ein Element an
- Pop() // Holt ein Element
- Empty() // Prüft, ob der Queue leer ist
- Top() // Auslesen des ersten Elementes
- Bottom() // Auslesen des letzten Elementes

26. Baumstruktur

Baumstrukturen spielen in der Datenorganisation eine wichtige Rolle. Auch im Alltag sind diese Strukturen allgegenwärtig (z.B.: Organisationsstruktur eines Unternehmens, bei der Einteilung von Texten in Kapitel, Abschnitte, Absätze). Ein Baum ist folgendermaßen definiert:

- Baumstrukturen haben Verzweigungen, die vom Ursprung-/Anfangselement und von Verzweigungen (Teilbäumen) selbst ausgehen können, bis schließlich Endpunkte erreicht werden.
- Die Elemente des Baumes heißen Knoten und das Anfangselement nennt man Wurzel (=root).
- Die Endpunkte heißen Blätter (=leaves).
- Die Wurzel ist der einzige Knoten der keinen Vorgänger hat.
- Knoten, die weder Wurzel noch Blatt sind, heißen Innere Knoten.
- Die Verweislinie zwischen zwei Knoten nennt man Kante.
- Die Folge der Kanten von der Wurzel zu einem beliebigen Knoten heißt Pfad.
- Die maximale Anzahl der direkten Nachfolger, die ein Knoten hat, heißt Grad des Baumes.

Beim Aufbau einer verketteten Liste können die einzelnen Listenelemente gleich nach einem gewissen Ordnungskriterium hintereinander gehängt werden.

In einem Baum der Tiefe n lassen sich höchsten 2^n-1 Knoten unterbringen. Im ungünstigsten Fall lassen sich nur n Elemente unterbringen, dies entspricht genau dem Fall, in dem jeder Knoten nur einen Nachfolger besitzt. Wenn die Knoten untereinander keine Ordnung besitzen ist der Baum ungeordnet. Bei geordneten Bäumen ist eine vollständige Ordnungsrelation zwischen den Knoten Vorraussetzung, diese wird oft über Schlüssel realisiert. (Bsp.: numerische Anordnung bei Zahlen und die lexikographische Ordnung beim Alphabet) Hat man nun eine vollständige Ordnungsrelation zwischen den Knoten eines Binärbaums gefunden, dann ist jeder Knoten des Baumes kleiner als alle Knoten seines rechten Teilbaums und zugleich größer oder gleich als alle Knoten seines linken Teilbaums.

26.1. Der Suchbaum

Der geordnete Baum erlaubt eine schnelle Suche nach Informationen, ähnlich wie das Binärsuchverfahren in linearen Strukturen. Der geordnete Baum heißt deshalb auch Suchbaum.

Bsp: gegeben ist folgende Liste

A → B → C→ D → E → F → G → H → NULL

Der direkte Zugriff auf die Elemente ist möglich. Bei der binären Suche wird auf das mittlere Element zugegriffen. Das Anfangselement ist somit „E" das die Liste in 2 Teile teilt.

```
┌───────────────────────── E────┐
A → B → C→ D                    F → G → H → NULL
```

Ist „E" nicht das gesuchte Element wird links oder rechts weitergesucht

```
     ┌───────────────────────── E────┐
┌───B────┐                   ┌────G────┐
A        C→ D                F         H
```

Teilt man die Liste solange nach diesem Verfahren, bis die Teilbäume aus nur noch je einem Element bestehen, so liegt ein geordneter Baum vor. Einen derart

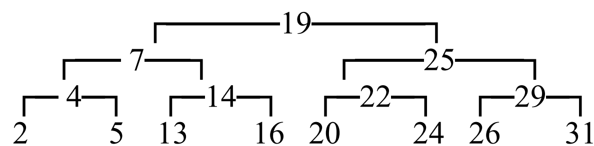
geordneten Baum bezeichnet man auch als binären Suchbaum. Dadurch wurde die linear verkettet Struktur in der nur sequentiell gesucht werden kann, durch Umstrukturierung in eine binäre Baumstruktur umgewandelt.

