Programmieren mit C++

Karsten Baensch, Thomas Müller
TU Chemnitz
Universitätsrechenzentrum

15. September 1997

Inhaltsverzeichnis

0 Start

- Ziel des Kurses
  - Vermitteln der Konzepte der ooP
  - Erlernen der (Syntax und) Semantik von C++
  - Praktische Hinweise für die Entwicklung mit C++
  - Fähigkeit, vorgefertigte Bibliotheken zu verwenden
  - nicht im Vordergrund: Umgang mit spezifischen C++ Entwicklungssystemen (z.B. IDE von Borland, spez. Debugger, Klassenbrowser, etc.), Inhalt und Anwendung von spezifischen Klassenbibliotheken (Borland, Microsoft, Rogue Wave, etc.)
- Inhalt
  - Begriffe und Methodik der ooP
  - Grundlegende Konzepte
  - (Ablaufsteuerung und Operatoren)
  - Klassen als Abstrakte Datentypen
  - Operatorfunktionen
  - Templates
  - Vererbung und Aufbau von Klassenhierarchien (Einfachvererbung, Mehrfachvererbung, virtuelle Ableitung, polymorphes Verhalten)
  - Run-Time-Type-Informations (RTTI)
  - C++ - Namespaces
  - Fehlerbehandlung mit Exception Handling
  - I/O mit Streams
- Voraussetzungen
  - Kenntnisse der Programmierungstechnik
  - praktische Erfahrungen in der Programmiersprache C
(hier Änderung gegenüber vergangenen Jahren!)
  - $\Rightarrow$ kein Kurs für Programmieranfänger
- Durchführung:
  - Vortrag/Seminar, unterstützt durch unmoderierte praktische Übungen

1 Einführung

1.1 Literatur

Nachschlagewerke:

Stroustrup, B.
The C++ Programming Language
Second Edition
Addison-Wesley 1991

bzw.

Stroustrup, B.:
Die C++ Programmiersprache
2. Auflage 1992, 4. überarbeiteter Nachdruck 1994
Addison-Wesley 1992

Stroustrup, B.; Ellis, M.A.:
The Annotated C++ Reference Manual
Addison-Wesley 1990

Stroustrup, B.:
The design and evolution of C++
Addison-Wesley 1994

bzw.

Stroustrup, B.:
Design und Entwicklung von C++
Addison-Wesley 1994

Plauger, P.J.:
The Draft Standard C++ library
Englew.Cl:Prent.Hall 1995

für Profis:

Bruce Eckel:
Thinking in C++
Englewood Cliffs, NJ (Prentice Hall) 1995

siehe auch: `http://www.cs.uidaho.edu/~frincke/misc/booklists/ood_books.html`

1.2 Abstraktionsebenen des Software-Engineering

Abstraktionsebene	typische Programmiersprache
Subroutine	Assembler, Basic, Fortran
↓	
Funktion, Prozedur	C, Pascal
↓	
Modul	Modula-2
↓	
Abstrakter Datentyp	Ada
↓	
Objektorientierte Klassenhierarchie	C++, Smalltalk, Eiffel, SIMULA 67, Java

1.3 Was ist ooP?

(Begriffe und Methodik der ooP)

- Methodik der ooP ist stark an der menschlichen Denk- und Vorstellungswelt orientiert.

- Gewöhnlich denkt man in Kategorien (Begriffen) und verwendet dabei den Begriff „Objekt“ (mein Auto, das Auto meines Nachbarn, das Zimmer 364a ...)
  ⇒ reale, greifbare Dinge (Objekte)
  Oder allgemeiner: ein Auto, ein Auto einer Automarke, ein Zimmer ...
  ⇒ bestimmte Eigenschaften, Fähigkeiten (Kategorien).

- Der Umgang mit Objekten der Umwelt erfolgt durch Inanspruchnahme bestimmter Fähigkeiten. ⇒ Dazu findet Kommunikation statt: Nachricht senden!
  (mit Kollege: „Schließe bitte das Fenster!“
  mit Radio: „Einschalten!“)

- Die Reaktion auf eine solche Kommunikationsanforderung wird unterschiedlich sein:
  Kollege: er tuts oder „Machs doch selbst!“
  Radio: In Abhängigkeit des korrekten Anschlusses und der Funktionstüchtigkeit wird es „spielen“.
  In jedem Fall ist die Reaktion individuell!
  (Das Radio spielt nur, wenn alle notwendigen Bedingungen erfüllt sind.)
  Es bleibt verborgen, <u>wie</u> es das Radio anstellt, tatsächlich zu spielen; oder <u>wie</u> der Kollege das Fenster schließt. Ggf. kommuniziert das angesprochene Objekt mit weiteren Objekten.

- Individualität aber auch: Versteht das angesprochene Objekt die Nachicht überhaupt?
  ⇒ Das Radio wird das Fenster nicht schließen können. (Es gibt keinen

Knopf, wo „Fenster schließen“ drauf steht.)
Aber: Eine Waschmaschine kann man auch einschalten, nur die Waschmaschine wird keine Musik spielen, sondern waschen.

⟶ gleiche Nachrichten an verschiedene Objekte

eines Typs	verschiedener Typen
Radio A spielt	Radio spielt
Radio B spielt nicht, weil Netzstecker gezogen ist.	Waschmaschine wäscht

- Welche Nachrichten versteht ein Objekt?
  Radio: Bedienungsanleitung lesen oder Erfahrungen nutzen
  ⇒ Man benötigt Kenntnisse über die „Schnittstelle“ eines Objektes.
  (Dabei reicht es, wenn man die Bedienanleitung des Radiotyps liest und kann dann jedes Radio dieses Typs bedienen (benutzen).)
  ⇒ Objekte haben eine Schnittstelle, die ihre Benutzung ermöglicht.

- Wo hat das Objekt seine Eigenschaften/Fähigkeiten her?
  mehrere Möglichkeiten:

  1. Es besteht aus Teilobjekten und kombiniert durch Zusammenwirken dieser Teilobjekte deren Eigenschaften/Fähigkeiten zu neuen Eigenschaften/Fähigkeiten (Radio besteht aus etlichen Einzelteilen).
  2. Weiterentwicklung bestehender Typen (bewährte Teile/Eigenschaften erhalten, neue ergänzen)
  3. Zusammenführen der Eigenschaften mehrerer Typen

- bei der Entwicklung neuer Typen:
  - vollständige Neuentwicklung
  - ausgehend von vorhandenen Typen, Modifizierungen durch:
    - ergänzen
    - weglassen
    - ersetzen
    - weiter kombinieren

⇒ Eigentlich ist mit diesen Bemerkungen das Wesen der ooP umrissen, die beschriebenen Abläufe werden durch die ooP aufgegriffen und unterstützt.

Zentrale Begriffe der ooP

ooP	herkömmliche Bedeutung/Erläuterung
Klasse	Begriff, Typ, Kategorie (z.B. „Auto“ oder auch spez. Autotyp, ...)
Objekt, Instanz	Exemplar, Gegenstand („ein Objekt (Instanz) einer Klasse“ → analog: „mein“ Auto ist ein Exemplar eines Autotyps)
Kapselung, *information hiding*	Struktur und Inhalt interner Daten sowie Implementation der Aktionen vor dem äußeren Betrachter versteckt; nur „offizielle“ Schnittstelle bekannt
Nachricht, Botschaft (Methodenaufruf)	Aufforderung zu einer Aktion
Methode, Funktion, Operation	Implementation einer Aktion
Argumente	zur Ausführung einer Aktion **gelieferte** Informationen / Werte / Objekte
Parameter	zur Ausführung einer Aktion **benötigte** Informationen / Werte / Objekte

ooP	herkömmliche Bedeutung/Erläuterung
Vererbung, Ableitung	Übernahme einer vorhandenen Kategorie (Eigenschaften / Fähigkeiten) zum Ziele der Erweiterung; (nicht im Sinne der Werteübernahme, nicht alle Autos sind rot, sondern jedes Auto hat eine farbige Lackierung); Is-A-Relation: „Ein Trabant **ist** ein Auto.“ „Ein Auto **ist** ein Fahrzeug.“
Basisklasse, Basis	Klasse, von der abgeleitet wird; allgemeinere Klasse
abgeleitete Klasse	Klasse, die von Basisklasse abgeleitet worden ist; spezialisiertere Klasse
multiple Vererbung	Ableitung von mehreren Basisklassen
Objekt als Aggregat	Objekt, welches durch Zusammenfügen mehrerer Objekte entsteht Has-A-Relation: „Ein Auto **hat** eine Menge von Rädern.“, und aus „Ein Trabi **ist** ein Auto.“ folgt: „Ein Trabi **hat** eine Menge von Rädern.“

ooP	herkömmliche Bedeutung/Erläuterung
(Instanz-)Variablen	speichern Objekte / Instanzen
Elemente eines Objektes	Variablen, in denen die Daten gespeichert sind, aus denen das Objekt (Aggregat) besteht; in der Regel nach außen nicht sichtbar
Polymorphismus	Gleichlautende Nachricht wirkt verschieden in verschiedenen Zusammenhängen.
(Vielgestaltigkeit)	der Objekte: individuelle Verarbeitung durch Objekte verschiedener Klassen („Einschalten!“: Radio, Waschmaschine) der Argumente: individuelle Verarbeitung der gleichlautenden Nachricht aufgrund unterschiedlicher Argumente („Waschen(Was?)!“: Wäsche, Hände oder Füsse)
statisches Binden	(zur Compilezeit:) Es ist von vornherein bekannt, an welche Methode und an welches Objekt eine Nachricht geht.
dynamisches Binden	(zur Laufzeit:) Es ist nur bekannt, von welchem Typ ein Objekt (mindestens) sein muß, damit es eine Nachricht befolgen kann.

Objekte bilden die Einheit für

- Kapselung von Daten/Informationen
- Aktivität (ausgelöst durch Empfang von Botschaften)
- Kommunikation (Austausch von Botschaften)
- evtl. Nebenläufigkeit (Selbständigkeit der Objekte)

Botschaften

Adressat	Selektor	Argumente
wer	was	womit
spezifiziert Zielobjekt direkte oder indirekte Angabe	welche Methode soll das Objekt ausführen	Informationen für auszuführende Operationen

Art und Weise des Zusammenwirkens von Objekten

1. Delegation: Inanspruchnahme von Dienstleistungen
(Client–Server– oder auch Master–Slave–Beziehung)
2. Kooperation: Weitergabe von Botschaften
(Informieren anderer Objekte über eingetretene Ereignisse)

Klassen

- Datenabstraktionen $\rightarrow$ Abstrakte Datentypen (ADT's)
Jedes Objekt ist Exemplar einer Klasse ($\rightarrow$ ADT-Instanz).
- beschreiben
  - die in Objekten der Klasse gekapselte Datenstruktur (Instanzvariablen), damit das Layout ihrer Instanzen
  - die darüber definierten Operationen (Methoden),
dabei sowohl Schnittstelle als auch Implementation
- sind im allgemeinen oo Sinne wiederum Objekte
$\Rightarrow$ Konsequenz
  - Klasse kann eigene Datenstrukturen kapseln (Klassenvariablen)
  - eigene Methoden besitzen (Klassenmethoden, statische Methoden)
- In C++ sind Klassen (per Sprache) keine Objekte! (Aber: Es gibt Objekte, die gewisse Klassen(Typ–)informationen speichern können.)
- eine Klasse legt die Datenstrukturen für **alle** ihre Objekte fest
- der Freiheitsgrad der Objekte liegt in der Belegung ihrer Datenstrukturen mit konkreten, individuellen Werten

- Klassenvariablen (in C++: statische Elemente) sind jedoch in ihrer Wertebelegung für alle Objekte der Klasse gleich

Fazit: Im Mittelpunkt eines oo Programmes stehen Datentypen und deren Funktionalität und nicht elementare Funktionen!

1.4 Die Historie von C++

- 1979: Start der Arbeit an „C mit Klassen" als effizientere Alternative zu SIMULA67 bei der Entwicklung großer Simulationsprojekte, also eher aus eigennützigen Gründen von Bjarne Stroustrup und seinen Kollegen; 1980: erstes internes Bell Labs Paper;
  1982: erste externe Veröffentlichung:

  Stroustrup, B.:
  Classes: An Abstract Data Type Facility for the C Language
  ACM SIGPLAN Notices Vol 17, no 1, January 1982

- 1983: Redesign, Erweiterungen (virtuelle Funktionen, Operator Overloading); Name „C++", Forschungsgruppe bei AT&T Bell Laboratories unter Leitung von Bjarne Stroustrup

  Stroustrup, B.:
  Data Abstraction in C
  AT&T Bell Laboratories Technical Journal Vol. 63, No 8, October 1984

- 1985: allgemeine Verfügbarkeit von C++ mit Cfront Release 1.0 (kommerziell)

  Stroustrup, B.:
  The C++ Programming Language
  Addison Wesley, 1986

- 1989: Cfront Release 2.0

- 1991: Cfront Release 3.0 (mit Templates)

  Stroustrup, B.:
  The C++ Programming Language (second edition)
  Addison Wesley, 1991

- Abstammung:
  - C (C ist Teilmenge von C++)
  - Simula-67 (Klassen, Ableitung, virtuelle Funktionen ...)
  - Algol-68 (Überladen von Operatoren, freie Plazierung von Deklarationen)

- C++ hat sich anfangs mehr oder weniger sporadisch entwickelt, es existierte kein Entwurfsgremium oder desgleichen, Entwurf, Implementation und Dokumentation entstanden parallel
  ⇒ Standardisierungsbestrebungen:
  - erster Quasistandard:
    Stroustrup, B.; Ellis, M.A.:
    The Annotated C++ Reference Manual
    Addison-Wesley 1990
  - 1989: X3J16-ANSI-Komitee einberufen, ANSI-Standardisierungsarbeiten sind Grundlage für ISO-Standardisierungsbemühungen (ISO WG21) für C++
  - 1995: Draft ANSI/ISO Standard (vorläufig)
  - 1996: ANSI/ISO stehen kurz vor dem Abschluß des Normierungsverfahrens für C++; ein Ergebnis ist auch die „Standard Template Library (STL)“

1.4.1 C++ - Compilerentwicklung

- zuerst als Precompiler (Translator) realisiert C++ → C
  Translator cfront findet auch heute noch (teilweise) Verwendung
- 1986: Cfront 1.1 von Glockenspiel
- 1987: erste GNU C++ Release
- 1988: erste Oregon Software C++ Release
- 1988: erste Zortech C++ Release
- 1990: erste Borland C++ Release
- 1992: erste Microsoft C++ Release
- 1992: erste IBM C++ Release
- Verschiedene Implementierungen der Sprache enthalten neben dem im jeweils aktuellen Cfront bzw. Ansi/ISO-Papier beschriebenen Sprachumfang eigene Erweiterungen.
  ⇒ Vorsicht bei der Anwendung, falls portabel programmiert werden soll!

2 C++ Grundlagen

2.1 Programmstruktur

Beispiel 1:

```
                    ↙ Headerfile (wo kommt das her? einige C++-Systeme:
stream.h)
#include    <iostream.h>      ← Anweisung für Präprozessor
  ↓1  ↓2             ↙3
int main()    //Beispiel 1: Erstes C++ - Programm
{←7
       cout << "Hello, world ! \n";
}←8     ↑4   ↑5        ↑6
```

2 Funktionsname, 1 Typ des Rückgabewertes
3 Kommentar
4 Objekt
5 Operator
6 Zeichenkette
7, 8 Funktionskörper

auch: `cout << "Hello," << " world ! \n";`

- Was macht das Pogramm?
- Ein C++ - Programm besteht aus Funktionen! (C-Historie)
- Eigentlich wird nur eine Funktion benötigt: `main`
  Dort geht die Ausführung los. (Es muß eine solche Funktion geben.)
- Offenbar findet eine Ausgabe statt: E/A gehört nicht zur Sprache C++, das Objekt `cout` ist Bestandteil einer Bibliothek (Sammlung vorgefertigter Programmbausteine).

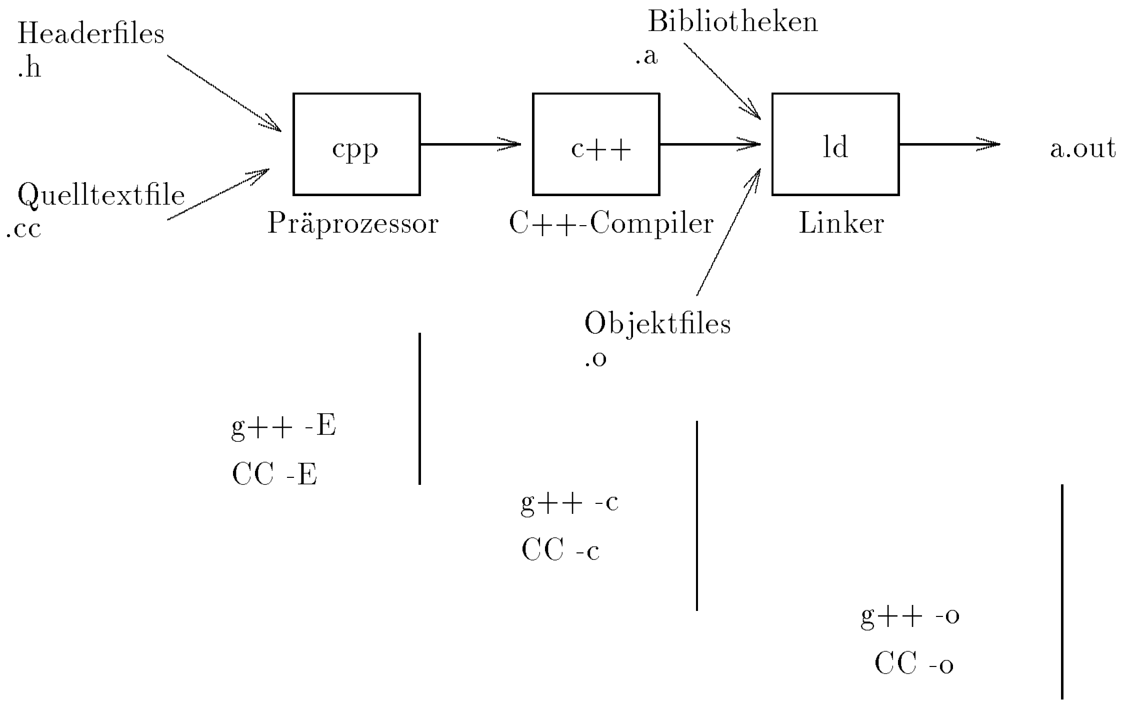
Beispiel 2:

```
#include  <stdio.h>
int main()               /* Beispiel 2: Erstes C++ Programm  */
{
      printf("Hello, world!\n");
}
```

- das ist gültiges C-Programm (C ist Teilmenge von C++)
- unterschiedliche Headerfiles
- ein Headerfile enthält Beschreibung von existierenden Implementationen, in seltenen Ausnahmen die Implementierungen selbst
- sie werden an zentraler Stelle im System gespeichert
  (`/usr/include; /usr/lib/g++-include`) (Standardheaderfiles)

- die dazugehörigen Implementationen werden in Bibliotheken gehalten, ebenfalls zentral gespeichert (`/lib; /usr/lib`)

- Arbeitsweise des C++ - Compilers



2.2 Lexikalische Konventionen

C++ Programm besteht aus Bezeichnern, Schlüsselworten, Literalen, Operatoren, Seperatoren, Kommentaren

2.2.1 Kommentare

- zwei Möglichkeiten:

  - /* Kommentar, auch über mehrere Zeilen */

    nicht schachteln, innerhalb können // /* ohne Sonderbedeutung verwendet werden

  - // Kommentar bis Zeilenende

    im Kommentar können // /* und */ verwendet werden

  - gegenseitig schachtelbar

2.2.2 Bezeichner

- beliebig lange Folge aus Buchstaben und Ziffern
- erstes Zeichen muß Buchstabe sein
- „_“ zählt als Buchstabe
- Groß- und Kleinschreibung wird unterschieden
- alle Zeichen sind signifikant
- dienen der Benennung von
  - Objekten (Variablen), Instanzvariablen
  - Klassen, Strukturen, ..., Enumerationen (Typen)
  - Funktionen (Methoden)
  - Konstanten

2.2.3 Seperatoren

- Leerzeichen, Tabulatoren (horizontal/vertikal), New-Line, Seitenvorschub (Form-Feed), Kommentare
  $\Rightarrow$ White Spaces
- trennen Bezeichner, Schlüsselworte, Konstanten
- werden sonst ignoriert

2.2.4 Schlüsselworte und Operatoren

asm, auto, bool, break, case, catch, char, class, 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, operator, 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, wchar_t, while, void, volatile, __rtti

Bemerkungen zu Schlüsselworten

- Neben diesen Schlüsselwörtern sollen Bezeichner, die mit einem doppelten Unterstrich (__) beginnen, nicht verwendet werden (reserviert für C++-Implementationen und Bibliotheken).
- ausschließlich C++ (nicht ANSI-C):
  bool, true, false,   class, private, protected, public, this, explicit
  virtual, friend, operator,   template,
  throw, try, catch,   __rtti, typeid, typename,
  static_cast, const_cast, dynamic_cast, reinterpret_cast,   mutable
  new, delete,   namespace, using,
  siganture (?), interface (?)
- Abhängig von der C++-Implementation kann es weitere Schlüsselwörter geben (z.B. Modifizierer für Anpassung an Speichermodelle (DOS/Windows), für Aufrufkonventionen (Windows), für Prolog/Epilog (Windows) oder Schlüsselwörter für noch nicht standardisierte Features, z.B. `signature`, `interface`)

Operatoren

arithmetische Operatoren
`+ ++ - -- * / %`
relationale Operatoren (comparison)
`< > <= >= == !=`
Bit-Operatoren
`&`(AND) `|`(OR) `^`(XOR) `~`(complement) `>> <<`(shift)
logische Operatoren
`&&`(AND) `||`(OR) `!`(NOT) `?:`(conditional expression)
Zuweisungsoperatoren (assignment)
`= += -= *= /= %= >>= <<= &= ^= |=`
Zugriffsoperatoren
`::`(scope res.) `&`(address of) `*`(deref.)
`. ->`(member sel.) `.* ->*`(member deref.)
`()`(func. call; val. constr.) `[]`(subscr.)
Speicherverwaltung
`new new[] delete delete[] sizeof`

andere
`throw`(throw exc.) `,`(sequencing)
zusätzlich von Präprozessor:
`#`

2.2.5 Literale Konstanten

Literal: es liegt nur der Wert vor, hat keinen Namen
Konstante: dieser Wert ist nicht änderbar
Integer-Konstanten, Zeichenkonstanten, Gleitkommakonstanten, Zeichenketten-konstanten
(nicht gemeint: symbol. Konstanten)

Zeichen-Konstanten:
ein (oder mehrere: `wchar_t`) in Apostroph eingeschlossene Zeichen
Bsp.: `'x'`
Der Wert einer Zeichen-Konstante ist der numerische Wert des Zeichens entsprechend des Zeichensatzes der Maschine (meist ASCII).
Darstellung einiger Sonderzeichen:

newline	`\n`
horiz. tab.	`\t`
vert. tab.	`\v`
backspace	`\b`
carriage return	`\r`
form feed	`\f`
alert (Glocke)	`\a`
backslash	`\\`
Fragezeichen ?	`\?`
Apostroph '	`\'`
Anführungszeichen "	`\"`
Oktalzahl	`\ooo`
Hexadezimalzahl	`\xhhh`
Null	`\0`

Integer-Konstanten: (Dezimal, Oktal, Hexadezimal-Konstante)[1]
Folge von Zahlen (0...9) – dezimal (nicht mit 0 beginnen !)