26.2. Bäume speichern (Traversierung)

Auch bei der Datenstruktur Baum besteht die Notwendigkeit (Möglichkeit) Elemente dauerhaft zu speichern und bei Bedarf zu laden. Dies führt zu dem Problem die binäre Struktur so in eine lineare Struktur umzuwandeln, dass sie Daten später wieder rekonstruiert werden können.

26.2.1. Inorder Traversierung

Bei der Inorder Traversierung werden die Daten nach folgender Reihenfolge sequentiell in eine Datei geschrieben: linker Teilbaum, Wurzel, rechter Teilbaum



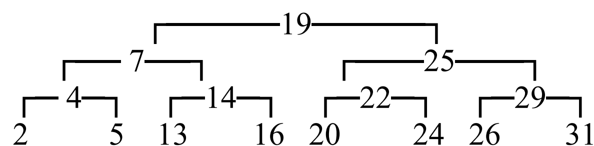
2 – 4 – 5 – 7 – 13 – 14 – 16 – 19 – 20 – 22 – 24 – 25 – 26 – 29 – 31

Anwendungsbeispiel:

Die Inorder Traversierung ist eine typische Strategie um eine vollständig sortierte Liste aus einem Suchbaum zu erhalten.

26.2.2. Postorder Traversierung

Bei der Postorder Traversierung werden die Daten nach folgender Reihenfolge sequentiell in eine Datei geschrieben: linker Teilbaum, rechter Teilbaum, Wurzel



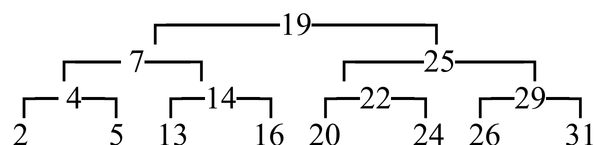
2 – 5 – 4 – 13 – 16 – 14 – 7 – 20 – 24 – 22 – 26 – 31 – 29 – 25 – 19

Anwendungsbeispiel:

Bei Taschenrechner oder einfachen Automaten müssen zuerst intern die Argumente einer Rechenoperation ins Register geladen werden und danach erst die verknüpfenden Operationen.

26.2.3. Preorder Traversierung

Bei der Preorder Traversierung werden die Daten nach folgender Reihenfolge sequentiell in eine Datei geschrieben: Wurzel, linker Teilbaum, rechter Teilbaum



19 – 7 – 4 – 25 – 14 – 13 – 16 – 25 – 22 – 20 – 24 – 29 – 26 – 31

Anwendungsbeispiel:
Im Zusammenhang mit Funktionen kann in der Wurzel der Funktionsaufruf stehen in den Teilbäumen die Argumente

27. Statische Bibliotheken erzeugen

Statische Bibliotheken enthalten jene Teile eines Programms die der Programmierer der die Biblothek benutzt nicht sehen darf. Um die Bibliothek verwenden zu können benötigt der Programmierer die Bibliothek selbst, die Headerdatei und Informationen (eine Dokumentation) dazu wie die Schnittstellen zu benutzen sind. Benötigt man eine oder mehrere Funktionen öfters so kann man diese in einer Statischen Bibliothek zusammenfassen. Diese muss dann nur mehr in das aktuelle Projekt eingefügt werden. Achtung, die Dateien die man in die Bibliothek einfügt, sollte man unbedingt aufbehalten um später eventuelle Fehler noch korrigieren zu können.

Erstellen einer statischen Bibliothek in Microsoft Visual C++:

1. Neu → Projekt → Win32-Bibliothek
2. Die Headerdatein und Cpp-Datein einfügen
3. Das Programm ausführen um die Bibliothek zu erzeugen
4. Die fertige Bibliothek liegt im Debug-Ordner des Projektes
5. Die Bibliothek und die Headerdatei gehören zusammen !

Einfügen einer statischen Bibliothek in ein Projekt:

1. Neues Projekt erzeugen
2. Das Hauptprogramm mit allen Schnittstellen ausprogrammieren
3. Den Pfad in dem die Bibliothek liegt im Menü Extras → Optionen → Verzeichnisse → Bibliothekdateien angeben
4. Die Headerdatei über die Präprozessorsyntax „#include<Name>“ einfügen
5. Am Schluss muss noch folgende Codezeile eingefügt werden:
   #pragma comment(lib, „Name.lib“)
6. Erstellen sie das Projekt