Folge von Zahlen (0...7) beginnend mit 0 – oktal

Folge von Zahlen (0...9, a - f, A - F) beginnend mit 0x oder 0X – hexadezimal

[1] Typ einer Integer-Konstanten

dezimal, kein Suffix:	abhängig vom Wert:	int, long int, unsigned long int
oktal/hexadez., kein Suffix:	abhängig vom Wert:	int, unsigned int, unsigned long int
Suffix u oder U:	abhängig von Wert:	unsigned int, unsigned long int
Suffix l oder L:	abhängig vom Wert:	long int, unsigned long int
ul, lu, ..., Ul, LU:	abhängig vom Wert:	unsigned long int

Beispiel: Zahl Zwölf: 12   014   0xC   0xc   0Xc   0XC

<u>Gleitkommakonstanten:</u>

Beispiel: 1.0   1.
0.347   .347
123.4   12.34E1   1.234E2   .01234E4   12340E-2

auch klein **e** möglich
Darstellungsform: doppelt genau
Suffix: **f** einfach genau
Beispiel: `3.0f`

<u>Zeichenketten:</u>

Zeichenfolge eingeschlossen in Anführungszeichen

```
"Hello, world ! \n"
```

Typ: ARRAY OF CHAR
Beachten: Compiler hängt an Zeichenfolge das Zeichen \0 automatisch an !
⇒ Ende einer Zeichenkette
Länge der Zeichenkette: Anzahl der Zeichen + 1 (Terminatorzeichen \0)

2.3 Grundlegende Konzepte

2.3.1 Typ-Kategorien

- Skalar
  - Integral (Zeichen, Integer und Aufzählung)
  - Arithmetisch (`float`, `double`)
  - Zeiger und Referenz
- Aggregat
  - Array
  - `struct`
  - `union`
  - `class`
- Funktion
- `void`

2.3.2 Grunddatentypen

- C++ ist eine typisierte Sprache, es finden wesentlich strengere Typprüfungen als in C statt
- nicht alle oo Sprachen besitzen ein Typkonzept (Smalltalk)
- Typkonzept gehört zu den grundlegendsten Konzepten von C++

- Datentypen für Zeichen, Integerwerte, Gleitkommazahlen

Zeichen:

- Objekte (Variable) des Typs `char` können Werte des Zeichensatzes der Maschine aufnehmen (meist 8 Bit)

- der Wert einer solchen Variablen entspricht dem Integer-Code des Zeichens

- es ist möglich, die Attribute `unsigned` und `signed` zu verwenden (Verwendung des höchstwertigen Bits als Vorzeichen)

- die Typen `char`, `unsigned char` und `signed char` sind drei verschiedene Typen

Integer:

- drei verschiedene Größen für Integerwerte:
  `short int`, `int`, `long int`

- `int`-Variablen sind so groß wie ein Maschinenwort
  (16-Bit-Rechner: 16 Bit, 32-Bit-Rechner: 32 Bit)

- in jedem Fall gilt: `long int` ist nicht kleiner als `short int`
  Länge von `short` <= Länge von `int` <= Länge von `long`

- meist gilt:

	16-Bit-Rechner	32-Bit-Rechner
`char`	8	8
`int`	16	32
`short int` (`short`)	16	16
`long int` (`long`)	32	32

- für `signed char`, `short`, `int`, `long` gibt es entsprechende `unsigned` Varianten:

```
unsigned char
unsigned int        (unsigned)
unsigned short int  (unsigned short)
unsigned long int   (unsigned long)
```

  ⇒ gleicher Speicherplatz (Länge) wie `signed`-Varianten
  ⇒ Variablen dieser Typen genügen der Arithmetik modulo 2^n (n - Anzahl der Bits) ⇒ `unsigned`-Arithmetik läuft nicht über !

  meist:

	16-Bit-Rechner	32-Bit-Rechner
`char`	-128 ... 127, 0 ... 255	-128 ... 127, 0 ... 255
`signed char`	-128 ... 127	-128 ... 127
`unsigned char`	0 ... 255	0 ... 255
`int`	$-2^{15} \dots 2^{15}-1$	$-2^{31} \dots 2^{31}$-1
`unsigned int`	$0 \dots 2^{16}-1$	$0 \dots 2^{32}-1$
`short`	$-2^{15} \dots 2^{15}-1$	$-2^{15} \dots 2^{15}$-1
`unsigned short`	$0 \dots 2^{16}-1$	$0 \dots 2^{16}-1$
`long`	$-2^{31} \dots 2^{31}$-1	$-2^{31} \dots 2^{31}$-1
`unsigned long`	$0 \dots 2^{32}-1$	$0 \dots 2^{32}-1$

diese Angaben findet man in <limits.h>

Gleitkommazahlen:

- drei verschiedene Typen (bezogen auf Genauigkeit)
  `float, double, long double`
  Genauigkeit:
  `double` nicht geringer als `float`
  `long double` nicht geringer als `double`
  - float: einfache Genauigkeit meist 32 Bit
  - double: doppelte Genauigkeit meist 64 Bit
    ⇒ <float.h>

Die `char`- und `int`-Typen sowie Enumerationen werden meist als integrale Typen bezeichnet und `float`-Typen als arithmetische Typen

der Typ `void`

- Typ für die leere Wertmenge
- es können keine Variablen dieses Typs vereinbart werden
- dient als Basis für einige abgeleitete Typen

2.3.3 Deklaration und Definition

- in einem C++-Programm müssen alle Bezeichner vor ihrer Verwendung deklariert werden (Vorausdeklaration)
- Namen bezeichnen ein Objekt, eine Funktion, eine Menge von Funktionen, einen Typ, ein Klassenmember, einen Wert oder ein Label
- ein Name ist nur innerhalb eines bestimmten Programmteils verwendbar: seinem Gültigkeitsbereich (scope)
- mit einem Namen ist ein Typ assoziiert
- der Typ bestimmt die Verwendung des Namens

- da C++-Programme in mehrere Übersetzungseinheiten (Quelltextfiles) aufgeteilt werden, bestimmen sogenannte linkage rules, ob ein Name ein und dasselbe Objekt, ein und dieselbe Funktion, ein und denselben Typ, Wert, ... bezeichnet

Deklaration: führt einen oder mehrere Namen in ein Programm ein

Definition: ist eine Deklaration und ist mit der Vergabe von Speicherplatz und/oder der Initialisierung verbunden

Beispiele:
`int a;`

- macht den Namen `a` bekannt
- Typ des Namens: `int`
- vereinbart Speicherplatz; soviel wie für die Aufnahme eines `int`-Wertes gebraucht wird

`float x,y;`

- wie oben: aber macht `x` und `y` bekannt

`extern int b;`

- macht den Namen `b` bekannt
- Typ des Namens: `int`
- `extern`: keine Definition → die Speicherplatzvergabe findet irgendwo anders statt

aber:
`extern const int c = 1;`

- macht den Namen `c` bekannt
- Typ des Namens: `int`-Konstante
- `extern`: dieser Name wird auch in anderer Übersetzungseinheit benutzt und soll sich dort auf den gleichen Wert beziehen
- es findet eine Initialisierung statt: ⇒ Definition

Merke: jede Deklaration, die einen Wert spezifiziert ist eine Definition:
Eine Deklaration ist eine Definition in folgenden Fällen:

- Deklaration einer Funktion mit Spezifikation des Funktionskörpers
  Bsp.:
  ```
  int f(int x) {return x+a;}
  ```

- Verwendung des Schlüsselwortes extern und Spezifikation einer Initialisierung oder eines Funktionskörpers
  Bsp.:
  ```
  extern unsigned n = 1;
  ```
  siehe oben mit vorangestellten extern

- wenn nicht nur der Name einer Klasse deklariert wird
  Bsp.:
  ```
  class X {
    int x;
  public:
    x(int xx);
    void print();
  };
  ```
  $\Rightarrow$ Typdefinition

Eine Deklaration ist <u>keine</u> Definition in folgenden Fällen

- Umkehrung der obigen 3 Punkte

- Deklaration von statischen Datenmember von Klassen
  Bsp.:
  ```
  class X {
     static int statx;   // keine Definition !
        .
        .
        .
  };
  ```

Allgemeine Form von Deklarationen:

```
<typ>  <name>, . . .;
```

Jedes Objekt, Funktion, Klasse, Enumerator muß genau einmal definiert werden ! Es kann aber mehrfach deklariert werden (im Sinne einer Deklaration, die keine Definition ist)
Diesen Umstand nutzt man bei der Programmierung:

Headerfiles: Typdefinitionen
Funktionsdeklararionen
Datendeklarationen
Enumerationen
Includes, Makros, Kommentare

Implementationsfiles:
Einschließen der Headerfiles mittels #include zu verschiedenen Zwecken:

1. Definition der im Headerfile deklarierten Namen (das geht nur einmal!)

2. Verwendung der deklarierten Namen !
   siehe Anfangsbeispiel:

   ```
   #include <iostream.h>
   main()
   ```

```
{
      cout << "Hello, world ! \n";
}
```

`cout` ist kein Schlüsselwort, also muß es ein Name sein, dieser Name hat einen Typ $\Rightarrow$ es sind bestimmte Operationen damit möglich, der Name muß also in iostream.h deklariert sein

Frage: geht so was ?:

```
int counter;
int counter;

extern unsigned error_nr;
extern unsigned short error_nr;

extern float geschwindigkeit;
extern float geschwindigkeit;

extern const double pi;
extern const double pi = 3.141592653589;

extern const double e = 2.7182818284;
extern double e;

extern int f();
int f() {return 3;}
```

- es gibt keinerlei Einschränkungen für den Ort einer Deklaration, wichtig ist lediglich, daß ein Name vor der Verwendung deklariert sein muß

2.3.4 Gültigkeitsbereich und Lebensdauer

- verbunden mit der Deklaration eines Namens ist die Einführung in einen Gültigkeitsbereich (scope), d. h. ein Name kann nur in einem bestimmten Teil eines Programms verwendet werden
  - wesentliche Kriterien: erfolgt die Deklaration
    1. innerhalb einer Funktion oder außerhalb
    2. innerhalb einer Klassendefinition oder außerhalb
    3. innerhalb einer `namespace`-Definition oder außerhalb
  - Namen, für die bei den Punkten 1-3 'außerhalb' gilt, sind vom Punkt der Deklaration bis zum Ende des Files gültig.
  - Deklaration innerhalb einer Funktion: vom Punkt der Deklaration bis zum Ende des Blockes gültig

- Deklaration innerhalb einer Klassendefinition: in dieser Klassendefinition und in Definitionen der Methoden dieser Klasse gültig
- Deklaration innerhalb einer `namespace`-Definition: wird durch `using`-Deklaration in einen bestimmten Gültigkeitsbereich eingeführt

  Beispiel:

```
int x;

int f()
{
    int x;           // lokales x verbirgt globales x
    x = 1;
    {
         float x;    // neues x mit anderem Typ
         x = 2.0;
    }
    return x;
}

float g()
{
    float y;
    y = x - 3.0;     // globales x
    return y;
}
```

  Bezugnahme auf verborgene Namen:

```
int x;

int f()
{
    int x = 1;
    ::x = 2;
    return x;
}
```

- Verbergen von Namen nur im Zusammenhang mit Gültigkeitsbereich (siehe oben)
- Im Zusammenhang mit der Klassendefinition existieren weitere Kriterien, die dort besprochen werden

• eng im Zusammenhang mit dem Gültigkeitsbereich steht der Begriff Lebensdauer

  - er beschreibt, wie lange ein Objekt existiert, also zu welchem Zeitpunkt es angelegt und wann es wieder zerstört wird

- außerhalb von Funktionen vereinbarte Objekte (globale Objekte) und Objekte, die innerhalb von Funktionen mit dem Schlüsselwort `static` vereinbart werden (statische Objekte), existieren (leben) während der gesamten Ausführungszeit des Programms
- alle anderen, innerhalb von Funktionen vereinbarten Objekte (automatische Objekte) existieren (leben) nur, wenn der Block, zu dem sie gehören, abgearbeitet wird
- das Schlüsselwort `static` beeinflußt auch den Gültigkeitsbereich von Namen, die außerhalb von Funktionen vereinbart sind, diese Namen sind dann nur innerhalb des Quelltextfiles gültig, zu dem sie gehören, man kann aus anderen Quelltextfiles nicht auf diese Objekte zugreifen
- mit den Operatoren `new/delete` kann man die Lebensdauer von Objekten explizit steuern

2.3.5 Objekte und Lvalues

- gewöhnlich wird auf den Wert eines Objektes über seinen Namen zugegriffen

```
int x, y = 2;
x = 3*y+4;
```

- es gibt die Möglichkeit Objekte anzulegen und zu verwenden, die keinen Namen haben, dann entstehen Ausdrücke wie

```
*p[a+10] = 12;
```

  man braucht also einen Begriff für `*p[a+10]`
  das ist "irgendetwas im Speicher" → Lvalue

- Ein Objekt nimmt Speicher ein.
  Ein Lvalue ist ein Ausdruck, der sich auf ein Objekt bezieht.

2.3.6 Ausdrücke und Operatoren

- Ausdrücke bestehen aus Operanden und Operatoren
- die Operanden sind Objekte (Variablen oder Konstanten) oder wiederum Ausdrücke
- die Auswerung jedes Ausdrucks liefert einen Wert, der sich aus der Anwendung von Operatoren auf die Operanden ergibt (diese Ausdrucks-bezogene und Steuerfluß-orientierte Betrachtungsweise hat in C++ durchaus noch ihre Berechtigung (C-Historie), obwohl man eigentlich vom oo Standpunkt aus das Botschaftenkonzept in den Vordergrund stellen müßte: die Auswertung eines Ausdrucks liefert einen Wert, der sich aus der Reaktion eines Objektes aufgrund einer Botschaft ergibt)
- Wertzuweisung
  - Wertzuweisung ist eine Ausdruck

- dient der Zuordung eines Wertes an ein Objekt (eine Variable)
  Bsp.:
  ```
  x = 0
  y = x + 4
  c = getchar()
  ```
- der rechts von = stehende Ausdruck wird berechnet und sein Wert dem Objekt (der Variablen) auf der linken Seite zugewiesen
- allgemein: `<lvalue> = <ausdruck>`
- jede Wertzuweisung besitzt einen Wert (sie ist ein Ausdruck!), ⇒ Mehrfachzuweisung:
  ```
  a = b = c = 1
  ```
  wird von rechts nach links berechnet, d.h.:
  ```
  a = (b = (c = 1))
  ```
- Wertzuweisungsoperator kann in einem Ausdruck wie jeder andere Operator benutzt werden
  Bsp.:
  ```
  (c = getchar()) == '\n'
  ```
  zunächst wird mittels `getchar()` das nächste Zeichen von der Standardeingabe gelesen und dann mit der Wertzuweisung der Variable c zugewiesen, anschließend wird geprüft, ob es sich bei diesem Wert um das Zeichen Newline (`'\n'`) handelt

- Arithmetische Operatoren
  - elementare arithmetische Operation: Bildung des Zweierkomplements unärer Operator:
    ```
    -          // negatives Vorzeichen
    ```
  - binäre Operatoren:
    ```
    +, -    // Addition, Subtraktion
    *, /    // Multiplikation, Division
    %       // Rest der ganzzahligen Division
    ```
  - bei der Division `x/y` von positiven Integerwerten `x` und `y` wird der gebrochene Teil abgeschnitten
  - ist einer der Operanden negativ, so ist die Form des Abschneidens maschinenabhängig
  - `x%y` ermittelt den Divisionsrest, das Vorzeichen des Ergebnisses ist maschinenabhängig

- Vergleichsoperatoren
  - Operatoren
    ```
    >, >=    // größer, größer gleich
    <, <=    // kleiner, kleiner gleich
    ==       // gleich
    !=       // ungleich
    ```

- per Definition besitzt ein Vergleichsausdruck der Form

  $<ausdruck_1> <rel\_op> <ausdruck_2>$

  den Wert 1, wenn die durch $<rel\_op>$ spezifizierte Bedingung erfüllt ist und sonst den Wert 0

- Bsp.:

```
int x, y, schalter;
    . . .
x = 4;
y = 2;
schalter = x > y;
```

  im Ergebnis der letzten Anweisung wird der Variablen `schalter` der Wert 1 zugewiesen

- Unterschied Operatoren = und == beachten!

2.3.7 Implizite Typkonvertierung

- in Ausdrücken können Objekte der Grunddatentypen gemischt werden
- wenn es möglich ist, werden die Werte so konvertiert, daß keine Information verloren geht
- im Zweifelsfall: Reference Manual

2.4 Ablaufsteuerung I

2.4.1 Einfache Anweisung (expression statement)

- $<ausdruck>$;
  durch nachgestelltes Semikolon wird ein Ausdruck zur Anweisung
- gewöhnlich Wertzuweisung oder Funktionsaufruf oder Methodenaufruf eines Objektes (Senden einer Nachricht an ein Objekt)

```
x = 5;
printf("Hello, world ! \n");
stack.push(5);
```

- einfachster Fall:

```
;         //Null-Anweisung/leere Anweisung
```

- in bestimmten Fällen sinnvoll: da, wo aus syntaktischen Gründen eine Anweisung erforderlich ist, aber nichts gemacht werden soll (später Beispiele)

2.4.2 Block

- geschweifte Klammern fassen mehrere Anweisungen und Deklarationen / Definitionen zu einem Block zusammen
- spezieller Block: Funktionskörper

```
main()
{
       int i;
        ...
       i = 0;
       int n = i+1;
        ...
}
```

2.4.3 if - else - Anweisung

- Auswertung von Entscheidungen `if (<ausdruck>)`

```
     <anweisung1 >
else
     <anweisung2 >
```

- Wahrheitswert:
  numerischer Wert von <ausdruck> verschieden von Null ⇒ True
  numerischer Wert von <ausdruck> gleich Null ⇒ False

- Beispiel:

```
if (z != 0)
    y = x/z;
else
    cout << "Division durch Null \n";
```

  der Ausdruck `z != 0` liefert den Wert 1, falls der Wert von z verschieden von 0 ist, er liefert den Wert 0, falls z gleich 0 ist

```
if (z)
     y = x/z;
else
     cout << "Division durch Null \n";
```

- anstelle einer einzelnen Anweisung kann ein Block stehen

2.4.4 while - Schleifen

- zwei Varianten:
  while
  do-while

- while:

```
while (<ausdruck>)
      <anweisung>
```

- zuerst wird <*ausdruck*> ausgewertet
- falls Wahrheitswert True wird <*anweisung*> ausgeführt und anschließend erneut <*ausdruck*> berechnet → Zyklus
- falls Wahrheitswert False wird die Abarbeitung der Schleife beendet
- Sprechweise: solange <*ausdruck*> wahr ist, führe <*anweisung*> aus

```
int s = 0, i = 1;

while (i <= 10) {
    s = s+i;
    i = i+1;
}
```

oder

```
int i, j = 0;
int f();

while (i = f())        // while((i = f()) != 0)
     j = j + i;
```

- do - while

```
do
     <anweisung>
while (<ausdruck>);
```

<*anweisung*> wird mindestens einmal ausgeführt!

2.4.5 Beispiele

1. Lebensdauer (Schlüsselwort `static`)

```
#include <iostream.h>

int a = 1;                 // Definition: globales Objekt

void f()                   // Definition: Funktion
{
    int b = 1;             // Definition: automatisches Objekt
    static int c = 1;      // Definition: statisches Objekt

    cout << "a = " << a;
    cout << " b = " << b;
    cout << " c = " << c << "\n";
    a = a + 1;
    b = b + 1;
    c = c + 1;
}

int main()
```

```
{
    extern int a;          // Deklaration, nicht notwendig
    extern void f();       // Deklaration, nicht notwendig

    while (a < 4) f();
}

/* Erwartete Ausgaben:
a = 1 b = 1 c = 1
a = 2 b = 1 c = 2
a = 3 b = 1 c = 3
*/
```

2. File kopieren mit Zeichen zählen

```
#include <iostream.h>

int main()
{
    char c;
    int z = 0;

    while(cin.get(c)) {
        cout.put(c);
        z = z + 1;
    }
    cerr << "Anzahl kopierter Zeichen: " << z << "\n";
}
```

oder:

```
#include <iostream.h>

int main()
{
    char c;
    int z = 0;

    while(cin >> c) {
        cout << c;
        z = z + 1;
    }
    cerr << "Anzahl kopierter Zeichen: " << z << "\n";
}
```

in der zweiten Variante, werden vom Objekt `cin` Whitespaces überlesen, d.h. die Variable `c` nimmt nur solche Werte an, deren Zeichen den Bildschirm „schwarz machen“

Aufgabe:
Programme testen
2. so modifizieren, daß Zeilenanzahl ermittelt wird

2.5 Abgeleitete Typen

2.5.1 Felder

- für einen Typ T ist
$$T[\,]$$
der Typ "Feld von Elementen des Typs T"

- eine Definition der Form
$$T < name > [< anzahl >]$$
definiert den Bezeichner $< name >$ als Name eines Feldes von $< anzahl >$ Elementen des Typs T
Bsp.:
```
float array[10];
const int ci = 15;
int feld[3 * ci];
```

- für $< anzahl >$ können Konstantenausdrücke verwendet werden (deren Wert muß zur Übersetzungszeit berechnet werden können)

- der Zugriff zu Feldelementen erfolgt durch Indizierung, die Indizees laufen von 0 bis $< anzahl >$-1
Bsp.:
```
int i = 0;
while (i < 10) {
     array[i] = 0;
     i = i + 1;
}
```

- erinnern: Zeichenkettenkonstanten sind vom Typ "ARRAY OF CHAR"
```
#include <iostream.h>

char alpha[]="abcdefghijklmnopqrstuvwxyz";

int main()
{
   int i=0;
   char c;

   cout << "Version 1 mit Feldindizierung\n";
   while((c = alpha[i]) != 0) {
```

```
        cout << c
             << " = " << dec << int(c)   // Manipulator dec
             << " = 0" << oct << int(c)  // Manipulator oct
             << " = 0x" << hex << int(c) // Manipulator hex
             << "\n";
        i = i +1;
    }
}

/* Erwartete Ausgaben
a = 97 = 0141 = 0x61
b = 98 = 0142 = 0x62
c = 99 = 0143 = 0x63
       ...
x = 120 = 0170 = 0x78
y = 121 = 0171 = 0x79
z = 122 = 0172 = 0x7a
*/
```

Beispiel erläutern und probieren lassen!

- Initialisierung von Feldern

```
 int f1[] = { 1, 2, 3 };
 int f2[] = { 'a', 'b', 'c' };
char c1[] = { 1, 2, 3 };
char c2[] = { 'a', 'b', 'c' };     // nur 3 Zeichen kein \0
```

- mehrdimensionale Felder

```
int matrix[5][4]; //  vereinbart 5 * 4 Matrix: 5 Zeilen, 4 Spalten
```

genauer: es wird ein Feld mit 5 Elementen vereinbart, wobei jedes Element ein
Feld mit 4 Elementen des Typs `int` ist
mehrdimensionale Felder werden zeilenweise gespeichert

- erinnern:

```
char name[20];          // Definition
extern char name[20];   // Deklaration
```

identisch:

```
extern char name[];     // deshalb kann anzahl weggelassen werden
```

der Typ des Bezeichers `name` ist `char[]`

2.5.2 Funktionen

- für einen Typ T ist

  $$T(< parameterdeklarationsliste >)$$

  der Typ "Funktion mit Parametern der in $< parameterdeklarationsliste >$ angegebenen Typen und einem Funktionswert des Typs T

- eine Vereinbarung der Form

  $$T < name > (< parameterdeklaraionsliste >)$$

  vereinbart den Bezeichner $< name >$ als Funktion, die Objekte der in $< parameterdeklaraionsliste >$ angeführten Typen als Parameter übernimmt und einen Funktionswert vom Typ T liefert
  Bsp.:

```
int f();        // f ist eine Funktion ohne Parameter,
                // die einen Wert vom Typ int liefert

int g(void);    // aequivalent

int h(int);

int i = f();
int j = g();
int k = h(i);
int l = h();     // Fehler: so eine Funktion gibt es nicht
                 // es gibt keine Fkt. mit dem Namen h, die
                 // keine Parameter uebernimmt

int m = f(k);    // Fehler: so eine Funktion gibt es nicht
                 // analog

int n = h(g());  // korrekt!

int f(int,int);             // das sind verschiedene Funktionen
int f(char);                // die alle den Namen f tragen!
int f(float, double, int);  // function overloading!

double x;
float  y;

int o = f(y, x, f(i,j));   // Argumentpolymorphismus:
                           // aufgrund der Typen der uebergebenen
                           // Argumente erkennt der Compiler,
                           // welche Funktion f gemeint ist
```

  Bemerkung zu (Beziehungen zu ANSI-C):

```
int f(void);
```

```
int g(...); int h(int, ...)  // vgl. printf
```

- Standardargumente:

  Beispiel:

  ```
  void f(int = 0);

  void h()
  {
      f(1);
      f();          // bedeutet:  f(0);
  }
  ```

  - es sind auch mehrere Standardargumente möglich
  - bei Funktionsaufruf können immer nur von rechts die Argumente weggelassen werden

    ```
    void g(float, int = 1, char = 'a');

    void h()
    {
        g(1.0, 2, 'x');
        g(2.0, 3);
        g(3.0);
    }
    ```

- Funktionsdefinition

  - schon gesehen, der in der Deklaration angegebene Funktionskopf muß mit dem in der Definition übereinstimmen

    Beispiel:

    ```
    int max(int, int);        // Deklaration


    int max(int a, int b)     // Definition
    {
       if (a > b) return a;
       else return b;
    }
    ```

  - wenn eine Funktion nicht vom Typ `void` ist, dann muß sie mindestens eine `return` $<$ *wert* $>$ Anweisung besitzen, wobei der Typ von $<$ *wert* $>$ gleich dem Typ der Funktion sein muß, ggf. findet implizite Typkonvertierung in den Typ der Funktion statt
  - `void`-Funktionen können `return`-Anweisungen besitzen; aber ohne $<$ *wert* $>$-Angabe

- Bei Aufruf einer Funktion werden die Parameter in der Form call-by-value an die gerufene Funktion übermittelt
- wichtig: die Übergabe der Argumente erfolgt in Form einer Initialisierung der Parameter ⇒ die Parameter der Funktion werden mit den bei Funktionsaufruf angegebenen Argumenten initialisiert!
- Gültigkeitsbereich

  ```
  void f(int a)
  {
     float a;           //unzulaessig; Redefinition von a
     ...
  }
  ```

  die Namen der Parameter gehören zum Gültigkeitsbereich des Funktionsblocks

2.5.3 Zeiger

- für die meisten Typ T ist
  $$T*$$
  der Typ "Zeiger auf Objekt des Typs T"
- eine Definition der Form
  $$T * name$$
  definiert den Bezeichner $< name >$ als Zeiger auf ein Objekt des Typs T
- eine Variable dieses Typs kann also die Adresse eines Objektes aufnehmen

  Beispiel:

  ```
  int* pi;   // pi ist ein Zeiger auf ein Objekt des Typs int

  int i;


  pi = &i;   // pi erhaelt den Wert der Adresse von i
  ```
- der Operator `&` bildet die Adresse eines Objektes, er ist für alle Typen implementiert
- die fundamentale Operation mit einem Zeiger ist dessen Dereferenzierung (Zugriff auf das Objekt, auf das er zeigt)

  Beispiel:

  ```
   i = 3;
  *pi = 4;     //modifiziert den Wert von i

  if (*pi) {
  ```

```
    ...
}
```

- besondere Bedeutung hat die Zuweisung des Wertes 0 an einen Zeiger:

  ```
  pi = 0; // dieser Zeiger zeigt auf kein Objekt
  ```

- die Vereinbarung eines Zeigers kann wie folgt erweitert werden:

  $$T * qualifikatorname$$

  wobei für $< qualifikator >$ die Schlüsselworte

  ```
  const
  volatile
  ```

  stehen können
  der Typ der vereinbarten Variablen ist dann {`const`, `volatile`} Zeiger auf Objekt des Typs T

- der Wert eines `const`-Zeigers kann nicht verändert werden, nur Initialisierung ist möglich

  ```
  int i, j;
  int* const cpi = &i;

  cpi = &j;       //Fehler: Zuweisung an konstanten Zeiger
  *cpi = 3;       // ok.
  ```

- das Schlüsselwort `volatile` ist ein Hinweis für der Compiler, Zugriffe auf ein mit diesem Schlüsselwort vereinbartes Objekt nicht zu optimieren, da es möglich ist, daß auf das Objekt zugegriffen wird, ohne daß es der Compiler festellen kann (dieses Schlüsselwort wird allerdings von vielen Compilern noch nicht unterstützt)

- besondere Beziehungen bestehen zwischen Zeigern und Feldern

  ```
  char string[20];
  ```

  es gelten folgende Entsprechungen:
  `string` ≡ `&string[0]`
  `*string` ≡ `string[0]`
  `*(string+1)` ≡ `string[1]`

  der Name eines Feldes kann als Zeiger auf das erste Feldelement betrachtet werden
  per Definition gilt:

  ```
  name[index] == *(name +(index))
  ```

- diese Beziehung gilt für alle Typen

- Rechnen mit Zeigern

```
#include <iostream.h>

char* alpha="abcdefghijklmnopqrstuvwxyz";

int main()
{
   char *p = alpha;
   char c;

   cout << "\nVersion 2 mit Zeigern\n";
   while((c = *p) != 0) {
      cout << c
           << " = " << dec << int(c)
           << " = 0" << oct << int(c)
           << " = 0x" << hex << int(c)
           << "\n";
      p = p +1;
   }
}
```

- Zeiger auf Funktionen

```
   int f();

   int (*pfint)();
   pfint = f;          //Name der Funktion steht fuer deren Adresse
                       //oder deutlicher (also besser):
   pfint = &f;         //Anwendung des Adressoperators
                       //Aufruf:
   int y = pfint();    //nur in C++ ohne Dereferenzierung
                       //oder deutlicher (also besser):
   int x = (*pfint)(); //einzige Moeglichkeit in C:
```

  eigentlich ist diese Betrachtungsweise konsequenter als in C: der Aufruf einer Funktion wird interpretiert als Anwendung des Funktionsaufrufoperators `()` auf einen Funktionsnamen und damit auf die Adresse einer Funktion

2.5.4 Referenzen

- für einen Typ T ist

$$T\&$$

  der Typ "Referenz auf ein Objekt des Typs T"

- eine Vereinbarung der Form

$$T\&name$$

vereinbart den Bezeichner *name* als Referenz auf ein Objekt vom Typ T

- eine Referenz ist ein alternativer Name für ein Objekt
- hauptsächliche Verwendung:
  - Übergabe von Argumenten *call-by-reference*
  - Spezifikation von Operationen auf Klassen
- Beispiel:

```
int i = 1;
int& ri = i;    // ri und i beziehen sich auf dasselbe Objekt
int x = ri;     // x = 1

ri = 2;         // i = 2
```

- Was machen folgende Funktionen ?

```
#include <iostream.h>

void swap1(int a, int b)
{
   int temp = a;
   a = b;
   b = temp;
}

void swap2(int * const a, int * const b)
{
   int temp = *a;
   *a = *b;
   *b = temp;
}

void swap3(int &a, int &b)
{
   int temp = a;
   a = b;
   b = temp;
}

int main()
{
   int i = 1, j = 2;

   swap1(i, j);
   cout << "i = " << i << " j = " << j << "\n";
```

```
    swap2(&i, &j);
    cout << "i = " << i << " j = " << j << "\n";
    swap3(i, j);
    cout << "i = " << i << " j = " << j << "\n";
}
```

- Funktionen, die Referenzen liefern

```
int& pos(int x)
{
    static int array[100];
    ...
    return array[x];     // liefert Referenz auf x. Feldelement
}

void f()
{
    int i;
    ...
    pos(i) = 0;          // Zugriff auf i. Feldelement
}
```

Vorsicht:

```
int & g()
{
    int i;
    ...
    return i;     // Fehler: Referenz auf automatische Variable
}
```

2.5.5 Konstanten und Enumerationen

Das Schlüsselwort `const` kann der Deklaration eines Objektes vorangestellt werden, wenn dieses Objekt keine Variable, sondern eine Konstante sein soll.
Beispiel:

```
int f();
int i;
const int ci = 1;
const double PI = 3.141592653589;
const (*zf)() = f;
const int *const cpci = &ci;
const int* pci = &ci;
int *const cpi = &i;
```

An Konstanten kann keine Zuweisung erfolgen, sie müssen initialisiert werden. Der Wert einer solchen Konstanten kann nicht geändert werden. Auch nicht über einen Zeiger !

```
ci = ci + 1;        // Fehler
*pci = 2;           // Fehler
pci = &i;           // ok.
int *pi = pci;      // Fehler
*cpi = 5;           // ok.
cpi = &ci;          // Fehler
```

häufige Verwendung:

```
int strlen(const char* c)
{
    int length = 0;
    while (*c)  {
            length = length + 1;
            c = c + 1;
   }
   return length;
}

const char* pc = "abcd";
pc[3] = 'x';                // Fehler
pc = "efgh";                // ok.

char *const cp = "abcd";
cp[3] = 'x';                // ok.
cp = "efgh";                // Fehler
```

- Enumerationen - Aufzählungstyp
  - Definition ganzzahliger Konstanten
    `enum Boolean { False, True};`
    $\uparrow_1$ $\uparrow_2$ $\uparrow_3$
    1 Das ist ein Typname!
    2 =0
    3 =1
  - Definition einer Variablen:
    (`enum`) `Boolean schalter`;
    - das Schlüsselwort `enum` kann weggelassen werden
    - `schalter` kann nur die Werte False (0) und True (1) annehmen
      → entspricht den Wahrheitswerten von Ausdrücken !
- es ist auch möglich, eine direkte Wertzuweisung an die Konstanten vorzunehmen:

  `enum { heute, gestern = -1, morgen = 1, übermorgen = 1};`
  ↑
  implizit = 0

  die zugewiesenen Werte müssen weder verschieden, noch aufsteigend, noch positiv sein

- anonyme Enumerationen

  `enum {Asm, Auto, Break, Case};`

  entspricht:

  ```
  const Asm = 0;
  const Auto = 1;
  const Break = 2;
  const Case = 3;
  ```

2.5.6 Strukturen

- Zusammenfassung von Objekten unterschiedlicher Typen
  vgl. RECORD in Pascal

- später den Begriff erweitern, zunächst nur im Sinne von C

- eine Struktur definiert in C++ einen neuen Typ
  Bsp.:

  ```
  struct Datum {
     short jahr;
     char  monat[4];
     short tag;
  };
  ```

  `Datum` ist ein Typbezeichner, es wird dabei kein Speicherplatz reserviert, sondern lediglich die Typen der zur Struktur gehörenden Komponenten festgelegt (trotzdem spricht man von einer Typdefinition)

- Instanzen einer Struktur werden vereinbart wie gewöhnliche Objekte (Variablen)

  ```
  Datum heute, morgen;
  ```

- jede Instanz einer Struktur verfügt über die Komponenten (Instanzvariablen) `jahr`, `monat` und `tag`

- der von einer Strukturinstanz belegte Speicherplatz ergibt sich nicht zwangsläufig aus der Summe des Speicherplatzbedarfs der zugehörigen Instanzvariablen; aufgrund der Ausrichtung (alignment) der Strukturkomponenten entstehen ggf. „Lücken“ zwischen den Instanzvariablen (das ist maschinenabhängig) ⇒ `sizeof`-Operator

- Instanzvariablen einer Struktur können von beliebigem Typ sein, insbesondere auch Strukturen (jedoch kann eine Struktur nicht in sich selbst enthalten sein)
  Bsp.:

```
struct Adresse {
   short plz;
   char  ort[20];
   char  strasse[20];
   short hnr;
};

struct Person {
   char    name[20];
   char    vorname[20];
   Datum   geb_datum;
   Adresse anschrift;
};
```

- für die Arbeit mit Strukturinstanzen existieren die Operatoren `.` und `->`
  Bsp.:

```
Person max, moritz, leute[10];

heute.jahr = 1992;
strcpy(heute.monat, "Feb");
heute.tag = 11;

leute[0].geb_datum.jahr = heute.jahr;

int i = 0;
Person* pp = leute;

while (i < 10) {
   pp->geb_datum = heute;
   pp = pp + 1;
   i = i + 1;
}
```

  - der Operator `.` stellt eine Bezugnahme auf eine Instanzvariable einer Strukturinstanz her
  - da eine Zuweisung von Zeichenkettenkonstanten an Zeichenfelder in C++ nicht unterstützt wird, muß die Standardfunktion `strcpy` benutzt werden
  - der Ausdruck `leute[0].geb_datum.jahr` wird von links ausgewertet, es wird also auf die Instanzvariable `jahr` der Instanzvariablen `geb_datum` des 0. Feldelements des Feldes `leute` zugegriffen
  - der Operator `->` ist eine Kurzschreibweise für

```
(*pp).geb_datum = heute;
```

dieser Operator dient der Bezugnahme auf Instanzvariablen von Strukturinstanzen, die über Zeiger adressiert werden
dieses Beispiel demonstriert auch die Zuweisung kompletter Strukturinstanzen

2.6 Spezielle Operatoren

- dieser Abschnitt behandelt (in Kurzform) sämtliche (bisher nicht behandelten) Operatoren von C++
- gerade die Vielzahl von Operatoren erscheint für den Anfänger zunächst verwirrend, macht aber gegenüber anderen Programmiersprachen einen wesentlichen Vorteil aus, da eine sehr kompakte Schreibweise unterstützt wird
- man sollte die Operatoren als Angebot der Sprache auffassen und bei der Programmierung darauf achten, daß man die Lesbarkeit von Programmen nicht zerstört (insbesondere die Inkrement-/Dekrementoperatoren und der Entscheidungsoperator sind dafür gute Kandidaten)

2.6.1 Inkrement/Dekrement

- der unäre Operator `++` dient dem Inkrementieren seines Operanden um den Wert 1
- analog dekrementiert der unäre Operator `--` seinen Operanden um den Wert 1
- die Operanden müssen Lvalue's (also irgendwas im Speicher) sein
- die Operanden können in der Präfix- und in der Postfixnotation verwendet werden

```
++<lvalue>          // Praefixnotation
--<lvalue>

<lvalue>++          // Postfixnotation
<lvalue>--
```

- der Wert des Ausdrucks ergibt sich
  - bei der Präfixnotation aus dem Wert des Operanden **nach** Inkrementierung bzw. Dekrementierung
  - bei der Postfixnotation aus dem Wert des Operanden **vor** Inkrementierung bzw. Dekrementierung
    Bsp.:

```
int x, y;

x = 3;
```

```
y = ++x;      // Praefix: x = x + 1; y = x

x = 3;
y = x++;      // Postfix: y = x; x = x + 1
```

- diese Operationen sind Operationen mit Nebeneffekt: der Haupteffekt besteht in der Berechnung des Ausdrucks, während die Inkrementierung bzw. Dekrementierung des Operanden den Nebeneffekt darstellt
- wird der Ausdruckswert nicht benötigt, so sind Präfix- und Postfixnotation gleichberechtigt
  Bsp.:

  ```
  int x;

  x = 3;
  x++;           // ist identisch zu ++x;
  ```

- Beispiel: Berechnung der Fakultät für natürliche Zahlen

  ```
  long fakultaet(int n)    // n! = 1 * 2 * ... * n; 0! = 1
  {
        long i, fak;

        fak = i = 1;
        while (++i <= n)
           fak = fak * i;
        return(fak);
  }
  ```

  der Vergleichsausdruck `++i <= n` wird folgendermaßen ausgewertet: der Teilausdruck `++i` besitzt den gleichen Wert wie `i` nach der Inkrementierung, deshalb geht dieser Wert in den Vergleich mit `n` ein; ist die Bedingung erfüllt, d.h. ist der Wahrheitswert des Vergleichsausdrucks TRUE, so wird der Körper der while-Schleife ausgeführt

- die Operatoren `++` und `--` können auch auf Zeiger angewendet werden
  Bsp.:

  ```
  int * pi;

  pi++;          //pi = pi + 1;
  ```

Aufgabe: Welche Werte besitzen `x` und `y` nach der if-Anweisung

a)
```
int i, x;
x = 1;
i = 0;
```

b)
```
int i, x;
x = 1;
i = 0;
```

```
if (--x)                    if (x--)
   x = i++;                    x = i++;
else                          else
   x = ++i;                    x = ++i;
```

Lösung:

```
x = 1   i = 1               x = 0   i = 1
```

2.6.2 Bit-Operatoren

- diese Operatoren ermöglichen die direkte Manipulation einzelner Bits
- sie sind insbesondere für maschinennahe Probleme interessant
- sie werden von den meisten Compilern effektiv auf entsprechende Maschinenbefehle umgesetzt
- binäre Operatoren:

```
&           // UND: Maskieren einzelner Bits
|, ^        // inklusives bzw. exklusives ODER
<<, >>      // Links- bzw. Rechtsverschiebung
```

- unärer Operator:

```
~           // Einerkomplement
```

- offenbar haben die Operatoren `<<` und `>>` in der Sprache C++ eine ganz andere Bedeutung, als bisher bekannt (Objekte: `cout, cin, cerr`) tatsächlich bietet C++ für Klassen die Möglichkeit, Operatoren mit neuer Bedeutung zu versehen (Verweis auf später)
- der Operator `&` ist bereits als Adreßbildungsoperator bekannt, der Compiler erkennt an der Anzahl der Operanden (und am Typ), welche Bedeutung des Operators gemeint ist
- an dieser Stelle nicht weiter auf diese Operatoren eingehen, da sie für den Kurs keine Bedeutung besitzen

2.6.3 Logische Operatoren

- binäre Operatoren

```
&&          // bedingtes logisches UND
||          // begingtes logisches ODER
```

- unärer Operator

```
!           // logische Negation
```

- Operator `!` liefert den Wert 1, wenn der Operand den numerischen Wert 0 besitzt, ansonsten den Wert 0
- enthält ein Ausdruck die bedingten logischen Operatoren `&&` bzw. `||`, so erfolgt die Auswertung (garantiert!) von links nach rechts so lange, bis der Wahrheitswert des Gesamtausdrucks eindeutig festgestellt werden kann
- ein logischer UND-Ausdruck besitzt demzufolge den Wert 1 (und damit den Wahrheitswert True) genau dann, wenn beide Operanden einen Wert verschieden von 0 besitzten
  der rechte Operand wird dabei nur dann ausgewertet, wenn der linke Operand bereits verschieden von 0 ist
- analog gilt für einen logischen ODER-Ausdruck, daß dessen rechter Operand nur dann ausgewertet wird, wenn der Wert des linken Operanden gleich 0 ist
- Beispiel:

```
int letter (char c)
{
    return c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z';
}
```

  diese Funktion liefert den Wert 1, falls das übergebene Zeichen ein Buchstabe ist und 0 sonst

2.6.4 Zusammengesetzte Zuweisungsoperatoren

- häufig auftretende Wertzuweisungen der allgemeinen Form

  lvalue `=` *lvalue op* `(` *ausdruck* `)`

  können folgendermaßen verkürzt geschrieben werden:

  lvalue op`=` `(` *ausdruck* `)`

  dabei kann für *op* einer der binären Operatoren

  `+, -, *, /, %, &, |, ^, <<, >>`

  angegeben werden
- Beispiele:

```
i += 2;          // i = i + 2
x *= y - 1       // x = x * (y - 1)   ! nicht: x = x * y - 1
```

2.6.5 Entscheidungsoperator

- Entscheidungsoperator verknüpft drei Ausdrücke:

  $ausdruck_1$ `?` $ausdruck_2$ `:` $ausdruck_3$

- Wert des Gesamtausdrucks ist vom Wahrheitswert des Gesamtausdrucks abhängig und entspricht dem Wert von $ausdruck_2$ im Fall True bzw. dem Wert von $ausdruck_3$ im Fall False

- es wird also nur einer der Ausdrücke $ausdruck_2$ und $ausdruck_3$ ausgewertet

- dieser Operator wird oft als abkürzende Schreibweise für Zweiwegentscheidungen der Form:

  `if (`$ausdruck_1$`)`
  $lvalue$ `=` $ausdruck_2$`;`
  `else`
  $lvalue$ `=` $ausdruck_3$`;`

  verwendet:

  $lvalue$ `=` $ausdruck_1$ `?` $ausdruck_2$ `:` $ausdruck_3$

- Bsp.:

  ```
  if (x > y)
       max = x;
  else
       max = y;
  ```

  kann verkürzt geschrieben werden:

  ```
  max = x > y ? x : y;
  ```

- Beachten: die Konstruktion `max = x > y ? x : y` ist ein Ausdruck und kann deshalb wie jeder andere Ausdruck verwendet werden
  Bsp.:

  ```
  int abs(int x)              // Absolutbetrag von x
  {
     return(x < 0 ? -x : x);
  }
  ```

  der Wert des `return`-Ausdrucks ist `-x` (bzw. `x`), wenn der Argumentwert `x` negativ (bzw. nichtnegativ) ist.

2.6.6 Explizite Typkonvertierung

- erinnern: in einigen Fällen führt der Compiler selbstständig Typkonvertierungen durch

- manchmal notwendig: Mechanismus zur expliziten Konvertierung eines Wertes von einem Typ in einen anderen

- zwei Möglichkeiten:

  1. cast-Operator

```
int i = 4;
double d;
d = (double)i;
```

diese Variante funktioniert für alle Typen, auch abgeleitete:

```
struct S1 {
   int a;
   float b;
};

struct S2 {
   int x;
   float y;
   char c;
};

void f()
{
   S1  s1v;
   S2  s2v;
   S1* s1p = &s1v;
   S2* s2p = &s2v;

   // jetzt moeglich:
   s1v.a = 1; s1v.b = 2.0;  s1p -> a++;  s1p -> b = 3.0;
   s2v.x = 4; s2v.y = 5.0;  s2v.c = 'a';  s2p ->x--;
   // nicht moeglich:
   //s1p = s2p;
   //s2p = s1p;
   // jedoch
   s1p = (S1*)s2p;
   s2p = (S2*)&s1v;    //mit aeusserster Vorsicht !
   // folgendes funktioniert dann:
   s1p -> a = 4;       //Zugriff auf s2v.x
   s1p -> b = 3.0;     //Zugriff auf s2v.y
                       //auf s2v.c kann ueber s1p nicht
                       //zugegriffen werden
   s2p -> x = 1;       //Zugriff auf s1v.a
   s2p -> y = 3.0;     //Zugriff auf s1v.b
   s2p -> c = 'b';     //wohin soll das gehen ?
}
```

2. Typ als Funktionsnamen

```
int i = 2;
double d = double(i);
```

   – funktioniert nur mit Grunddatentypen; Klassen, für die entsprechende Typkonvertierungsoperatoren (Konstruktoren) definiert sind (Verweis auf später) und sog. `typedef`-Namen `typedef` führt einen neuen Typnamen ein

Bsp.:

```
typedef int INTEGER;

INTEGER i, j;       //vereinbart i, j als int

typedef  S1* S1P;
typedef  S2* S2P;

 s1p = S1P(s2p);
 s2p = S2P(&s1v);
```

- allgemein:
  Die Vereinbarung eines neuen Typs mittels `typedef` erfolgt so wie die Deklaration eines Namens, der Name ist dann der neue Typname:

```
typedef void (* PVFV)();    //Zeiger auf Fkt.:  void (*)()
typedef float *A4PFl[4];    //Feld von 4 Zeigern auf float
typedef int (*A4PIFV[4])(); //Feld von 4 Zeigern auf Fkt.

PVFV p;         // p ist ein Zeiger auf eine Fkt. vom Typ void
A4PFl array1;
A4PIFV array2;
PVFV f();       //void (*f())();
                //f ist eine Funktion, die einen Zeiger auf eine
       //Funktion: void (*)() liefert
```

2.6.7 Kommaoperator

- zwei durch Komma getrennte Ausdrücke werden von links nach rechts ausgewertet

  $ausdruck_1$, $ausdruck_2$

```
while (x = -i, y += x) {      |x = -i;
           ...                |while(y+=x) {
                              |  ...
                              |  x=-i;
}                             |}
```

- zunächst erhält `x` den vorzeichenverkehrten Wert von `i` und anschließend wird dieser Wert zu `y` addiert

- der Wert des Gesamtausdrucks ergibt sich aus dem neuen Wert von `y`, d. h. der Block wird abgearbeitet, solange der jeweils neue Wert von `y` verschieden von Null ist

2.6.8 Speicherverwaltung

- Operatoren `sizeof`, `new`, `delete`
- `sizeof`
  - bestimmt die Menge an belegtem Speicherplatz von einzelnen Objekten bzw. von Typen, dabei ist auf jeder Maschine laut Definition `sizeof(char) == 1`
  - Beispiel:

    ```
    char c;
    int i;

    int sizeof_c = sizeof(c);  // 1
    int sizeof_i = sizeof(i);  // 32-Bit-Rechner: 4, 16-Bit-Rechner: 2
    ```

    funktioniert auch für komplizierte Typen

    ```
    int sizeof_Datum = sizeof(Datum);
    ```

- C++ bietet Operatoren an, die es erlauben, Objekte zur Laufzeit des Programms zu erzeugen und zu beseitigen
- mit diesen Operatoren kann also die Lebensdauer eines Objektes explizit gesteuert werden
- die so erzeugten Objekte liegen im Freispeicher (Heap)
- Operatoren:

  `new` *typname*   `new` *typname*`[]`
  `delete` *name*   `delete[]` *name*

  werden verwendet:

  ```
  int* pi = new int;
  int* pj = new int(5);     //neues Objekt wird mit 5 initialisiert
  int* pf = new int[5];     //Feld von 5 int's

  delete pi;                 //Speicherplatz des ueber pi erreichbaren
                               Objekts wird freigegeben
  delete pj;
  delete[] pf;
  ```

  Beispiel:

  ```
  #include <iostream.h>
  #include <string.h>

  char* save_string(const char * p)
  {
      char* s = new char[strlen(p)+1];
      strcpy(s, p);
  ```

```
    return s;
}

int main()
{
    char* hallo = save_string ("Hello, world!\n");
    cout << hallo;
    delete [] hallo;
}
```

- `strlen` und `strcpy` sind Funktionen zur Bearbeitung von Zeichenketten, sie gehören neben anderen derartigen Funktionen zur Standardbibliothek von C
  durch `#include <string.h>` werden diese Funktionen bekannt gemacht

2.6.9 Zusammenfassung/Vorrang und Assoziativität

⇒ Folie (Schlüsselworte und Operatoren)

- Vorrang und Assoziativität geben an, welche Operanden an welche Operatoren gebunden werden, wenn in einem Ausdruck verschiedene Operatoren auftreten

- dabei bindet zuerst der Operator mit dem höchsten Vorrang seine Operanden

- sind in einem Ausdruck mehrere Operatoren mit dem gleichen Vorrang, so entscheidet die Assoziativität, welcher Operator welche Operanden bindet, d.h. ob bei der Bindung der Operanden an die Operatoren innerhalb des Ausdrucks von links oder von rechts vorgegangen wird

- Vorrang und Assoziativität haben keinen Einfluß auf die Reihenfolge der Berechnung der einzelnen Teilausdrücke, das ist von der Compilerimplementation abhängig

- i. allg. braucht man diese Reihenfolge auch nicht zu beachten, jedoch kann es dann zu Problemen führen, wenn diese Ausdrücke Nebeneffekte enthalten

- laut Sprachdefinition besitzt ein Ausdruck einen undefinierten Wert, wenn sich der Wert einer Variablen in dem Ausdruck durch einen Nebeneffekt verändert und dieselbe Variable an einer anderen Stelle im gleichen Ausdruck benutzt wird
  Bsp.:

  ```
  x = f(y) + g(y++);

  a = h(x, g(++x));
  ```

- der Compiler kann Ausdrücke, die assoziative und kommutative Operatoren (`+, *, |, ^, &`) enthalten, auch dann umordnen, wenn einzelne Teilausdrücke explizit geklammert sind, d.h. es ist nicht möglich, durch Klammerung eine bestimmte Auswertungsreihenfolge zu erzwingen, die Klammerung beeinflußt lediglich die Bindung der Operanden an die Operatoren

2.7 Ablaufsteuerung II

2.7.1 for-Anweisung

- Syntax der for-Anweisung

  `for(`*initialisierung*`;` $ausdruck_1$`;` $ausdruck_2$ `)`
  anweisung

  initialisierung ist eine Definition oder ein Ausdruck
  $ausdruck_1$ ist der Wiederholungstest, wobei gilt: *anweisung* wird ausgeführt, wenn $ausdruck_1$ den Wahrheitswert True besitzt
  $ausdruck_2$ ist die Wiederinitialisierung und wird meist verwendet, um Objekte zu verändern, die in der *initialisierung* stehen

- Beispiel: Verarbeitung eines Feldes

```
int feld[20];

for(int i = 0; i < 20; i++)
    feld[i] = 0;
```

  die Angabe `int i = 0` ist in C nicht möglich
  Achtung: der Gültigkeitsbereich der Variablen `i` erstreckt sich auf den gesamten Block, zu dem die `for`-Anweisung gehört[2]

- diese `for`-Anweisung entspricht der `while`-Schleife

```
int i = 0;
while(i < 20) {
    feld[i] = 0;
    i++;
}
```

- diese Beziehung gilt allgemein, solange innerhalb des Schleifenkörpers keine `continue`-Anweisung auftritt

- Beispiel: Funktion, die den Index des ersten Auftretens eines Zeichens in einer Zeichenkette liefert (bzw. -1, falls das Zeichen nicht enthalten ist)

[2]nicht möglich ist folgende Konstruktion:
`for(int i = 0, j = 0; ...)`

```
int index(const char* z, char c)
{
    int i;
    for (i = 0; z[i] != c && z[i] ; i++) ;
    return (z[i] == c ? i : -1);
}
```

- Aufgabe: Ermitteln des letzten Auftretens eines Zeichens in einer Zeichenkette (bzw. -1, falls das Zeichen nicht enthalten ist)
  Lösung:

```
int rindex(const char* z, char c)
{
    int i = 0;

    for (i = 0; z[i]; i++) ; // ans Ende der Zeichenkette
    for ( ; i >= 0; i--)
       if (z[i] == c)
          return (i);
    return(-1);
}
```

2.7.2 switch-Anweisung

- `switch`-Anweisung dient der Mehrwegauswahl
- Syntax:

  `switch(` *ausdruck* `) {`
      `case` $konst\_ausdruck_1$ `:`
          $anweisungs\_liste_1$
          ...
      `case` $konst\_ausdruck_i$ `:`
          $anweisungs\_liste_i$
      `default:`
          $anweisungs\_liste_d$
      `case` $konst\_ausdruck_{i+1}$ `:`
          $anweisungs\_liste_{i+1}$
          ...
      `case` $konst\_ausdruck_n$ `:`
          $anweisungs\_liste_n$
  `}`

- $anweisungs\_liste_i$ ist ein Folge von Anweisungen, kann auch leer sein
- es können fehlen
  - der `default`-Zweig,
  - die `case`-Zweige vor `default` oder

  - die `case`-Zweige nach `default`

- Konstantenausdrücke müssen voneinander verschieden sein
- *ausdruck* muß einen Integer-Wert liefern
- der Wert des `switch`-Ausdrucks wird so lange mit den einzelnen `case`-Werten verglichen, bis Gleichheit festgestellt wird; in dem Fall beginnt die Abarbeitung bei der zu dem `case`-Zweig gehörenden Anweisungsfolge
- am Ende eines `case`-Zweiges wird die Abarbeitung der `switch`-Anweisung nicht automatisch beendet, sondern es werden die Anweisungen des darauffolgenden `case`- oder `default`-Zweiges abgearbeitet
- die `case`- bzw. `default`-Marken stellen also so etwas wie Einsprungstellen in die gesamte Anweisungsfolge der `switch`-Anweisung dar
- zum Abbruch der Abarbeitung einer Anweisungsfolge ist die `break`-Anweisung zu verwenden (siehe nächster Abschnitt)
- stimmt keiner der `case`-Werte mit dem `switch`-Wert überein, so beginnt die Abarbeitung mit dem `default`-Zweig; enthält dieser Zweig kein `break`, so werden ggf. nachfolgende `case`-Anweisungsfolgen ebenfalls abgearbeitet
- Beispiel (Ersetzen mehrerer Leerzeichen und Tabulatoren in einem Text durch ein Leerzeichen):

```
#include <iostream.h>

int main()
{
    char c;
    int leer_folge = 0;

    while (cin.get(c))
       switch (c) {
          case '\t':
          case ' ' :
             if(!leer_folge) {
                leer_folge = 1;
                cout.put(' ');
             }
             break;
          default:
             leer_folge = 0;
             cout.put(c);
             break;   // eigentlich nicht notwendig
       }
 }
```

2.7.3 break, continue, goto

- diese Anweisungen dienen der unbedingten Steuerungsübergabe
- `break` bricht die Abarbeitung einer unmittelbar übergeordneten `switch-`, `while-`, `do-while-` bzw. `for`-Anweisung ab
- Beispiel: Verlassen einer unendlichen `for`-Schleife

```
for (;;) {
     ...
     if ( ... )
          break;
     ...
}
```

- siehe auch `switch`-Anweisung
- `continue` leitet die nächste Iteration der umgebenden `while-`, `do-while-`, bzw. `for`-Schleife ein
- bei `while-` bzw. `do-while` wird der nächste Schleifentest durchgeführt und bei `for` wird die Wiederinitialisierung abgearbeitet
- die Anweisung

```
goto <marke>;
```

bewirkt die Fortsetzung der Abarbeitung bei einer markierten Anweisung

```
<marke>: <anweisung>
```

- für Marken gelten die gleichen Bildungsregeln wie für andere Bezeichner
- Sprünge sind nur innerhalb von Funktionsblöcken erlaubt

2.8 Initialisierung und Zuweisung

- Unterschied beachten:
  Initialisierung (C-Stil):

```
int i = 0;
```

Initialisierung (C++-Stil, ab C++ 3.0):

```
int i(0);
```

Zuweisung:

```
int i;
. . .
i = 0;
```

- Achtung:
  C++-Stil unterscheidet zwar syntaktisch sehr gut Initialisierung und Zuweisung, birgt aber folgende Gefahr in sich:

  ```
  int i(); // deklariert eine Funktion i, die int liefert
  ```

3 Klassen als ADT's

3.1 Der Klassenbegriff

3.1.1 Definition, Implementation und Verwendung

- erinnern: Klassen: ADT
- vereinen: Daten + dazugehörige Operationen
- Klassendefinition (=Typdefinition)

  class *name* {
     private:
        private_member_deklaration;
     protected:
        protected_member_deklaration;
     public:
        public_member_deklaration;
  };

  member:

  – Datendeklarationen
  – Funktionsdeklarationen (Methoden, Memberfunktionen)

- Beispiel:

```
class String {
private:                        // kann weggelassen werden
   char *s;                     // Instanzvariablen
   int size;                    // diese Daten besitzt jedes
                                // Objekt dieser Klasse
public:
   void set(const char *);      // oeffentliche Methoden
   int length();                // erlauben den Zugriff auf ein
                                // Objekt dieser Klasse
};                              // Semikolon nicht vergessen !
```

  – es können auch Methoden privat oder protected sein und es kann öffentliche Instanzvariablen geben
  – private Daten (Member):
    auf die so deklarierten Bezeichner darf nur in der Implementation der Methoden einer Klasse (oder friends dieser) zugegriffen werden, sie sind an der Schnittstelle der Klasse nicht sichtbar und können nicht verwendet werden
  – protected Daten (Member):
    auf die so deklarierten Bezeichner darf nur in der Implementation der Methoden einer Klasse (oder friends dieser) und in einer von dieser abgeleiteten Klasse (oder friends) zugegriffen werden

- eine solche Klassendefinition wird gewöhnlich in einem Headerfile untergebracht ⇒ String.h
- die Implementation der Methoden schreibt man in ein separates File ⇒ String.C (oder String.cc oder String.cpp, das ist vom Compiler abhängig)
- die Verwendung der Klasse erfolgt dann in einem dritten File ⇒ z.B. mainString.C
- Implementation: String.C
  hier Methoden implementieren: Funktionsdefinitionen
  bei den Implementationen muß die Bezugnahme Klasse - Methode hergestellt werden:

```
#include "String_1.h"
#include <string.h>

void String::set (const char * st) //die Methode set der
                                   //Klasse String
{
    s = new char [size = strlen(st) + 1];
                                   //size enthaelt Anzahl der char
    strcpy(s, st);
}

int String::length()
{
     return size - 1;
}
```

- Verwendung dieser Klasse: mainString.C

```
#include <iostream.h>
#include "String_1.h"

int main()
{
   String s1;  //Definition eines Objektes s1 der Klasse String

   s1.set("Hello, world !\n");
   int i = s1.length();
   cout << "Laenge: " << i << "\n";
}
```

- zur Bezugnahme auf die Methoden des Objektes wird der (bereits bekannte) Operator . benutzt

```
s1.length();
```

```
s1.set("Hello, world !\n");   // das ist eine Botschaft
```

`s1` – Adressat, an weches Objekt geht die Botschaft
`set` – Selektor, welche Methode soll das Objekt ausführen
`"Hello, world!\n"` – Argument, notwendige Information zur Ausführung der Methode

- als Operationen mit einem Objekt der Klasse String sind bisher nur die Methoden `set` und `length` vorgesehen, andere Bezugnahmen sind nicht möglich, z. B. gibt es keine(!) Möglichkeit an die Zeichenkette heranzukommen:

```
char * cp = s1.s;     //Fehler: Zugriff auf private Daten
```

- das gezeigte Programm wird so zwar funktionieren, aber es gibt ein paar offene Fragen:

1. Welchen Wert haben die Instanzvariablen nach der Zeile

```
String s1;
```

2. Was passiert bei

```
String s2;
int ls2 = s2.length();
```

3. Was passiert mit dem in `String::set` zugewiesenen Speicher ?
4. Was passiert bei einem zweiten Aufruf von `set` ?

```
s1.set("Good bye, world !\n");
```

5. Was passiert bei Verwendung weiterer Objekte der Klasse String?

```
String s3;
s3.set ("Guten Tag !\n"); //mit welchen Instanzvariablen
                          //size und s arbeitet set jetzt?
...
s3 = s1;          //geht das ueberhaupt? |\
                                         | Wo ist der Unterschied ?
String s4 = s1; //geht so etwas?         |/
```

6. Welche Möglichkeiten der Instanziierung einer Klasse gibt es noch ?

```
String * s5p = new String;      //geht das ?
```

7. Wie greift man auf ein solches Objekt zu ?
8. Kann man die Adresse eines solchen Objektes bilden, und was tut man dann damit?

```
String * s6p;
s6p = &s1;
```

9. Kann man Objekte einer Klasse als Instanzvariablen anderer Klassen verwenden?

```
class Name  {
    String vorname;
    String familienname;
    ...
};
```

10. Gibt es konstante Objekte ?

```
const String s7;
```

11. Kann man Objekte vergleichen ?

```
if(s1 == s2);
...
```

man kann sich weitere derartige Fragen vorstellen, die letztendlich auf eine einzige Frage hinauslaufen:

Welche Unterschiede gibt es zwischen den Objekten der Grunddatentypen (int, char, ...) und Objekten von Klassen ?

In den folgenden Abschnitten werden die aufgezählten Fragen schrittweise beantwortet.

3.1.2 Konstruktoren und Destruktoren

- sind spezielle Methoden einer Klasse, die durch ihren Namen ausgezeichnet sind
- Konstruktoren heißen genau wie die Klasse
- Destruktoren heißen wie die Klasse, jedoch mit dem Zeichen ˜ vor dem Namen
- Konstruktoren:
  - belegen Instanzvariable mit initialen Werten
  - werden bei Instanziierung einer Klasse automatisch gerufen
- Destruktoren:
  - geben angeforderte Speicherbereiche frei, die über Instanzvariablen addressiert werden
  - werden bei Ende der Lebensdauer eines Objektes automatisch gerufen
- Konstruktoren/Destruktoren haben keinen Typ, liefern keinen Wert (auch nicht `void`)

```
class String {
private:
    char *s;
    int size;
```

```
public:
   String();  //Default-Konstruktor, wird gerufen bei String s1;
   String(const char*);
              //wird gerufen bei String s2("Hi noon!\n");
   ~String(); //nur ein Destruktor pro Klasse, immer parameterlos
   void set(const char*);
   int length();
   void print(); // neue Methode
};
```

- zwei verschiedene Konstruktoren (Methoden mit dem Name `String` ⇒ überladene Funktion -> Möglichkeit ein Objekt der Klasse `String` auf zwei unterschiedliche Arten zu erzeugen)

```
#include <iostream.h>
#include <string.h>
#include "String_2.h"

String::String()
{
   s = new char[1]; // Zuweisung (nicht Initialisierung)
   *s = 0;
   size = 0;        // an die Elemente
}

String::String(const char * st)
{
   s = new char [size = strlen(st) + 1];
   strcpy(s, st);
}

String::~String() {
   delete [] s;
}

void String::set(const char * st) {
   delete [] s;
   s = new char[size = strlen(st) + 1];
   strcpy(s, st);
}

int String::length() {
   return size ?  size-1 : 0;
}

void String::print() {
   cout << "[" << length() << "] ";
```

```
        if (s) cout << s; else cout << "(NULL)";
        cout << "\n";
    }
```

- jetzt funktionieren folgende Dinge:

```
    #include <iostream.h>
    #include "String_2.h"

    int main()
    {
        String s1;
        String s2("High");

        s1.print();
        s1.set("Hallo");
        s1.print();
        s2.print();
        s2.set("Bye");
        s2.print();
      } //hier endet die Lebensdauer von s1 und s2
        // => fuer s1 und s2 wird der Destruktor gerufen
```

Aufgabe: 1) Programm testen
2) Entwickeln einer Klasse Rechteck
3) Entwickeln char-Stack

Lösung zu 2)

File Rechteck.h

```
class Rechteck {
private:
    float a;     // Seitenlaenge a
    float b;     // Seitenlaenge b
public:
    Rechteck(float =.0, float =.0);
    void set(float, float);
    float flaeche();
    float umfang();
};
```

File Rechteck.C

```
#include "Rechteck_1.h"

Rechteck::Rechteck(float aa, float bb)
{
    set(aa,bb); // Fragwuerdig: Aufruf von
                // Methoden im Konstruktor
}
```

```
void Rechteck::set(float aa, float bb)
{
   a = aa;
   b = bb;
}

float Rechteck::flaeche()
{
   return a*b;
}

float Rechteck::umfang()
{
   return 2.*(a+b);
}
```

File mainRechteck.C

```
#include <iostream.h>
#include "Rechteck_1.h"

int main()
{
   Rechteck r1(3.0,4.0);
   Rechteck r2;

   cout << "Flaeche r1: " << r1.flaeche()
        << " Umfang r1: "  << r1.umfang() << "\n";
   cout << "Flaeche r2: " << r2.flaeche()
        << " Umfang r2: "  << r2.umfang() << "\n";
   r2.set(4.5,5.0);
   cout << "Flaeche r2: " << r2.flaeche()
        << " Umfang r2: "  << r2.umfang() << "\n";
 }
```

Lösung zu 3)

File CharStack.h

```
class CharStack {
private:
   short size;
   char  *mem;         // Anfang des zugehoerigen Speichers
   char  *top;         // aktuelles Element
public:
   CharStack(short s);
   ~CharStack();
   char pop();
   void push(char c);
   int empty();
```

```
   int full();
};
```

File CharStack.C

```
#include "CharStack.h"

CharStack::CharStack(short s)
{
   top = mem = new char[size = s];
}

CharStack::~CharStack()
{
   delete[] mem;
}

void CharStack::push(char c)
{
   if(!full())
      *top++ = c;
}

char CharStack::pop()
{
   if(!empty())
      return *--top;
   return 0;
}

int CharStack::empty()
{
   return mem == top;
}

int CharStack::full()
{
   return mem + size == top;
}
```

File mainCharStack.C

```
#include <iostream.h>
#include <string.h>
#include "CharStack.h"

const MAX = 20;

int main()
{
```

```
    CharStack cs(MAX);

    char word[MAX+1];

    cout << "Wort (max. " << MAX << " Zeichen): ";
    cin >> word;
    for(unsigned i = 0; i < strlen(word); i++)
       cs.push(word[i]);
    while(!cs.empty())
       cout << cs.pop();
    cout << "\n";
 }
```

3.1.3 Bilden von Objekten, Gültigkeit, Lebensdauer

Erweitern der Konstruktoren/Destruktoren ⇒ diese Methoden sollen sich bei ihrer Abarbeitung melden

```
String::String()
{
        ...
        cout << "Defaultkonstruktor String::String()\n";
}

String::String(const char* st)
{
        ...
        cout << "Konstruktor String::String(" << s << ")";
}

String::~String()
{
        cout << "Destruktor String::~String() s = " <<  s << "\n";
        ...
}
```

- global sichtbare Objekte
  Definition außerhalb von Funktionen
  File StringsUsage.h

  ```
  #include  "String_2.h"

  extern String globalString;  //Deklarationen
  extern void S1print();
  extern void S2print();
  ```

  File StringsUsage1.C

  ```
  #include  <iostream.h>
  ```

```
#include  "StringsUsage.h"

String globalString("Hallo");  //Definition

void S1print()
{
    cout << "StringsUsage1.C: S1print()\n";
    globalString.print();
}
```

File mainStrings.C

```
#include <iostream.h>
#include "StringsUsage.h"

void S2print()
{
    cout << "mainStrings.C: S2print()\n";
    globalString.print();
    globalString.set("Bye");
}

int main()
{
   S1print();
   S2print();
   S1print();
}
```

Frage: Wann wird das Objekt `globalString` angelegt und wann wird es beseitigt?

Aspekte:

- die Zeile
  `extern String globalString;`
  im File StringsUsage.h macht den Bezeichner `globalString` in allen Files bekannt, die das File StringsUsage.h mittels `#include` einschließen
- Objekte, die nur in bestimmten Files bekannt sein sollen, werden einfach nicht in allgemeinen Headerfiles deklariert, damit sind die Bezeichner nicht bekannt

- lokal sichtbare Objekte
  – automatische Objekte
  Definition innerhalb von Funktionen
  File autoString.C

```
#include <iostream.h>
```

```
#include  "String_2.h"

void  f(const char * st)
{
    String s1(st);
    s1.print();
}

int main()
{
    f("Automatisches");
    f("Objekt");
}
```

Frage: Was passiert beim Aufruf bzw. Beendigung von f() ?

– statische Objekte

```
#include <iostream.h>
#include "String_2.h"

void  f(const char * st)
{
    static String s1("Statisches");

    s1.print();
    s1.set(st);
}

int main()
{
    f("Objekt");
    f("Ende");
}
```

Frage: Was passiert? Welche Abläufe, was wann?

- Objekte im Freispeicher

```
#include <iostream.h>
#include "String_2.h"

String * f(const char * str)
{
   return new String(str);
}

int main()
{
```

```
   String* ps = f("Objekt");
   ps->print();
   cout << f("im Freispeicher")->length() << "\n";
   delete ps;
}
```

Diskutieren: Abläufe, was passiert mit dem Objekt, welches beim zweiten Aufruf von f() angelegt wird?

- Konstante Objekte

  ```
  #include <iostream.h>
  #include "String_3.h"

  int main()
  {
     const String cs("Konstante");
     cs.print();
  }
  ```

  Warum funktioniert das nicht ?

  - ein Objekt kann dann als konstantes Objekt betrachtet werden, wenn es nicht möglich ist, seine Instanzvariablen zu verändern
  - der Compiler kann jedoch aus der Klassendefinition nicht wissen, welche Methoden modifizierenden Charakter haben und deshalb für konstante Objekte nicht angewendet werden können
    ⇒ Klassendefinition verändern:

    ```
    class String {
       ...
    public:
           ...
           int length() const;
     void print() const;
    }
    ```

  - entsprechend muß die Implementation der Methode modifiziert werden

    ```
    int String::length () const
    {
         return size ? size - 1  : 0;
    }
    ```

  - der Compiler kann prüfen, ob die Methode tatsächlich keinen modifizierenden Zugriff zu den Instanzvariablen durchführt

- Objekte als Instanzvariablen anderer Klasse (Memberobjekte bzw. Aggregationen)

```
#include "String_3.h"

class Name {
private:
   String vorname;
   String familienname;
public:
   Name(const char * vn, const char * fn);
   void print() const;
};
```

```
#include <iostream.h>
#include "Name.h"

Name::Name(const char * vn, const char * fn)
  : vorname(vn), familienname(fn)
{ }

void Name :: print() const
{
   vorname.print();         //Delegation an Aggregat
   familienname.print();
}
```

die syntaktische Konstruktion

```
Name::Name(const char * vn, const char * fn)
 : vorname(vn), familienname(fn)
```

bewirkt die Initialisierung der Instanzvariablen (Memberobjekte) `vorname` und `familienname` durch den Aufruf der entsprechenden Konstruktoren vor der Abarbeitung des Rumpfes des Konstruktors
`Name::Name(const char* vn, const char* fn)`
diese syntaktische Konstruktion bringt den Initialisierungscharakter stärker zum Ausdruck als z.B. eine Zuweisung an eine Instanzvariable innerhalb eines Konstruktors (in diesem Beispiel wäre eine Zuweisung sogar falsch) Zuweisungen an Instanzvariablen sollten deshalb in Konstruktoren überhaupt nicht verwendet werden, statt dessen sollte man diese Initialisierungsschreibweise verwenden

```
#include "Name.h"

int main()
{
    Name nikolaus("Vaeterchen", "Frost");

    nikolaus.print();
}
```

Aufgabe: Untersuchen der Aufrufreihenfolge der Konstruktoren / Destruktoren

- Felder von Objekten

```
#include "String_3.h"

int main()
{
   String strings[5];  //Es ist ein Defaultkonstruktor
                       //erforderlich, sonst funktionierts
                       //nicht; es ist nicht moeglich, Feld-
                       //elemente zu initialisieren
   strings[0].set("Null");
   strings[1].set("Eins");
   strings[2].set("Zwei");
   strings[3].set("Drei");
   strings[4].set("Vier");
   for (int i = 0; i < 5; i++)
      strings[i].print();
}
```

3.1.4 Friends

- der Zugriff zu den privaten Member ist nur den Methoden einer Klasse erlaubt
- das Schlüsselwort `friend` ermöglicht die gezielte Verletzung dieses Sicherheitsmechanismus
- dazu muß in der entsprechenden Klasse erklärt werden, wer außerdem Zugriff zu den privaten Member der Klasse erhalten soll
- `friend`s einer Klasse können sein:
  - ausgezeichnete Funktionen
  - ausgezeichnete Methoden anderer Klassen
  - alle Methoden einer anderen Klasse

Bsp.:

```
class A {
  ...
public:
     ...
     void methode1();
};

class B {
```

```
  ...
public:
      ...
      void methode1();
      void methode2();
      ...
};

class C {
      friend void function();
      friend void A::methode1();
      friend B;
      int x;
public:
      void set(int i);
      int  get();
};

C c;
```

Zugriff zu `c.x` besitzen:

- alle Methoden der Klasse C:

  ```
  void C::set(int);
  int C::get();
  ```

- die Funktion

  ```
  void function();
  ```

- die Methode `methode1()` der Klasse `A`:

  ```
  void A::methode1();
  ```

- alle Methoden der Klasse B

- der friend-Mechanismus sollte nur in bestimmten Fällen angewendet werden:
  - Funktionen, die zu den privaten Member mehrerer Klassen zugreifen müssen
  - Operatorfunktionen
  - rein private Klassen (alle Member der Klasse sind privat)
- die Stellung der `friend`-Deklarationen innerhalb einer Klassendefinition hat keinen Einfluß auf die Sichtbarkeit der so deklarierten `friends`

3.1.5 Selbstreferenz

- die Methoden einer Klasse arbeiten jeweils mit den Instanzvariablen des Objektes, für das sie aufgerufen wurden

- Beispiel:

```
class X {
   int m;
public:
   int get() const;
   void set(int);
};

int X::get() const
{
   return m;
}

X x1;
X x2;

void f()
{
   int a = x1.get();
   int b = x2.get();
   ...
}
```

  beim ersten Aufruf von `X::get()` infolge von `x1.get()` bezieht sich `m` auf `x1.m` und beim zweiten Aufruf auf `x2.m`

- für diese Bezugnahme existiert in jeder Methode ein (verborgener) Zeiger, der über das Schlüsselwort `this` angesprochen werden kann

- dieser Zeiger stellt die Bezugnahme zu dem Objekt her, für das die Methode gerufen wurde
  in jeder Methode einer Klasse `X` ist also ein Zeiger

```
   X *const this;         X::set()
```

  oder

```
   const X *const this;   X::get()
```

  implizit deklariert

- man könnte also auch schreiben:

```
int X::get() const
{
   return this->m;
}
```

- i.allg. wird `this` beim Zugriff auf die Member einer Klasse innerhalb von Methoden nicht verwendet

- `this` wird dort verwendet, wo mit Zeigern auf Objekten gearbeitet wird (Listenoperationen) oder wo eine Methode als Ergebnis das Objekt liefert, wofür sie aufgerufen wurde
  Bsp.:

```
class X {
   ...
public:
   ...
   X& method();
};

X& X::method()
{
   ...
   return *this;
}
```

3.2 Operatorfunktionen

3.2.1 Prinzipien und Möglichkeiten

- C++ bietet die Möglichkeit, die „Natürlichkeit“ der Programmiersprache aufrecht zu erhalten

- die Definition einer neuen Bedeutung für die meisten Operatoren der Sprache für Objekte von Klassen ist eine solche Möglichkeit

- tatsächlich ist es möglich, <u>alle</u> Operatoren bis auf
  .   ::   ?:   (.*)
  mit einer neuen Bedeutung zu versehen (zu überladen)

- dabei kann aber weder die Priorität noch die Ausdrucks-Syntax dieser Operatoren geändert werden; neue Operator-Zeichen können ebenfalls nicht definiert werden

- Operatorfunktionen tragen den Namen:
  `operator @`
  wobei `@` für den Operator steht

- eine Operatorfunktion muß entweder

- Methode einer Klasse (Memberfunktion) sein oder
- mindestens ein Objekt bzw. eine Referenz auf ein Objekt einer Klasse als Parameter übernehmen
- Ausnahmen: `new` und `delete`

- mit Ausnahme von `operator = ()` werden Operatorfunktionen vererbt
- bekannte Beziehungen zwischen Operatoren

  Bsp.: `++x;    ==    x += 1;`

  für Objekte von Grunddatentypen müssen nicht (aber sollten) aufrecht erhalten werden
- eine Operatorfunktion kann keine Standardargumente besitzen
- unäre Operatoren können als
  - Memberfunktionen ohne Parameter oder als
  - "normale" Funktionen mit einem Parameter

  vereinbart werden

  Interpretation

  `@x` und `x@` werden interpretiert als
  `x.operator @()` oder
  `operator @(x)`
- binäre Operatoren können als
  - Memberfunktionen mit einem Parameter oder als
  - "normale" Funktionen mit zwei Parametern

  vereinbart werden

  Interpretation

  `x @ y` wird interpretiert als
  `x.operator @(y)` oder
  `operator @(x, y)`
- für unäre und binäre Operatoren gilt: wenn beide Formen der Operatorfunktionen deklariert sind, entscheiden die sogenannten Argument-Matching-Regeln, welche der beiden Funktionen verwendet wird (siehe später)
- Besonderheiten
  - Zuweisung
    * `operator =()` muß Memberfunktion sein
    * "Standardimplementation" : instanzvariablenweises Zuweisen
  - Funktionsaufruf
    * `operator ()()` muß Memberfunktion sein

  * wird als binärer Operator betrachtet
    `<primary_expression> (<expression_list>)`
    `<expression_list>` kann leer sein
    ⇒ `x(arg1, arg2)` wird interpretiert als
    `x.operator()(arg1, arg2)`
- Feldindizierung (Subscription)
  * `operator []()` muß Memberfunktion sein
  * wird als binärer Operator betrachtet
    `<primary_expression>[<expression>]`
    ⇒ `x[y]` wird interpretiert als
    `x.operator [](y)`
- Zugriff zu Member
  * `operator ->()` muß Memberfunktion sein
  * wird als unärer Operator betrachtet
    ⇒ `x->m` wird interpretiert als
    `(x.operator ->()) -> m`
    `operator ->()` muß
    - einen Zeiger auf ein Objekt
    - ein Objekt oder
    - eine Referenz auf ein Objekt liefern
- Inkrement und Dekrement
  müssen Memberfunktionen sein

```
operator ++()       //ohne Parameter: Praefix
operator ++(int)    //ein Parameter: Postfix
                    //Aufruf erfolgt mit Argument 0
```

  Beispiel:

```
class X {
     ...
     operator ++();      // Praefix
     operator ++(int);   // Postfix
};

void f()
{
     X x;

     ++x;       // x.operator++();
     x++;       // x.operator++(0);
}
```

3.2.2 Beispiel

Erweitern der Klasse `String`

– Operationen zum
  * Zuweisen von `String`-Objekten
  * Vergleichen von `String`-Objekten
  * Ausgeben von `String`-Objekten auf `ostreams`
  * Eingeben von `String`-Objekten von `istreams`

```
#include <iostream.h>

class String {
private:
   char *s;
   int size;
public:
   String();
   String(const char*);
   ~String();
   void set(const char*);
   int length() const;
   void print() const;
   String& operator = (const String&);
   friend int operator == (const String&, const String&);
   friend ostream& operator << (ostream&, const String&);
   friend istream& operator >> (istream&, String&);
};

int operator != (const String&, const String&);
```

Bemerkung: Die Operatorfunktionen zur Ein- bzw. Ausgabe sind eigentlich Operatoren für die Klassen `istream` bzw. `ostream`. Sie müssen jedoch als `friend` der Klasse `String` vereinbart werden, da sie auf die privaten Instanzvariablen dieser Klasse zugreifen und nur bereits bekannte Operatorfunktionen der Klassen `istream` bzw. `ostream` benutzen.

```
#include <string.h>
#include "String_4.h"

String& String::operator =(const String& s1)
{
      if (this != &s1)   // verhindert string1 = string1
            set(s1.s);
      return *this;
}

int operator == (const String& s1, const String& s2) {
      return strcmp(s1.s, s2.s) == 0;
}
```

```
int operator != (const String& s1, const String& s2) {
      return !(s1 == s2);
}

ostream& operator << (ostream& os, const String& s1)
{
   os << "[" << s1.length() << "] ";
   if(s1.s)
      os << s1.s;
   else
      os << "(NULL)";
   return os;
}

istream& operator >> (istream& is, String& s1)
{
   char buffer[256];
   is >> buffer;
   s1 = buffer;
   return is;
}
```

Die Rückgabe des {i,o}streams sichert, daß der jeweilige Operator mehrfach für einen Strom angewendet werden kann.

- Anwendung

```
#include <iostream.h>
#include "String_4.h"

int main()

{
    String s1, s2, s3;
    cout << "Zeichenkette eingeben: ";
    cin >> s1;          // operator >> (cin, s1)
    cout << "echo: " << s1 << endl;
                        // operator << (operator << (cout, "echo"),  s1)
    s3 = s2 = s1;       // s3.operator = (s2.operator = (s1))
    if(s3 == s1)        // operator == (s3, s1)
        cout << "s3 == s1, wie erwartet\n";
    else
        cout << "s3 != s1, wieso das ?\n";
    s2 = "Hallo";       // s2.operator = (String::String("Hallo"))
                        // implizite Typkonvertierung
    if("Hallo" == s2)   // operator == (String::String("Hallo"), s2)
                        // implizite Typkonvertierung
                        // wuerde nicht funktionieren,
```

```
                            // wenn operator == () Memberfunktion waere
            cout << "Typkonvertierung funktioniert \n";
        else
            cout << "das sollte nicht passieren \n";
    }
```

- offenbar finden an einigen Stellen implizite Typkonvertierungen statt
- der Compiler wird immer dann solche Typkonvertierungen organisieren, wenn es keine passenden Funktionen gibt
- hätte man eine Funktion

  `String::operator = (const char *)`

  würde die Typkonvertierung nicht benutzt
- im Reference Manual von C++ befinden sich exakte Regeln für die Durchführung von Typkonvertierungen (Argument Matching Rules)
- in Kurzform kann man sich merken:
  - der Compiler wird immer dann, wenn es Funktionen für die übergebenen Argumenttypen gibt, diese auch benutzen
  - gibt es keine solche Funktionen, wird der "beste" Weg zur Typkonvertierung gesucht, dazu werden entweder die Standardtypkonvertierungen für die Grunddatentypen, die Konvertierung über die Anlegung anonymer Objekte mittels entsprechender Konstruktoren oder explizite Typkonvertierungsfunktionen (siehe nächster Punkt) benutzt
  - gibt es keinen eindeutigen Weg für die Typkonvertierung, wird der Compiler eine entsprechende Fehlermitteilung oder Warnung ausgeben

Allgemein lassen sich folgende Empfehlungen für die Entscheidung `friend`- oder Memberfunktionen für Operatoren geben:

1. Unäre Operatoren sollten generell als Memberfunktionen vereinbart werden.
2. Operatoren, die Instanzvariablen manipulieren, sollten als Memberfunktion realisiert werden.
3. Nichtkommutative Operatoren sollten als Member realisiert werden.
4. Der Vorteil der `friend`-Variante besteht darin, daß auf beide Operanden des Operators implizite Typkonvertierungen angewendet wird. Wenn also gerade auf diese Eigenschaft Wert gelegt wird, muß diese Variante benutzt werden.

3.2.3 Explizite Typkonvertierung

- ein Konstruktor einer Klasse mit <u>einem</u> Parameter ist de facto eine Typkonvertierung

  Typ des Parameters → Klasse, zu der der Konstruktor gehört

- es kann auf diesem Weg also nur der Zieltyp der Konvertierung angegeben werden, d.h. wie aus dem Quelltyp ein Objekt des Zieltyps erzeugt werden kann und nicht umgekehrt

- für die Grunddatentypen und davon abgeleitete Typen (insbesondere Zeiger) gibt es die Möglichkeit, eine Typkonvertierung der Form

  Objekt einer Klasse → Grunddatentyp (bzw. Zeiger darauf)

  anzugeben

- diese Form der Typkonvertierung wird erreicht durch eine folgende Memberfunktion:

  `operator` *type*`()`

  wobei *type* der gewünschte Zieltyp ist

Beispiel:

```
class String  {
      ...
public:
      ...
      operator const char*() const;
};

String::operator const char*() const
{
      return s;
}
```

Typkonvertierungsfunktionen haben selbst keinen Typ, da sich ihr Typ „automatisch“ aus ihrem Namen ergibt

- Frage: was wird hier benutzt?

  ```
  String s("Frage");
  cout << s;
  ```

  Möglich sind:

  ```
  ostream& operator << (cout, s) //   oder
  ostream& operator << (cout, s.operator char*())
  ```

- Konvertierungen werden implizit in folgenden Fällen verwendet:
  - Zuweisung
  - Argumentübergabe an Funktionen
  - Rückgabe von Funktionswerten (return)

3.2.4 Zuweisung und Initialisierung

- schon mehrfach auf den Unterschied zwischen Zuweisung und Initialisierung hingewiesen

- Zuweisung:

```
String s1, s2;
s1.set("Beispiel");

s2 = s1;                    // Zuweisung
```

- Initialisierung:

```
String s3 = s1;              // C-Stil
String s4(s1);               // C++-Stil

void f(String str)  { ... }  // Argumentuebergabe
void g() { f(s1); ...  }

String g() { String s; ... ; return s;}
```

- für die Initialisierung wird ein bestimmter Konstruktor (Copy-Konstruktor) verwendet:
  Beispiel:

```
#include <iostream.h>

class String {
private:
   char *s;
   int size;
public:
   String();
   String(const char*);
   String(const String&);   // Copy-Konstruktor
   ~String();
   void set(const char*);
   int length() const;
   void print() const;
   String& operator = (const String&);
   friend int operator == (const String&, const String&);
   friend ostream& operator << (ostream&, const String&);
   friend istream& operator >> (istream&, String&);
};
int operator != (const String&, const String&);
```

- der Konstruktor `String::String(const String&)` erzeugt aus einem bereits bestehenden Objekt der Klasse `String` ein neues Objekt dieser Klasse (daher der Name Copy-Konstruktor):

```
#include <string.h>
#include "String_5.h"

String::String(const String& s1)
{
   s = new char[size = s1.size];
   strcpy(s, s1.s);
}
```

- standardmäßig ist dieser Konstruktor als instanzvariablenweises Kopieren implementiert
- die folgende Definition der Klasse X zeigt eine Klasse; deren Objekte in allen Zusammenhängen korrekt verwendet werden können:

  ```
  class X   {
      ...
  public:
        X();                     //Instanziierung: X x1;
        X(Irgendwas);            //Instanziierung: X x2(irgendwas);
        X(const X&);             //Initialisierung: X x3(x1), x4 = x1;
        X& operator=(const X&)   //Zuweisung:  x2 = x3;
        virtual ~X();            //Beseitigung
        ...
  };
  ```

  die Definition dieser Funktionen ist insbesondere dann erforderlich, wenn in einem Konstruktor Speicherplatz zugewiesen und im Destruktor wieder frei gegeben wird

3.2.5 Aufgabe

- Erweitern der Klasse `String` um Operatorfunktionen für die Operatoren `+` und `+=`
- dabei beachten, daß folgende Dinge anschließend funktionieren:

```
main()
{

   String s1,s2,s3,s4;

   ...
   s1 = s2 + s3 + s4;
   ...
   s1 += s2 += s3;
}
```

- Hinweis: Funktion der Standardbibliothek

  ```
  strcat(char* s1, const char* s2);
  ```

  diese Funktion kopiert die Zeichenkette `s2` an das Ende der Zeichenkette `s1`

Lösung:

```
#ifndef _String_H_
#define _String_H_

#include <iostream.h>

class String {
private:
   char *s;
   int size;
public:
   String();
   String(const char*);
   String(const String&);
   ~String();
   void set(const char*);
   int length() const;
   void print() const;
   String& operator = (const String&);
   friend int operator == (const String&, const String&);
   friend ostream& operator << (ostream&, const String&);
   friend istream& operator >> (istream&, String&);
   String& operator+=(const String&);
   friend String operator+(const String&, const String&);
};

int operator != (const String&, const String&);

#endif /* _String_H_ */

#include <string.h>
#include "String_6.h"

String& String::operator+=(const String& s1)
{
   int new_size = size + s1.length();
   char* new_s = new char[new_size];
   strcpy(new_s, s);
   strcat(new_s, s1.s);
   delete [] s;
```

```
    size = new_size;
    s = new_s;
    return *this;
}

String operator+(const String& s1, const String& s2)
{
    String s3(s1);

    s3 += s2;
    return s3;
}

#include <iostream.h>
#include "String_6.h"

int main()
{
    String s1("High ");
    String s2("Noon! ");
    String s3;

    s3 = s1 + s2;
    cout << s3 << endl;
    cout << s3 + s1 + s2 << endl;
    s3 += s1;
    cout << s3 << endl;
    s3 += s2 += s1;
    cout << s1 << s2 << s3 << endl;
}
```

3.3 Klassenvariablen und Klassenmethoden

- Klassenvariablen (statische Membervariablen) unterscheiden sich von Instanzvariablen in folgenden Punkten:

Instanzvariablen	Klassenvariablen
jedes Objekt besitzt eigene Instanzvariablen und belegt diese mit individuellen Werten	jedes Objekt teilt mit allen Objekten seiner Klasse Klassenvariablen, die folglich für jedes Objekt identische Werte besitzen (Klassenvariablen gibt es für jede Klasse einmal)
sind nur über Objekte zugreifbar	sind zugreifbar, ohne daß ein Objekt dieser Klasse existieren muß

- Beispiel:

```
class A {
```

```
     static int a;    // Deklaration: private Klassenvariable
public:
     static int b;    // Deklaration: oeffentliche Klassenvariable
     void meth();
};

void A::meth()
{
     a = ... ;
}

int A::a = 1;             // Definition
int A::b;

void f()
{
     A object;

     object.b = 2;    // Zugriff ueber Objekt
     A::b = 3;        // Zugriff ueber die Klasse
}
```

- Klassenmethoden (statische Memberfunktionen) unterscheiden sich von anderen Methoden in folgenden Punkten

Methoden	Klassenmethoden
werden infolge einer Botschaft an ein Objekt abgearbeitet (können nur für Objekte gerufen werden)	können gerufen werden, ohne daß ein Objekt dieser Klasse existieren muß
können auf Instanz- und Klassenvariablen zugreifen	können nur auf Klassenvariablen zugreifen

- Beispiel:

```
class A {
     ...
public:
     static void meth();    // oeffentliche Klassenmethode
};

void A::meth()
{
     ...
}

void f()
{
     A object;
```

```
        A::meth();              // Aufruf ueber die Klasse
        object.meth();          // Aufruf ueber Objekt
}
```

3.4 Besonderheiten

3.4.1 Klassen und Strukturen

- eine Klasse, die keine privaten Member enthält, kann als Struktur vereinbart werden
- mit anderen Worten: eine Struktur ist eine Klasse, deren Member generell öffentlich sind
- die beiden folgenden Definitionen sind daher identisch:

```
class X  {
public:
       ...
};

struct X  {
      ...
};
```

- umgekehrt gilt die folgende Identität:

```
struct Y  {
private:
     ...
};

class Y  {
     ...
};
```

- eine Struktur kann also ebenfalls Methoden, insbesondere auch Konstruktoren und einen Destruktor besitzen

3.4.2 Inline Funktionen

- Funktionsaufrufe von Funktionen, die mit dem Schlüsselwort `inline` definiert sind, versucht der Compiler durch Expansion des Funktionsblocks zu realisieren
- `inline`-Funktionen werden deshalb für kurze Funktionen benutzt, um den Overhead eines Funktionsaufrufs zu sparen
- dieser Mechanismus ist ebenfalls für Memberfunktionen anwendbar
  Beispiel:

```
inline int String :: length()
{
     return size ? size - 1 : 0;
}
```

- für jeden Aufruf der Funktion wird der Compiler den Funktionsblock einsetzen:

  ⇒ das setzt voraus, daß der Compiler zum Zeitpunkt der Übersetzung eines Funktionsaufrufs die Funktionsdefinition kennen muß, was sonst nicht üblich ist

  ⇒ `inline`-Funktionen müssen deshalb im Headerfile untergebracht werden

  ⇒ Verletzung des Information-Hiding-Prinzips

- die Definition einer `inline`-Memberfunktion kann innerhalb der Klassendefinition vorgenommen werden:

```
class X  {
private:
    int x;
public:
    void set(int i) {x = i;}
    int get() {return x;}
    X(int i): x(i) {}
};
```

- alle Methoden der Klasse X sind somit implizit `inline` definiert

3.4.3 Beispiel

- Zählen der Häufigkeit von Worten in einem Text, der über die Standardeingabe gelesen wird. Dazu soll ein assoziatives Array verwendet werden, also ein Feld, welches durch Zeichenketten (hier: Worte) indiziert wird und selbst Integer-Werte als Feldelemente enthält.

- zweckmäßigerweise soll die Klasse `String` verwendet werden

- Ziel ist also eine `main`-Funktion, die folgendermaßen aussieht:

```
#include <iostream.h>
#include "AssocArray.h"
#include "String_6.h"

int main ()
{
   String buf;
   AssocArray avek(512);
```

```
      while (cin >> buf)
         avek[buf]++;
      avek.print_all();
   }
```

- von besonderem Interesse ist die Zeile

```
avek[buf]++;
```

  die Klasse `AssocArray` definiert also eine neue Bedeutung des Operators `[]`: als Index wird ein `String` angegeben und das durch diesen Operator referenzierte Element wird inkrementiert

- Ausgangspunkt für die Implementation von assoziativen Arrays ist die Überlegung, eine Menge von Wertepaaren (String und Häufigkeit) mit Hilfe einer Klasse geeignet zu organisieren:

```
#include "String_6.h"

struct Paar {
   String wort;
   int    haeufigkeit;
};
class AssocArray {
private:
   Paar* vek;
   int   anzahl;
   int   frei;
public:
   AssocArray(int = 16);
   ~AssocArray();
   int& operator[](const String&);
   void print_all();
};
```

  dabei besitzen die Member folgende Bedeutung:

`vek`	verweist auf das erste Element eines Feldes von `Paaren`
`anzahl`	ist die aktuelle Anzahl von Elementen im über `vek` addressierten Feld, ggf. wird die Anzahl während der Abarbeitung erhöht
`frei`	ist der Index des nächsten freien Eintrags im über `vek` addressierten Feld
`AssocArray(int = 16)`	ist der Default-Konstruktor, als Parameter wird die initiale Anzahl von Feldelementen übernommen
`int& operator[] (const String&)`	ist der Zugriffsoperator zu einem Objekt der Klasse `AssocArray`, er übernimmt einen `String` und liefert als Resultat den Zugriff zur dazugehörigen Integer-Zahl
`void print_all(void)`	gibt alle Wertepaare auf die Standardausgabe aus

- ```
  #include <iostream.h>
  #include "AssocArray.h"

  AssocArray::AssocArray(int s)
   : anzahl(s < 1 ? 16: s), frei(0) {
     vek = new Paar[anzahl];
  }
  AssocArray::~AssocArray() {
     delete[] vek;
  }

  int& AssocArray::operator[](const String& s1) {
     Paar* pp;

     for(pp = &vek[frei - 1]; vek <= pp; pp--)
        if (s1 == pp->wort)
           return pp->haeufigkeit;
     if (frei == anzahl) {
        Paar* nvek = new Paar[anzahl*2];
        for (int i= 0; i<anzahl; i++)
           nvek[i] = vek[i];
        delete [] vek;
        vek = nvek;
        anzahl *= 2;
     }
     pp = &vek[frei++];
     pp->wort = s1;
     pp->haeufigkeit = 0;
     return pp->haeufigkeit;
  }
  ```
  ```

```
void AssocArray::print_all() {
   for (int i=0; i < frei; i++)
      cout << vek[i].wort << ": " << vek[i].haeufigkeit << "\n";

}
```

4 C++-Templates

Wir arbeiten weiter am Beispiel `AssocArray`. Bis zu diesem Punkt haben wir einen Algorithmus entwickelt, der die Häufigkeit von Worten in einem Text zählt. Die Worte und die zugehörige (*assoziierte*) Häufigkeit werden dann an die Standardausgabe gegeben. Es entstand ein einfacher Container, in welchem Assoziationen vom Typ <`String` → `int`> als Objekte (nicht als Pointer/Referenzen) gespeichert sind. Assoziationen dieser Art sind aber nicht nur für diese, sondern für beliebige Typkombinationen denkbar, also z.B. auch für <`int` → `int`> oder <`float` → `int`>. Ein Container, in welchem Assoziationen gespeichert sind, wird oft auch als *Dictionary* oder als *Map* (Abbildung einer Menge von Schlüsseln *Keys* auf eine Menge von Werten *Values* bezeichnet. Im Unterschied zum bisherigen `AssocArray` sind die Elemente in einem *Map*-Container anhand ihrer Schlüssel geordnet; d.h. für die Schlüsseltypen müssen die <-Operatoren definiert sein.

4.0.4 Nachteile des bisherigen Designs

- Container und auf den Container anzuwendende Funktionalität (`print_all`) innerhalb derselben Klasse und damit voneinander stark abhängig

- wird andere/zusätzliche Funktionalität benötigt, muß die Containerklasse geändert werden

- wird dieselbe Containerfunktionalität auch für andere Assoziationstypen benötigt, dann muß die Klasse kopiert und der Assoziationstyp stets gegen den neuen ausgetauscht werden

4.1 Templates: Grundlagen

Zumindest der letzte der obigen Punkte kann sofort durch Verwendung von Templates automatisiert werden; eine Verbesserung der übrigen Punkte ergibt sich aus der Erarbeitung eines guten Template-Designs im Zusammenhang mit Iteratoren.

- ein C++-Template (auch *generische Klasse oder parametrisierter Typ*) ist eine Schablone für gleichartige Klassen, die sich nur durch eine oder mehrere Typdeklarationen enthaltener Elemente (Daten oder Funktionen) unterscheiden

- mit Hilfe eines C++-Templates kann der Compiler für die als Typparameter übergebenen realen C++-Typen reale C++-Klassen erzeugen

- Templates enthalten daher mindestens einen Typparameter, sozus. als Platzhalter für die realen Typen

- ein Template ist somit eine Art intelligentes Makro, welches die Sichtbarkeits-, Namens- und Typregeln (Typprüfung) von C++ unterstützt

- aber: bei Verwendung von Templates muß davon ausgegangen werden, daß für jeden unterschiedlichen, als Typparameter übergebenen realen Typen auch physisch Code erzeugt wird (Codegröße!)

4.2 Beispiel: Map-Template

- wir definieren zuerst eine allgemeingültige Assoziation als Template mit zwei Typparametern File MapAssocT.h):

```
#ifndef _MAPASSOCT_H_
#define _MAPASSOCT_H_

#include <iostream.h>

template<class K, class V>
struct Assoc  // setzt Defaultkonstruktoren
{             // fuer K und V voraus
  K Key;
  V Value;
};

template<class K, class V>
ostream& operator<<(ostream& ros, Assoc<K,V>& ra)
{
  ros << ra.Key << ": " << ra.Value;
  return ros;
}

#endif
```

- die Syntax `template<class K, class V>` kennzeichnet nachfolgende Typdefinition als Template mit den Typparametern `K` und `V`

- `Assoc<K,V>` ist damit als Klassen-Template definiert und `ostream& operator<<(ostream&, Assoc<K,V>&)` als Funktions-Template; beide Templatedefinitionen verwenden die zwei Typparameter `K` und `V`

- bei der Verwendung dieser Templates erzeugt der Compiler dann den Code für `Assoc`-Klassen und überladene `operator<<`-Funktionen entsprechend den übergebenen realen C++-Typen

- wir definieren jetzt einen Map-Container, in welchem Assoziationen auf der Basis von `IMPLEMENTATION_TYPE<K,V>` gespeichert werden können (File MapT.h):

```
#ifndef _MAPT_H_
#define _MAPT_H_

#include <iostream.h>
```

```
#include "String_7.h"
#include "MapAssocT.h"
#include "MapIteratorT.h"

template<class K, class V>
class Map
{
  typedef Assoc<K,V>* IMPLEMENTATION_TYPE;

private:
  IMPLEMENTATION_TYPE Impl;
  V InitVal;
  int anzahl;
  int frei;

public:
  Map(V _InitVal, int = 16);
  ~Map();
  V& operator[](const K&);

  MapIterator<K,V> begin(){return MapIterator<K,V>(Impl, 0);}
  MapIterator<K,V> end(){return MapIterator<K,V>(Impl, frei);}
};

template<class K, class V>
Map<K,V>::Map(V _InitVal, int s)
: InitVal(_InitVal), anzahl(s < 1 ? 16: s), frei(0)
{
  Impl = new Assoc<K,V>[anzahl]; // setzt Defaultkonstruktor
}                                 // fuer Assoc voraus
template<class K, class V>
Map<K,V>::~Map()
{
  delete[] Impl;
}
template<class K, class V>
V& Map<K,V>::operator[](const K& k1)
{
  IMPLEMENTATION_TYPE pp = Impl, qq;

  if(frei > 0)
  { // setzt Operator < fuer Key-Typ voraus:
    for(; (pp <= &Impl[frei - 1]) && (pp->Key < k1); pp++);

    if(pp < Impl + anzahl)
      if(k1 == pp->Key)
        return pp->Value;
```

```
// pp zeigt nun auf Pos., an der neuer Schluessel eingefuegt wird,
// hier muessen wir eine Luecke schaffen:
    if(frei < anzahl)
      for(qq = &Impl[frei - 1]; qq >= pp; qq--)
        *(qq + 1) = *qq;         // um eins nach hinten schieben
    else
    {
      int i;

      IMPLEMENTATION_TYPE nImpl = new Assoc<K,V>[anzahl * 2];
      for(i = 0; i < (pp - Impl); i++)
        nImpl[i] = Impl[i];    // vorw. bis zur Luecke kop.
      for(i = anzahl - 1; i >= (pp - Impl); i--)
        nImpl[i + 1] = Impl[i];// rueckw. bis zur Luecke kop.
      pp = nImpl + (pp - Impl);
      delete[] Impl;
      Impl = nImpl;
      anzahl *= 2;
    }
  }
  frei++;
  pp->Key = k1;
  pp->Value = InitVal;
  return pp->Value;
}

#endif
```

- `IMPLEMENTATION_TYPE<K,V>` ist eine geschachtelte Typedefinition und gibt an, wie die Map implementiert ist. Die Implementation einer Map kann auf verschiedenen generischen Algorithmen basieren, z.B. verschiedene Listen, Suchbäume, Hashtabellen etc. Der Einfachheit wegen wird unsere Map jedoch als Low-Level-Array implementiert. Das hat den Nachteil, daß bei der Suche nach einem Key das Array sequentiell durchsucht werden muß. Im schlechtesten Fall sind dabei soviele Vergleichsoperationen notwendig, wie Elemente im Container gespeichert sind.

- an Stelle der `print_all`-Funktion sind zwei andere Methoden hinzugekommen, `begin()` und `end()`, die jeweils einen Iterator zur Iteration über den Inhalt des Containerobjektes zurückliefern

- `begin()` initialisiert den gelieferten Iterator mit der Anfangsposition und `end()` seinen mit der Endposition

- weitere Informationen zur Iteration siehe Kapitel *Iteration in Containern*

- die Containerklasse ist so definiert, dass sie ausschließlich zur Verwaltung des Inhaltes (Zufügen von Elementen, Zugriff, evtl. Löschung) geeignet ist; Algorithmen, die mit den enthaltenen Elementen arbeiten, werden basierend auf Iteration über einen Bereich des Containers entwickelt

- ACHTUNG! die durch `begin()` und `end()` gelieferten Iteratoren arbeiten der einfachen Implementation wegen mit den Adressen der Elemente; das bedeutet: nach Aufruf des Subscription-Operators `[]` können wegen der Arrayerweiterung durch Neuallokation die Iteratoren ungültig werden!

4.3 Iteration

- ein Container speichert Objekte, d.h. Daten, mit denen das Programm im weiteren Verlauf arbeitet; denkbar wären Aktionen wie Aus-, Eingabe, Suchen, Verknüpfen, Kopieren, Einfügen, Löschen, Umordnen (Sortieren), ...
- müssen im Zusammenhang mit einer dieser Aktionen (z.B. Suchen) entsprechende Operationen (z.B. Vergleich) auf mehrere Objekte, also auf einen Bereich des Containers (z.B. von Anfang bis Ende) angewendet werden, dann spricht man von einer Operation auf der Basis einer *Iteration* über diesen Bereich
- Iterationen können möglicherweise von verschiedenen Stellen in einem Programmsystem aus gleichzeitig über demselben Containerbereich durchgeführt werden; das Speichern des Zustandes einer Iteration (z.B. aktuelles Element) sollte daher nicht im Container selbst erfolgen
- man definiert Iteratorobjekte, die einerseits den Zustand der Iteration speichern und andererseits Funktionen zur Steuerung der Iteration besitzen;
- für die syntaktische Konstruktion von Iterationen gibt es viele Möglichkeiten, z.B.
  - ```
    Container_Type c;
    Iterator_Type i;
    for(i = c.First(); i; c.next(i)) Use(c(i));
    ```
    * z.B. `c.First()` liefert einen Iterator, der mit dem ersten Element initialisiert ist
    * `c.next(i)` inkrementiert den Iterator
    * `Use(c(i))` ruft eine Funktion für das vom Iterator als aktuelles bestimmte Element des Containers `c` auf
  - ```
    Element_Type e;
    Container_Type c;
    Iterator_Type i(c);
    while(e = i.next()) Use(e);
    ```
    * z.B. `i.next()` gibt in `e` jeweils das aktuelle Element oder NULL (Ende!) zurück und inkrementiert die interne Position der Iteration

- oft auch zeigerartige Notation:
  ```
  Container_Type c;
  Iterator_Type i;
  for(i = c.First(); i; i++) Use(*i);
  ```
  - `*i` *dereferenziert* den Iterator und liefert eine Referenz auf das aktuelle Element
- in der STL wird der Iterationsbereich oft durch zwei Iteratoren abgesteckt, die untereinander verglichen werden können:
  ```
  Container_Type c;
  Iterator_Type First = c.begin(), Last = c.end();
  while(First != Last) Use(*first++);
  ```

Wir greifen die letzte Notation auf und definieren eine Iterator-Klasse für unsere Map (File MapIteratorT.h):

```
#ifndef _MAPITERATORT_H_
#define _MAPITERATORT_H_

#include <iostream.h>
#include <bool.h>
#include "String_7.h"
#include "MapAssocT.h"

template<class K, class V>
class MapIterator
{
  typedef Assoc<K,V>* IMPLEMENTATION_TYPE;

private:
  IMPLEMENTATION_TYPE Impl;
  int Position;

public:
  MapIterator()
  : Impl(NULL), Position(0){}

  MapIterator(IMPLEMENTATION_TYPE _Impl, int _Index = 0)
  : Impl(_Impl), Position(_Index){}

  Assoc<K,V>& operator*()
  {
    return *(Impl + Position);
  }
  Assoc<K,V>& operator++()     // prefix
  {
    return *(Impl + ++Position);
  }
```

```
  Assoc<K,V>& operator++(int)  // postfix
  {
    return *(Impl + Position++);
  }
friend bool operator==(MapIterator&, MapIterator&);
friend bool operator!=(MapIterator&, MapIterator&);
};

template<class K, class V>
bool operator==(MapIterator<K,V>& I1, MapIterator<K,V>& I2)
{
  return ((I1.Impl == I2.Impl) && (I1.Position == I2.Position));
}
template<class K, class V>
bool operator!=(MapIterator<K,V>& I1, MapIterator<K,V>& I2)
{
  return !operator==(I1, I2);
}

#endif
```

4.4 Generische Algorithmen

- eine (Template-)Funktion, die einen Algorithmus enthält, der (z.B. durch Verwendung von Iteratoren) auf Container verschiedenen Typs (Vektoren, Listen, Sets, Maps, Stacks, ...) und verschiedener Typparameter angewendet werden kann, wird oft auch als *generischer* Algorithmus bezeichnet

- ein generischer Algorithmus kann definiert werden, ohne daß der reale C++-Typ der Objekte im Container bereits bekannt ist

- ein generischer Algorithmus gibt aber oft Auflagen für die Definition der zu nutzenden C++-Typen, z.B. könnte die Definition spezieller Operationen gefordert werden (Beispiel: `operator<` für einen *sort*-Algorithmus)

- In unserem Beispiel implementieren wir neben der `print_all`-Funktion auch noch eine `print_sorted`-Funktion, die die <`Key` → `Value`>-Paare zusätzlich nach der Größe von `Value` geordnet ausgeben kann (File MapFuncT.h):

```
#ifndef _MAPFUNCT_H_
#define _MAPFUNCT_H_

#include <iostream.h>
#include "String_7.h"
#include "MapAssocT.h"
#include "MapIteratorT.h"
```

```
template<class K, class V>
ostream& print_all(ostream& ros,
                   MapIterator<K,V> first,
                   MapIterator<K,V> last)
{
  while(first != last)
  {
    ros << *first << endl;
    first++;
  }
  return ros;
}

template<class K, class V>
ostream& print_sorted(ostream& ros,
                      MapIterator<K,V> first,
                      MapIterator<K,V> last)
{
  int ActVal = 0;
  bool NextVal = true;
  MapIterator<K,V> Next;

  while(NextVal)
  {
    Next = first;
    NextVal = false;

    while(Next != last)
    {
      if((*Next).Value == ActVal)
        ros << *Next << endl;
      else
        if((*Next).Value > ActVal)
          NextVal = true;

      Next++;
    }

    ActVal++;
  }
  return ros;
}

#endif
```

- Damit kann man eine einfache `main()`-Funktion schreiben, die die Funktionalität "Zählen der Häufigkeit von Worten in einem Text, der über die Standardeingabe gelesen wird" erfüllt, zusätzlich aber die Elemente erstens geordnet nach den Worten und zweitens geordnet nach der Häufigkeit ausgibt (File mainMap_String_int.cc):

```
#include <iostream.h>
#include "MapT.h"
#include "MapIteratorT.h"
#include "MapFuncT.h"

int main ()
{
  String buf;
  Map<String,int> avek(0, 2);

  while (cin >> buf)
    avek[buf]++;

  cout << endl << "sorted by keys:" << endl;
  print_all(cout, avek.begin(), avek.end());

  cout << endl << "sorted by values:" << endl;
  print_sorted(cout, avek.begin(), avek.end());
}
```

- eine mögliche Programmausgabe:

```
../BEISPIELE > mainMap_String_int
a1
b3
d4 d4
b3
d4
b3
z2 z2
t6 t6 t6
s1
t6 t6
d4 t6

sorted by keys:
[2] a1: 1
[2] b3: 3
[2] d4: 4
[2] s1: 1
[2] t6: 6
[2] z2: 2
```

```
sorted by values:
[2] a1: 1
[2] s1: 1
[2] z2: 2
[2] b3: 3
[2] d4: 4
[2] t6: 6
```

- Bitte testen Sie auch das Programm mainMap_int_int, welches dieselbe Funktionalität für Integer-Zahlen anstatt für Strings implementiert unter Verwendung desselben Container-Templates!

5 Vererbung - Aufbau von Klassenhierarchien

- bisher immer auf dem Niveau abstrakter Datentypen
- zwar neue Typen gebaut, jedoch noch keine ooP
- nächster Schritt in Richtung ooP: Vererbung

5.1 Ableiten von Klassen

- Prinzip am Beispiel zeigen
- Ausgangspunkt:
  - geometrischer Sachverhalt
  - Arbeit mit kartesischen Koordinaten
  - Punkt in der Ebene durch zwei Koordinaten beschreiben: x und y

  File: Punkt.h

```
#ifndef _Punkt_H_
#define _Punkt_H_

struct Punkt {
   float x, y;

   Punkt (float xx = .0,  float yy = .0) : x(xx), y(yy) {}
};

#endif /* _Punkt_H_ */
```

- ein Rechteck wird durch zwei Punkte eindeutig beschrieben (unter der Voraussetzung, daß seine Seiten parallel zu den Achsen des Koordinatensystem verlaufen):

  File: Rechteck_2.h

```
#ifndef _Rechteck_H_
#define _Rechteck_H_

#include "Punkt.h"

class Rechteck {
private:
   Punkt pa;
   Punkt pb;
public:
   Rechteck() {}          // wird beim Ableiten gebraucht
   Rechteck(const Punkt& p1, const Punkt& p2)
   : pa(p1), pb(p2){}
```

```
   void setze_punkte(const Punkt&, const Punkt&);
   float flaeche() const;
};

#endif /* _Rechteck_H_ */
```

File: Rechteck_2.C

```
#include <math.h>
#include "Rechteck_2.h"

void Rechteck::setze_punkte(const Punkt& p1, const Punkt& p2) {
  pa = p1;
  pb = p2;
}

float Rechteck::flaeche() const {
   return fabs((pa.x - pb.x) * (pa.y - pb.y));
}
```

- diese Klasse kann nun verwendet werden:

```
#include <iostream.h>
#include "Rechteck_2.h"

int main()
{
     Rechteck  r(Punkt(1., 1.), Punkt(10.,  10.));
     Rechteck * pr = new Rechteck;

     cout << "Flaeche von r: " <<  r.flaeche() << "\n";
     pr->setze_punkte(Punkt(-1., -1.), Punkt(-3.4, 7.89));
     cout << "Flaeche von *pr: " << pr -> flaeche() << "\n";
}
```

- jetzt neue Klassen schaffen:
  Rechteck mit abgerundeten Ecken
  - eigentlich ist das auch ein Rechteck, jedoch braucht man eine zusätzliche Größe zur Beschreibung eines solchen Objekts: Radius der abgerundeten Ecken
  - die Fläche berechnet sich so ähnlich, wie für das Rechteck, jedoch müssen Teilflächen der wegfallenden Ecken abgezogen werden
  - ...

File: RechteckRund.h

```
#ifndef _RechteckRund_H_
```

```
#define _RechteckRund_H_

#include "Rechteck_2.h"

class RechteckRund : public Rechteck  {
private:
   float radius;
public:
   RechteckRund() : radius(.0) {}
   RechteckRund(const Punkt& p1, const Punkt& p2, float r)
    : Rechteck(p1,p2), radius(r) {}
   void setze_radius(float);
   float flaeche() const;
};

const float PI = 3.14159265389;

#endif /* _RechteckRund_H_ */
```

- die Klasse `RechteckRund` wird von `Rechteck` abgeleitet (ein `RechteckRund` *ist ein* `Rechteck`)
- dabei erbt die Klasse `RechteckRund` alle Instanzvariablen <u>und</u> Methoden der Klasse `Rechteck`
- die Angabe `: public Rechteck` bedeutet, daß die Klasse `RechteckRund` alle öffentlichen Methoden der Klasse `Rechteck` ebenfalls an ihrer öffentlichen Schnittstelle bereitstellt
  ⇒ für ein Objekt der Klasse `RechteckRund` kann ebenfalls die Methode `setze_punkte` gerufen werden !
- ein Objekt von `RechteckRund` hat ebenfalls die Instanzvariablen `pa` und `pb`, diese sind jedoch völlig verborgen, d. h. nicht sichtbar, auch nicht für die Methoden von `RechteckRund`
  File: RechteckRund.C

```
#include "RechteckRund.h"

void RechteckRund::setze_radius(float r)
{
   radius = r;
}

float RechteckRund::flaeche() const
{
   return Rechteck::flaeche() - radius * radius * (4. - PI);
}
```

- Verwendung im File mainRechteckRund.C:

```
#include <iostream.h>
#include "RechteckRund.h"

int main()
{
   RechteckRund rr(Punkt(1., 1.), Punkt(10., 10.), .5);

   cout << "Flaeche: " << rr.flaeche() << "\n";
}
```

- allgemein:

  `class` $name_1$ `: private|protected|public` $name_2$ `{`

  `...`

  `};`

  - die Klasse $name_1$ wird von der Klasse $name_2$ abgeleitet
  - die Klasse $name_2$ wird Basisklasse, Vaterklasse oder Oberklasse genannt
  - die Klasse $name_1$ heißt abgeleitete Klasse, Kindklasse oder Unterklasse
  - `:public` - die `public` Member von $name_2$ sind auch `public` Member von $name_1$, die `protected` Member von $name_2$ sind auch `protected` Member von $name_1$
  - `:protected` - die `public` und `protected` Member von $name_2$ sind `protected` Member von $name_1$
  - `:private` - die `public` und `protected` Member von $name_2$ sind `private` Member von $name_1$
  - `:private` ist Standard, wenn von einer `class` abgeleitet wird; `:public` ist Standard, wenn von einer `struct` abgeleitet wird
  - die `private` Member von $name_2$ sind völlig verborgen, sie sind auch für die Methoden von $name_1$ nicht zugreifbar
  - ein Objekt von $name_1$ verfügt über <u>alle</u> Instanzvariablen von $name_2$, die von $name_2$ geerbten Methoden arbeiten mit diesen Instanzvariablen
  - Ableiten ist über mehrere (beliebig viele) Stufen möglich
  - die Unterklasse kann
    * neue Instanzvariable hinzufügen
    * neue Methoden hinzufügen
    * vorhandene Methoden redefinieren (überschreiben)
    * beim `private`-Ableiten können über eine sog. Zugriffsdeklaration Teile der öffentliche Schnittstelle wieder zugänglich (öffentlich) gemacht werden
  - Ableiten wird benutzt, um Spezialisierung entlang einer konzeptuellen Hierarchie auszudrücken

Aufgabe: • Vervollständigen des Geometrie-Beispiels, neue Klassen: Kreis, Dreieck etc., weitere Funktionen z.B. E/A von geom. Objekten

Beispiel:

```
#ifndef _Dreieck_H_
#define _Dreieck_H_

#include "Punkt.h"

class Dreieck {
private:
   Punkt pa,pb,pc;
public:
   Dreieck(Punkt,Punkt,Punkt);
   float flaeche() const;
};

#endif /* _Dreieck_H_ */

#include <math.h>
#include "Dreieck.h"

Dreieck::Dreieck(Punkt p1, Punkt p2, Punkt p3)
 : pa(p1), pb(p2), pc(p3)
{}

float Dreieck::flaeche() const
{
   return fabs(pa.x*(pb.y-pc.y)+pb.x*(pc.y-pa.y)+pc.x*(pa.y-pb.y))/2.;
}
```

5.2 Objekte von abgeleiteten Klassen

- alle Möglichkeiten zur Instanziierung von Objekten
- Besonderheit: Zeiger

```
#include <iostream.h>
#include "RechteckRund.h"

Punkt P1(1., 1.);
Punkt P2(2., 7.);
Punkt P3(8., 17.);
Punkt P4(2., 4.);
int main()
{
   Rechteck r(P1, P2);
```

```
    RechteckRund rr(P3, P4, 1);

    cout << "Flaeche r: " << r.flaeche() << "\n";
    cout << "Flaeche rr: " << rr.flaeche() << "\n";
    Rechteck * pr = &r;
    cout << "Flaeche *pr: " << pr ->flaeche() << "\n";
    RechteckRund * prr = &rr;
    cout << "Flaeche *prr: " << prr -> flaeche() << "\n";
    pr = &rr;                   // Typkonflikt ? (1)
    cout << "Flaeche *pr: " << pr -> flaeche() << "\n";
    // prr = pr;                // Typkonflikt ? (2)
    prr = (RechteckRund*)pr;    // Loesung
    cout << "Flaeche *prr: " << prr -> flaeche() << "\n";
}
```

- die Arbeit mit Zeigern funktioniert offenbar zunächst wie erwartet
- insbesondere ist die Anweisung (1) kein Typkonflikt
- einem Zeiger auf eine Oberklasse kann die Adresse eines Objekts einer Unterklasse zugewiesen werden
- jedoch wird in der anschließenden Benutzung (`pr->flaeche()`) offenbar die Methode `Rechteck::flaeche() const` benutzt
- umgekehrt ist die Anweisung (2) ein Typkonflikt, durch cast - Operation aber lösbar

  ```
  prr = (RechteckRund *) pr;
  ```

- anschließend funktioniert `prr->flaeche()` aber korrekt, es sind also keine Informationen verloren gegangen
- Ursache:
  Aufgrund der Botschaft

  `r.flaeche()` bzw.
  `pr->flaeche()`

  ermittelt bereits der Compiler, welche Methode ausgeführt werden soll: `Rechteck::flaeche() const`

  - dieser Vorgang wird statisches Binden genannt, an eine Botschaft wird bereits zur Übersetzungzeit eine Methode gebunden
  - ausschlaggebend für das statische Binden ist dabei der Typ des Objektes

    ```
    Rechteck* pr;
    pr = &rr;
    pr->flaeche();
    ```

    die auszuführende Methode wird aufgrund des Typs der Variable `pr` ermittelt (`pr` ist vom Typ `Rechteck*`) der Typ des Objektes, welches über `pr` addressiert wird, findet dabei keine Berücksichtigung

5.3 Konstruktoren und Destruktoren

- in welcher Reihenfolge werden Konstruktoren/Destruktoren für Objekte abgeleiteter Klassen gerufen

- Beispiel:

```
class Base   {
      int i;
public:
      Base(int ii)  {i = ii;}
      ...
};
class Derived:  public Base  {
      Base b;
      int j;
public:
      Derived(int x) : Base(x), b(x+1)  {j = x+2;}
};
```

- folgende Abläufe:
  - zuerst wird der Konstruktor der Oberklasse gerufen, d. h. bei Ableitungen über mehrere Stufen wird zuerst der Konstruktor der höchsten Klasse (Wurzelklasse) gerufen
  - anschließend werden die Konstruktoren für Memberobjekte gerufen (ausschlaggebend ist dabei die Reihenfolge der Deklaration der Memberobjekte in der Klassendefinition)
  - danach wird der Funktionsblock des Konstruktors abgearbeitet
  - diese Reihenfolge wird immer eingehalten, unabhängig von der Reihenfolge in der Definition des Konstruktors:

```
Derived(int x) : b(x+1), Base(x) {j = x+2;}
```

Destruktoren werden in genau umgekehrter Reihenfolge abgearbeitet

5.4 Mehrfachvererbung

- Aussagen zur Mehrfachvererbung (multiple inheritance) hier nur prinzipell an „konstruierten" Beispielen

- Ausgangspunkt: bisher bei der Ableitung immer von <u>einer</u> Oberklasse ausgegangen, bei der Mehrfachvererbung besitzt eine abgeleitete Klasse mehrere Oberklassen

- Beispiel

```
class A {                             class B {
    int x;                                char y;
public:                               public:
```

```
    A() : x(1) {}                        B() : y('*') {}
    void printA()                        void printB()
      { cout << x << "\n"; }               { cout << y << "\n"; }
};                                   };
```

```
class C : public A, public B {
    float z;
public:
    C() : z(2.5) {}
    void printC()
      { cout << z << "\n"; }
};
```

- Was passiert bei der Instanziierung eines Objektes der Klasse `C`?

  ```
  C c;
  ```

  das Objekt `c` besteht aus den (geerbten) Instanzvariablen `int A::x` und `char B::y` sowie der (eigenen) Instanzvariablen `float z`

- die Reihenfolge der Konstruktor-Aufrufe bei der Initialisierung des Objektes `c` richtet sich nach der Reihenfolge der Angabe der Oberklassen, es werden also die Konstruktoren `A::A()`, `B::B()` und `C::C()` in dieser Reihenfolge abgearbeitet

- im Zusammenhang mit der Mehrfachvererbung stehen eine Reihe Probleme, von denen zwei hier angedeutet werden sollen
  erstens soll das Problem betrachtet werden, welches entsteht, wenn in den Oberklassen gleichnamige Methoden vereinbart sind:

```
class A {                            class B {
    int x;                               char y;
public:                              public:
    A() : x(1) {}                       B() : y('*') {}
    void print()                        void print()
      { cout << x << "\n"; }               { cout << y << "\n"; }
};                                   };
```

```
class C : public A, public B {
    float z;
public:
    C() : z(2.5) {}
    void printC()
      { cout << z << "\n"; }
};
```

sowohl in der Klasse `A` als auch in der Klasse `B` gibt es in diesem Beispiel eine Methode `void print()`, das Problem wird deutlich, wenn nun auf ein Objekt der Klasse `C` diese Methode angewendet werden soll:

```
C c;

c.print();   // error: ambigous request for function from
             // distinct base classes of type ‘C’
             // first candidate ‘void B::print()’
             // second candidate ‘void A::print()’
```

der Compiler kann nicht entscheiden, welche der beiden geerbten Methoden anzuwenden ist

Lösung:

```
class C : public A, public B {
    float z;
 public:
     C() : z(2.5) {}
     void print()
       { A::print(); B::print(); }
     void printC()
       { cout << z << "\n"; }
 };
```

- das zweite Problem behandelt folgende Situation:

```
class A {
   int x;
public:
   A(int i = 1) : x(i) {}
   void print()
     { cout << x << "\n"; }
};
```

```
class B : public A {
    int r;
public:
    B(int i) :  A(2), r(i) {}
    void printB()
      { cout << r << "\n";
        print(); }
};
```

```
class C : public A {
    int s;
public:
  C(int i) : A(4), s(i) {}
    void printC()
      { cout << s << "\n";
        print(); }
};
```

```
class D : public B, public C {
    int t;
public:
    D(int i) : B(5), C(6), t(i) {}
```

```
        void printD()
          { cout << t << "\n"; }
};
```

- Was passiert bei der Instanziierung eines Objektes der Klasse `D`?

  `D d(10);`

  das Objekt `d` besteht aus den (geerbten) Instanzvariablen `int B::A::x`, `int B::r`, `int C::A::x` und `int C::s` sowie der (eigenen) Instanzvariablen `int t`, aufgrund der Reihenfolge der Konstruktoraufrufe werden diese Instanzvariablen folgendermaßen initialisiert

  ```
  B::A::x = 2
  B::r = 5
  C::A::x = 4
  C::s = 6
  t = 10
  ```

- das Problem wird deutlich, wenn für das Objekt `d` die Methode `void print()` gerufen werden soll, da in diesem Fall der Compiler nicht entscheiden kann, welche der über verschiedene Wege geerbten Methode gemeint ist

  ```
  d.print(); // error: member ‘print’ belongs to multiple inheritance
             // distinct base classes ‘A’
  ```

- Lösung:

  1. Überschreiben der Methode `void print()` in der Klasse `D`
  2. Verwendung virtuelle Basisklassen (virtual base classes)

- virtuelle Basisklassen:

```
class A {
   int x;
public:
   A(int i = 1) : x(i) {}
   void print()
     { cout << x << "\n"; }
};
```

```
class B : public virtual A {
    int r;
public:
    B(int i) :  A(2), r(i) {}
    void printB()
      { cout << r << "\n";
        print(); }
};
```

```
class C : public virtual A {
    char s;
public:
  C(int i) : A(4), s(i) {}
    void printC()
      { cout << s << "\n";
        print(); }
};
```

```
class D : public B, public C {
    int t;
public:
    D(int i) : B(5), C(6), t(i) {}
    void printD()
      { cout << t << "\n"; }
};
```

Die Verwendung virtueller Basisklassen hat zur Konsequenz, daß ein Objekt der Klasse `D` den Teil der Klasse `A` nicht mehrfach erbt, sondern nur einmal enthält. Damit verbunden sind folgende Konsequenzen:

1. die Konstruktoraufrufe `A(2)` bzw. `A(4)` aus `B` bzw. `C` werden ignoriert
2. die Initialisierung der virtuellen Basisklasse (hier Klasse `A`) wird direkt von `D::D` aus vorgenommen, entweder explizit, z.B.:

   ```
   D::D(int i) : A(9), B(5), C(6), t(i) {}
   ```

   oder implizit: `A` muß dann einen Default-Konstruktor besitzen (Im Gegensatz dazu kann bei nicht virtuellen Klassen immer nur der Konstruktor der unmittelbaren Oberklassen gerufen werden.)

5.5 Polymorphes Verhalten durch virtuelle Funktionen

5.5.1 Definition und Verwendung

- zurück zum Beispiel `RechteckRund`:

  ```
  ...
  Rechteck r(P1, P2);
  RechteckRund rr(P3, P4, 1.)
  ...
  Rechteck * pr = & r;
  cout << "Flaeche *pr:" << pr -> flaeche() << "\n";
  pr = &rr;
  cout << "Flaeche * pr:" << pr -> flaeche() << "\n";
  ```

  ⇒ Problem statisches Binden wird gelöst durch virtuelle Funktionen

- dazu Klassendefinition von `Rechteck` ändern:

  ```
  #ifndef _Rechteck_H_
  #define _Rechteck_H_

  #include "Punkt.h"

  class Rechteck {
  private:
     Punkt pa;
  ```

```
   Punkt pb;
public:
   Rechteck() {}          // wird beim Ableiten gebraucht
   Rechteck(const Punkt& p1, const Punkt& p2)
   : pa(p1), pb(p2){}
   void setze_punkte(const Punkt&, const Punkt&);
   virtual float flaeche() const;
};

#endif /* _Rechteck_H_ */
```

- das Schlüsselwort virtual erzwingt dynamisches Binden, d. h. aufgrund der Botschaft

  ```
  pr->flaeche();
  ```

  wird erst zur Laufzeit des Programms ermittelt, welche Methode ausgeführt werden soll, ausschlaggebend ist der Typ des Objektes, worauf der Zeiger tatsächlich verweist

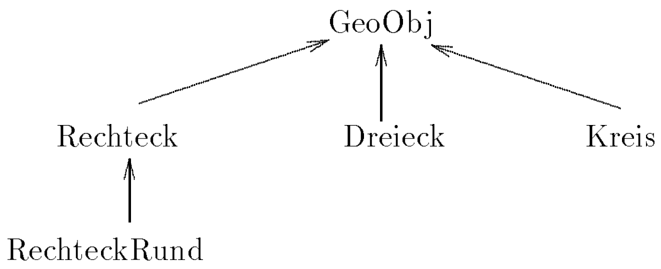
- dynamisches Binden bedeutet, daß zur Laufzeit entschieden wird, welche Methode an die Botschaft gebunden wird

5.5.2 Abstrakte Konzepte

- bisher folgende Klassen

- jede Klasse besitzt eine Methode `flaeche()`

- Idee: gemeinsame Oberklasse schaffen und Methode `flaeche()` herausziehen



- File GeoObj.h

```
#ifndef _GeoObj_H_
#define _GeoObj_H_

class GeoObj {
public:
```

```
   virtual float flaeche() const = 0;
};

#endif /* _GeoObj_H_ */
```

- die Methode `float flaeche() const` ist hier nicht implementierbar, was die Notation `=0` anzeigt
- solche Methoden heißen *rein virtuelle Funktionen (pure virtual functions)*
- diese Klasse wird damit zur abstrakten Klasse, es macht keinen Sinn davon Objekte zu erzeugen
- alle Klassen entsprechend ihrer Stellung in der Vererbungshierarchie nun von `GeoObj` ableiten

```
#ifndef _Rechteck_H_
#define _Rechteck_H_

#include "Punkt.h"
#include "GeoObj.h"

class Rechteck : public GeoObj {
private:
   Punkt pa;
   Punkt pb;
public:
   Rechteck() {}          // wird beim Ableiten gebraucht
   Rechteck(const Punkt& p1, const Punkt& p2)
   : pa(p1), pb(p2){}
   void setze_punkte(const Punkt&, const Punkt&);
   float flaeche() const;
};

#endif /* _Rechteck_H_ */
```

- nun funktioniert

```
void f(GeoObj * p)
{
     cout << p -> flaeche() << "\n";
}

void g()
{
   Rechteck r(Punkt(1.,1.), Punkt(5.,7.));

   f(&r);
   . . .
 }
```

- auch für Referenzen

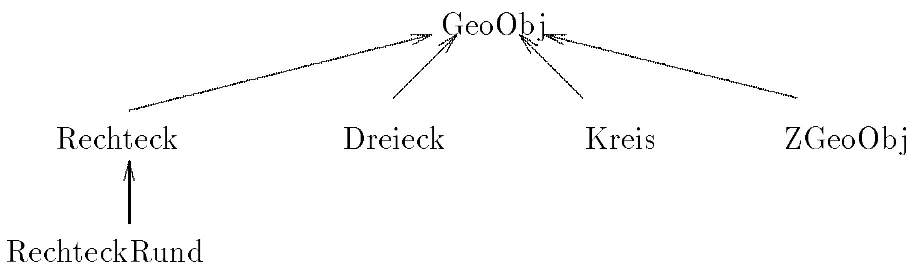
```
void h(GeoObj &o)
{
    cout << o.flaeche() << "\n";
}

void l()
{
   Rechteck r(Punkt(1.,1.), Punkt(5.,7.));

   f(r);
   . . .
 }
```

5.5.3 Beispiel: einfacher Container für polymorphe Klassen

- Idee: Klassenhierarchie erweitern
  Zusammengesetztes geometrisches Objekt
  - ein zusammengesetztes geometrisches Objekt *ist ein* geometrisches Objekt
  - ein zusammengesetztes geometrisches Objekt `besteht aus` anderen geometrischen Objekten
- Klassenhierarchie



- File ZGeoObj.h

```
#include "GeoObj.h"

const int MAX = 10;

class ZGeoObj : public GeoObj {
private:
   GeoObj * go[MAX];
   int anzahl;
public:
   ZGeoObj() : anzahl(0) {}
   void set(GeoObj& g);
   float flaeche() const;
```

```
};
```

- File ZGeoObj.C

```
#include "ZGeoObj.h"

void ZGeoObj :: set(GeoObj& g)
{
   if(anzahl < MAX)
      go[anzahl++] = &g;
}

float ZGeoObj::flaeche() const
{
   float fl = .0;
   for(int i = 0; i < anzahl; i++)
      fl +=  go[i] -> flaeche();
   return fl;
}
```

- Verwendung im File mainZGeoObj.C

```
#include <typeinfo>
#include <iostream.h>
#include "Rechteck_4.h"
#include "Dreieck_2.h"
#include "ZGeoObj.h"

int main()
{
   Rechteck r1(Punkt(0., 0.), Punkt(6., 5.));
   Rechteck r2(Punkt(6., 0.), Punkt(15., 5.));
   Dreieck   d(Punkt(0., 5.), Punkt(6., 5.), Punkt(3., 9.));
   ZGeoObj z1, z2;

   cout << "Flaeche r1: " << r1.flaeche() << "\n";
   cout << "Flaeche r2: " << r2.flaeche() << "\n";
   cout << "Flaeche  d: " << d.flaeche() << "\n";
   z1.set(r1);
   z1.set(d);
   cout << "Flaeche z1: " << z1.flaeche() << "\n";
   z2.set(z1);
   z2.set(r2);
   cout << "Flaeche z2: " << z2.flaeche() << "\n";
}

/* erwartete Ausgabe:
   Flaeche r1: 30
```

```
   Flaeche r2: 45
   Flaeche  d: 12
   Flaeche z1: 42
   Flaeche z2: 87
*/
```

6 Run-Time-Type-Informations (RTTI)

Ein Container wie der im vorherigen Kapitel vorgestellte `ZGeoObj`-Typ speichert in der Regel Pointer oder Referenzen auf eine innerhalb der C++-Klassenhierarchie vorhandene Basisklasse, die, wie gesehen, abstrakt sein kann. Bei der Verwendung des Containers können dann Pointer oder Referenzen auf Objekte ganz verschiedenen Typs an diesen übergeben werden. Die Typen dieser Objekte haben eines gemeinsam: Sie sind direkt oder indirekt von dem Basistyp abgeleitet, für den der Container ausgelegt ist.

In unserem Beispiel ist `GeoObj` die abstrakte Basisklasse und `ZGeoObj` der Container, der Pointer auf `GeoObj`'e aufnehmen kann. Dabei *ist* `ZGeoObj` ebenfalls ein `GeoObj`, genau wie `Rechteck`, `Dreieck`, etc.. Entsprechend der *IST*-Relation können an `ZGeoObj` also auch Pointer auf `Rechteck`'e, `Dreieck`'e und sogar auf andere `ZGeoObj`'e übergeben werden.

Die (hier implizite) Konvertierung eines Pointers auf ein `Rechteck` in einen Pointer auf ein `GeoObj` nennt man auch *statisch*. Eine statische Konvertierung kann bereits vom Compiler vorgenommen werden, da hierfür nur das Wissen über die Ableitungshierarchie vorhanden sein muß. Siehe auch `static_cast`.

Bei Verarbeitung der Objekte in einem Container konnten bis hierhin nur durch die Verwendung virtueller Funktionen typselektive Operationen durchgeführt werden. Manchmal ist es aber notwendig, den genauen Typ eines vom Container gelieferten Objektes zu kennen, z.B. wenn ein solches Objekt kopiert werden soll und Speicher der korrekten Größe zur Verfügung gestellt werden muß. In Umkehrung der oben erwähnten impliziten statischen Konvertierung von einem Pointer auf die abgeleitete Klasse in einen Pointer auf die Basisklasse ist oft auch die Konvertierung eines Pointers auf die Basisklasse (wie im Container gespeichert) in einen Pointer auf die abgeleitete Klasse (die real referenzierten, kompletten Objekte) notwendig.

Eine Konvertierung in diese Richtung kann auf zwei verschiedenen Wegen durchgeführt werden:

- traditionell: expliziter Cast
  z.B. `Dreieck* pdr = (Dreieck*)(z1[0]);`
  - ist gefährlich und ein grober Fehler, wenn durch `z1[0]` nicht wirklich ein `Dreieck` referenziert wird

- mit RTTI: dynamischer expliziter Cast
  z.B. `Dreieck* pdr = dynamic_cast<Dreieck*>(z1[0]);`
  - ist rel. sicher, da im Fehlerfalle eines Pointercasts ein NULL-Pointer zurückgeliefert und im Fehlerfalle eines Referenzcasts eine Ausnahme ausgeworfen wird

Das Abfragen von Typinformationen sowie die dynamische Konvertierung von Pointern und Referenzen zur Laufzeit im Anwenderprogramm wird möglich durch das Speichern von Typinformationen vom Laufzeitsystem *Run-Time-Type-Informations (RTTI)*.

6.1 Verfügbarmachen von Typinformationen zur Laufzeit

Um Typinformationen zur Laufzeit verwenden zu können, sind von Compiler zu Compiler verschiedene Maßnahmen notwendig. Folgende Aufstellung soll als Anhaltspunkt dienen:

- Compileroption zum Erzeugen der RTTI-Informationen einschalten
  Borland: `-RT`; GNU: `-frtti`
- Header mit Definition der `typeinfo`-Klasse vor allen anderen Headern includieren
- sollen dynamische Typinformationen benutzt werden, muß die jeweilige Klasse polymorph sein, d.h. mindestens eine virtuelle Funktion besitzen

6.2 Typinformationen abfragen und verwenden

Zur Abfrage der Typinformation dient der Operator `typeid`:

- `typeid(ausdruck)`
  `typeid(typ_name)`
- `typeid`-Aufrufe liefern eine Referenz auf ein Objekt des Typs `const typeinfo` zurück. Dieser Typ ist oft so definiert:

```
class type_info {
private:
  // assigning type_info is not supported.  made private.
  type_info& operator=(const type_info&);
  type_info(const type_info&);
public:
  virtual ~type_info() {}
  type_info() {}

  bool operator==(const type_info& arg) const;
  bool operator!=(const type_info& arg) const;
  bool before(const type_info& arg);
  const char* name() const;
};
```

  Wenn der Operand eine Referenz auf einen polymorhen Typ ist, dann repräsentiert das zurückgelieferte `typeinfo`-Objekt unabhängig vom (statischen) Typ des Operanden den realen Typen des referenzierten, also des kompletten Objektes (dynamische Abfrage). Wenn der Typ des referenzierten Objektes nicht polymorph ist, dann repräsentiert das zurückgelieferte `typeinfo`-Objekt den statischen Typen des Operanden.

- Der `typeid`-Operator sollte mit eingebauten Basistypen genauso funktionieren wie mit selbstdefinierten C++-Typen (Vorsicht hier beim GNU-Compiler!).

- Wenn der Operand ein dereferenzierter Nullzeiger ist, dann wird in der Regel eine Exception (bad_typeid) ausgelöst.

- Das zurückgelieferte typeinfo-Objekt besitzt Methoden bzw. Operatorfunktionen zum Vergleich mit anderen typeinfo-Objekten sowie zur Ausgabe der Typbezeichnung als char[].

6.3 Möglichkeiten der Typumwandlung

In C++ gibt es alternative Möglichkeiten zur Typumwandlung, die die von C her bekannten expliziten Casting-Operationen erweitern:

- const_cast<T>(arg)
  - dient zum Manipulieren der const oder volatile-Modifizierer vom Typ des Argumentes arg
  - T und arg müssen vom gleichen Typ sein und dürfen sich nur durch ihre const oder volatile-Modifizierer unterscheiden, sonst ist die Operation ein Fehler
  - die Typumwandlung findet bereits beim Compilieren statt

- static_cast<T>(arg)
  - arbeitet analog der impliziten Typkonvertierung; d.h. gibt es eine implizite Typkonvertierung von Typ von arg nach T, dann ist static_cast korrekt, ansonsten ein Fehler
  - die Typumwandlung findet ebenfalls beim Compilieren statt

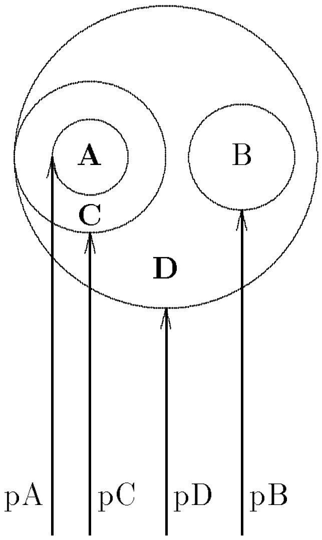
- reinterpret_cast<T>(arg)
  - arbeitet analog der expliziten Typkonvertierung; d.h. ist ein besserer Ersatz für den oft implementationsabhängig funktionierenden Casting-Operator (T)arg bzw. (Funktionsnotation) T(arg)
  - es können unvollständige Typen (z.B. deklarierte, aber noch nicht definierte Klassen) verwendet werden
  - die Typumwandlung findet beim Compilieren statt

- dynamic_cast<T>(arg)
  - dient zur Konvertierung eines Zeigers oder einer Referenz auf ein Objekt einer Basisklasse in einen Zeiger bzw. Referenz auf ein Objekt einer abgeleiteten Klasse, wenn das referenzierte Objekt tatsächlich vom abgeleiteten Typ ist
  - Konvertierungen in der Ableitungshierarchie nach oben (statische) werden bereits beim Compilieren eingesetzt und funktionieren analog static_cast
  - Konvertierungen in der Ableitungshierarchie nach unten (dynamische) werden erst zur Laufzeit berechnet

- arg und T müssen entweder beide einen Pointer oder beide einen Referenz-Typen haben
- wenn arg und T Pointer sind, und die Konvertierung kann nicht durchgeführt werden, wird ein NULL-Zeiger zurückgeliefert
- wenn arg und T Referenzen sind, und die Konvertierung kann nicht durchgeführt werden, wird in der Regel eine Exception (bad_cast) ausgelöst
- Sonderfall: wenn T ein void* ist, muß arg ebenfalls ein Pointer sein; in dem Fall wird ein Pointer geliefert, der auf das komplette Objekt zeigt, also das Objekt der Klasse, die sich in der Hierarchie am weitesten unten befindet

6.4 Beispiel mainRTTI_1

Die folgende Skizze soll die Anwendung der verschiedenen Konvertierungsoperatoren im Programm verdeutlichen:



pD zeigt auf ein Objekt der Klasse D. Die Klasse D hat zwei direkte Basisklassen, B und C und eine indirekte Basisklasse, A, da C von A abgeleitet ist. Somit können zwischen den Pointern pA, pB, pC und pD z.B. folgende korrekte Typkonvertierungen formuliert werden:

pD → pA	implizit	statisch	static_cast<A*>(pD)
pD → pC	implizit	statisch	static_cast<C*>(pD)
pC → pA	implizit	statisch	static_cast<A*>(pC)
pA → pC	explizit	dynamisch	dynamic_cast<C*>(pA)
pA → pD	explizit	dynamisch	dynamic_cast<D*>(pA)
pC → pD	explizit	dynamisch	dynamic_cast<D*>(pC)
pB → pA	explizit	dynamisch	dynamic_cast<A*>(pB)
pB → pC	explizit	dynamisch	dynamic_cast<C*>(pB)

Der Quelltext des Beipielprogrammes (File mainRTTI_1.cc):

```
#include <typeinfo>
#include <iostream.h>

class A {
  int a;
public:
  virtual int blaA() = 0;
};
class C: public A {
  int c;
};
class B {
  int b;
public:
  virtual int blaB() = 0;
};
class D: public C, public B {
  int d;
public:
  virtual int blaA(){return 0;};
  virtual int blaB(){return 0;};
};

int main()
{
  D d;
  A* pA = &d; // implizite Konvertierung
  B* pB = &d; // implizite Konvertierung
  C* pC = &d; // implizite Konvertierung
  D* pD = &d;

  cout
  << "pA = " << hex << pA << endl
  << "pB = " << hex << pB << endl
  << "pC = " << hex << pC << endl
  << "pD = " << hex << pD << endl
  << "static_cast<A*>(pD) = " << hex << static_cast<A*>(pD) << endl
  << "static_cast<B*>(pD) = " << hex << static_cast<B*>(pD) << endl
  << "static_cast<C*>(pD) = " << hex << static_cast<C*>(pD) << endl
  << "static_cast<A*>(pC) = " << hex << static_cast<A*>(pC) << endl
  << "dynamic_cast<C*>(pA) = " << hex << dynamic_cast<C*>(pA) << endl
  << "dynamic_cast<D*>(pA) = " << hex << dynamic_cast<D*>(pA) << endl
  << "dynamic_cast<D*>(pC) = " << hex << dynamic_cast<D*>(pC) << endl
  << "dynamic_cast<D*>(pB) = " << hex << dynamic_cast<D*>(pB) << endl
  << "dynamic_cast<A*>(pB) = " << hex << dynamic_cast<A*>(pB) << endl
  << "dynamic_cast<C*>(pB) = " << hex << dynamic_cast<C*>(pB) << endl
  << "dynamic_cast<B*>(pA) = " << hex << dynamic_cast<B*>(pA) << endl
```

```
  << "dynamic_cast<B*>(pC) = " << hex << dynamic_cast<B*>(pC) << endl;
}
```
Folgende Ausgabe kann mit dem Beispielprogramm erzeugt werden:

```
pA = 0xbffff858
pB = 0xbffff864
pC = 0xbffff858
pD = 0xbffff858
static_cast<A*>(pD) = 0xbffff858
static_cast<B*>(pD) = 0xbffff864
static_cast<C*>(pD) = 0xbffff858
static_cast<A*>(pC) = 0xbffff858
dynamic_cast<C*>(pA) = 0xbffff858
dynamic_cast<D*>(pA) = 0xbffff858
dynamic_cast<D*>(pC) = 0xbffff858
dynamic_cast<D*>(pB) = 0xbffff858
dynamic_cast<A*>(pB) = 0xbffff858
dynamic_cast<C*>(pB) = 0xbffff858
dynamic_cast<B*>(pA) = 0xbffff864
dynamic_cast<B*>(pC) = 0xbffff864
```

Aufgabe:
Beachten Sie besonders den Offset des Pointers `pB` gegenüber den anderen Pointern auf die Subobjekte von `D` in der Ausgabe des Programmbeispieles! Wodurch kommt dieser Offset zustande?

6.5 Beispiel `mainZGeoObjRTTI`

Wir arbeiten weiter am `ZGeoObj`-Beispiel aus dem vorangegangenen Kapitel.

- zuerst definieren wir für die `ZGeoObj`-Klasse einen Subscription-Operator für den direkten Zugriff auf die im Container gespeicherten Pointer auf `GeoObj`-Objekte (File ZGeoObjRTTI.cc):

```
#include <typeinfo>
#include "ZGeoObjRTTI.h"

void ZGeoObj::set(GeoObj& g)
{
  if(anzahl < MAX)
    go[anzahl++] = &g;
}

float ZGeoObj::flaeche() const
{
  float fl = .0;
  for(int i = 0; i < anzahl; i++)
    fl +=  go[i] -> flaeche();
  return fl;
```

```
}

GeoObj*  ZGeoObj::operator[](int i) const
{
  if(i < anzahl)
    return go[i];

  return 0;
}
```

In den nachfolgenden Punkten werden die verschiedenen Abschnitte der `main()`-Routine, File mainZGeoObjRTTI.cc, vorgestellt:

- es werden zwei Container, `z1` und `z2`, angelegt; zum Container `z1` wird das Rechteck `r1` und das Dreieck `d` hinzugefügt; zum Container `z2` wird der Container `z1` und das Rechteck `r2` hinzugefügt:

```
#include <typeinfo>
#include <iostream.h>
#include "Rechteck_4.h"
#include "Dreieck_2.h"
#include "ZGeoObjRTTI.h"

int main()
{
  Rechteck r1(Punkt(0., 0.), Punkt(6., 5.));
  Rechteck r2(Punkt(6., 0.), Punkt(15., 5.));
  Dreieck   d(Punkt(0., 5.), Punkt(6., 5.), Punkt(3., 9.));
  ZGeoObj z1, z2;

  z1.set(r1);
  z1.set(d);
  z2.set(z1);
  z2.set(r2);

  ...
```

- nun soll der Inhalt der beiden Container ausgegeben werden in der Form "`[`*index im container*`]` `<`*typbezeichnung*`>` `Flaeche:` *flaeche*":

```
  ...

  cout << "Inhalt des Containers z1:" << endl;
  for(int i = 0; z1[i]; i++)
  {
    cout << "[" << i << "] " << "<" << typeid(*z1[i]).name() << "> ";
    cout << "Flaeche: " << z1[i]->flaeche() << endl;
  }
```

```
cout << "[z1] " << "<" << typeid(z1).name() << "> ";
cout << "Flaeche: " << z1.flaeche() << endl;

cout << "Inhalt des Containers z2:" << endl;
for(int i = 0; z2[i]; i++)
{
  cout << "[" << i << "] " << "<" << typeid(*z2[i]).name() << "> ";
  cout << "Flaeche: " << z2[i]->flaeche() << "\n";
}
cout << "[z2] " << "<" << typeid(z2).name() << "> ";
cout << "Flaeche: " << z2.flaeche() << endl;

...
```

- dieser Abschnitt des Programmes sollte folgende Ausgaben erzeugen:

```
Inhalt des Containers z1:
[0] <TD.8Rechteck> Flaeche: 30
[1] <TD.7Dreieck> Flaeche: 12
[z1] <TD.7ZGeoObj> Flaeche: 42
Inhalt des Containers z2:
[0] <TD.7ZGeoObj> Flaeche: 42
[1] <TD.8Rechteck> Flaeche: 45
[z2] <TD.7ZGeoObj> Flaeche: 87
```

- nun können die (dynamischen) Typen der Objekte in den Containern abgefragt und untereinander verglichen werden:

```
...

cout << "Abfragen:" << endl;
cout << "(typeid(int) == typeid(char))? "
     << ((typeid(int) == typeid(char))? "true" : "false") << endl;
cout << "(typeid(*z1[0]) == typeid(Rechteck))? "
     << ((typeid(*z1[0]) == typeid(Rechteck))? "true" : "false") << endl;
cout << "(typeid(*z1[1]) == typeid(Dreieck))? "
     << ((typeid(*z1[1]) == typeid(Dreieck))? "true" : "false") << endl;
cout << "(typeid(*z1[0]) == typeid(Dreieck))? "
     << ((typeid(*z1[0]) == typeid(Dreieck))? "true" : "false") << endl;
cout << "(typeid(z1) == typeid(GeoObj))? "
     << ((typeid(z1) == typeid(GeoObj))? "true" : "false") << endl;
cout << "(typeid(z1) == typeid(ZGeoObj))? "
     << ((typeid(z1) == typeid(ZGeoObj))? "true" : "false") << endl;
cout << "(typeid(*z2[0]) == typeid(ZGeoObj))? "
     << ((typeid(*z2[0]) == typeid(ZGeoObj))? "true" : "false") << endl;

...
```

- dieser Abschnitt des Programmes sollte folgende Ausgaben erzeugen:

```
Abfragen:
(typeid(int) == typeid(char))? false
(typeid(*z1[0]) == typeid(Rechteck))? true
(typeid(*z1[1]) == typeid(Dreieck))? true
(typeid(*z1[0]) == typeid(Dreieck))? false
(typeid(z1) == typeid(GeoObj))? false
(typeid(z1) == typeid(ZGeoObj))? true
(typeid(*z2[0]) == typeid(ZGeoObj))? true
```

- als nächstes versuchen wir einige dynamische Typkonvertierungen, einschließlich einer fehlerhaften:

```
  ...

  cout << "dynamisches Casting:" << endl;
  cout << "dynamic_cast<Rechteck*>(z1[0])="
       << hex << (dynamic_cast<Rechteck*>(z1[0])) << endl;
  cout << "dynamic_cast<Dreieck*>(z1[1])="
       << hex << (dynamic_cast<Dreieck*>(z1[1])) << endl;
  cout << "dynamic_cast<Dreieck*>(z1[0])="
       << hex << (dynamic_cast<Dreieck*>(z1[0])) << endl;
  cout << "dynamic_cast<ZGeoObj*>(z2[0])="
       << hex << (dynamic_cast<ZGeoObj*>(z2[0])) << endl;

  ...
```

- dieser Abschnitt des Programmes sollte die folgenden Ausgaben erzeugen, wobei das Nullresultat auf der 3. Zeile die versuchte fehlerhafte Konvertierung kennzeichnet:

```
dynamisches Casting:
dynamic_cast<Rechteck*>(z1[0])=0xeffff7a0
dynamic_cast<Dreieck*>(z1[1])=0xeffff748
dynamic_cast<Dreieck*>(z1[0])=0x0
dynamic_cast<ZGeoObj*>(z2[0])=0xeffff700
```

- zum Schluß sollen die Nachteile des traditionellen Casting-Operators deutlich gemacht werden; es wird ein Laufzeitfehler durch falsches Casting erzeugt:

```
  ...

  cout << "Laufzeitfehler bzw. Absturz durch "
       << "falsches explizites Casting:" << endl;

// richtig: *z1[0] ist ein Rechteck
```

```
  Rechteck* pre = (Rechteck*)(z1[0]);
// richtig: kopiere Rechteck
  Rechteck copyre = *pre;
// falsch: *z1[0] ist kein Dreieck
  Dreieck* pdr = (Dreieck*)(z1[0]);
// falsch: kopiere ein Rechteck ueber ein Dreieck
  Dreieck copydr = *pdr;
  cout << "Flaechen: <Rechteck>: "
       << flush << copyre.flaeche()
       << flush << " <Dreieck>: "
       << flush << copydr.flaeche() << endl;
}
```

- Im Gegensatz zu den ersten beiden Anweisungen dieses Abschnitts ist die Konvertierung von `z1[0]` (Typ: `GeoObj*`) in einen `Dreieck*` unzulässig, da `z1[0]` in Wirklichkeit auf ein Rechteck zeigt. Der Cast liefert aber einen Pointer zurück, der nicht NULL ist. Beim Kopieren des Objektes kommt der Fehler zum Tragen: Ein Dreieck kann nicht korrekt durch die Daten eines Rechtecks initialisiert werden und die nachfolgende Berechnung der Fläche bringt irreguläre Resultate:

```
Laufzeitfehler bzw. Absturz durch falsches explizites Casting:
Flaechen: <Rechteck>: 30 <Dreieck>: 3.96094e+29
```

7 Namespaces in C++

7.1 Motivation

Ein C++-Projekt besteht oft aus mehreren Quelldateien, die von mehreren Entwicklern bearbeitet werden. Die globalen Namen, die in den verschiedenen Quelldateien verwendet werden, gehören traditionell demselben Namensbereich `namespace` an. Werden die Module dann zum Endprodukt gelinkt, gibt es oft Linkerfehler aufgrund von Namenskonflikten (*multiple definition of 'symbol'*).
Durch die Einführung verschiedener Namensbereiche entsprechend dem Namensbereichmechanismus von C++ kann die Namensgebung geplant und Namenskonflikte somit vermieden werden. Bestehende Konflikte können sogar nachträglich noch behoben werden.
Folgende grundlegende Vorgehensweise:

1. Definieren eines Namensbereiches mit dem Schlüsselwort `namespace`

2. Zugriff auf Elemente eines Namensbereiches entweder explizit (`scope operator ::`) oder implizit nach `using`-Deklaration oder `using`-Anweisung (Unterschied)

7.2 Definieren von Namensbereichen

- ein Namensbereich wird durch eine `namespace`-Definition festgelegt:

```
namespace F {
  float x = 9;
}
```

- Namensbereiche können geschachtelt werden:

```
namespace G {
  using namespace F;
  float y = 2.0;
  namespace INNER_G {
    float z = 10.01;
  }
}
```

- die Definition eines Namensbereiches kann beliebig erweitert werden; Achtung: benötigte Erweiterungen vor der Benutzung des Namensbereiches vornehmen, der Compiler kennt bei einer `using`-Anweisung nur die bisher zum Namensbereich gehörigen Namen; Beispiel:

```
using namespace F; // xx wird hier nicht bekannt

namespace F {
  double xx = 99;
}
```

```
using namespace F; // hier wird xx bekannt
```

  - nach der zweiten Definition für F gehören dem Namensbereich F die zwei Symbole `float x` und `double xx` an
  - die Anweisung `using namespace F;` vor der zweiten Definition macht allerdings hier nur den Namen `float x` bekannt

- für definierte Namensbereiche können Aliase eingeführt werden:

```
namespace ALIAS_G = G;
namespace ALIAS_INNER_G = G::INNER_G;
```

- unbenannte Namensbereiche ersetzen die traditionellen `static`-Definitionen, um Namen nur innerhalb der Übersetzungseinheit bekannt zu machen:

```
#include <iostream.h>
extern void func(void);
namespace {
  float pi = 3.14;
}
int main() {
  cout << "pi = " << pi << endl;
  func();
}
```

  - innerhalb der in einer anderen Übersetzungseinheit definierten (`externen`) Funktion `func` kann auf das in diesem unbenannten Namensbereich vorhandene `pi` nicht zugegriffen werden
  - Die traditionelle `static`-Definition wird mit Einführung des Namensbereichmechanismus von C++ nicht mehr benötigt und sollte auch nicht mehr verwendet werden.

- eine Klassendefinition definiert zugleich auch einen Namensbereich, der allerdings nachträglich nicht mehr erweitert werden kann

7.3 Zugriff auf Elemente eines Namensbereiches

Für den Zugriff auf Elemente eines Namensbereiches gibt es folgende drei Möglichkeiten:

1. explizite Zugriffsqualifizierung, Beispiel:
   `G::INNER_G::z = 10.01;`

2. `using`-Deklaration, fügt den deklarierten Namen in den lokalen Namensbereich ein; Beispiele:

```
using G::INNER_G::z;
using A::a;
z = 10.01;
a = 111;
```

- using-Deklarationen fügen Aliase für die originalen Namen in den lokalen Sichtbarkeitsbereich ein; keine Beeinflussung des Typs, Linkage etc.
- redundante using-Deklarationen sind erlaubt

3. using-Anweisung, macht alle Namen des angegebenen Namensbereiches im lokalen verwendbar; Einfügung erfolgt erst mit Verwendung; Beispiele:

```
int main() {
  using namespace G;
  using namespace G::INNER_G;
  ...
}
```

- lokale Deklarationen haben stets Vorrang gegenüber denen durch using-Anweisungen eingefügten; siehe `float x` im nachfolgenden Beispiel
- using-Anweisungen können in Klassendefinitionen nicht verwendet werden

7.4 Beispiel

```
#include <iostream.h>
class A {
public:
  static int a;
  int print(){return a;}
};
namespace F {
  float x = 9;
}
namespace G {
  using namespace F;
  float y = 2.0;
  namespace INNER_G {
    float z = 10.01;
  }
}
int A::a = 0;

int main() {
  using A::a;
```

```
  using namespace G;
  using namespace G::INNER_G;
  a = 111;
  float x = 19.1;
  A aa;
  cout << "x = " << x << endl;
  cout << "y = " << y << endl;
  cout << "z = " << z << endl;
  cout << "A::a = " << aa.print() << endl;
  return 0;
}

/* Ausgabe:
x = 19.1
y = 2
z = 10.01
A::a = 111 */
```

8 Fehlerbehandlung

8.1 Nachteile traditioneller Fehlerbehandlung

Laufzeitfehler:

- Anforderung einer Ressource (Hauptspeicher, File-Handle, IPC, ...) konnte nicht erfüllt werden

- Auswirkungen falscher und nicht gefilterter Benutzereingaben

- Auswirkungen von allg. Programmierfehlern (z.B. Dereferenzierung ungültiger Pointer)

- falsche oder unerwartete Funktion anderer Softwarekomponenten oder Hardware

häufige Situation:

- ein Anwenderprogramm wird auf der Basis einer oder mehrerer Bibliotheken erstellt

- der Autor einer Bibliothek ist u.U. in der Lage, Laufzeitfehler zu entdecken (falsche Parameterübergaben, Ressourcensituation), aber er weiß nicht, wie er darauf reagieren soll

- der Autor des Anwenderprogrammes kann wissen, wie er auf Laufzeitfehler reagieren soll, jedoch ist er oft nicht in der Lage, diese Fehler zu entdecken

eine traditionelle Fehlerbehandlung hätte folgende Möglichkeiten:

1. nach Entdeckung des Fehlers Programm sofort terminieren
   – in der Regel nicht das, was der Anwendungsprogrammierer will

2. Fehler durch Rückgabe eines Fehlercodes kennzeichnen
   – setzt voraus, daß die Funktion Fehlercodes zurückliefern kann (Operatoren?!) und daß der Anwendungsprogrammierer systematisch die Rückgabewerte testet (negativ für: Programmgröße, Laufzeiteffizienz, Programmlogik))

3. Programm im Fehlerzustand belassen
   – Programm wird nachfolgend nicht mehr korrekt arbeiten und bedienbar sein, Folgefehler, Programmabsturz (Systemabsturz?)

4. vorher definierte Handlerfunktion aufrufen – diese hat dann wieder die Möglichkeiten 1-3 zur Verfügung, um auf den Fehler zu reagieren

8.2 Ausnahmen (Exceptions) und deren Behandlung (-handling) in C++

8.2.1 Grundlagen

- eine Ausnahme in C++ kennzeichnet den Sachverhalt, daß ein Block bzw. eine Funktion nicht auf regulärem Wege (z.B. return) verlassen werden soll

- diese Anforderung besteht oft, wenn nach Bemerken eines Fehlerzustandes (z.B. dynamischer Speicher konnte nicht allokiert werden) die vollständige Bearbeitung des nachfolgenden Codes (z.B. Konstruktion eines Objektes) sinnlos ist und sogar zum Programmabsturz führt

- an der Fehlerstelle (z.B. Bibliotheksfunktion) wird ein spezielles Ausnahmeobjekt erzeugt und als Ausnahme ausgeworfen (throw)

- Ausnahmeobjekt kann Informationen über Ort und Ursache des Fehlers aufnehmen

- das Ausnahmeobjekt kann an geeigneter Stelle aufgefangen (catch) und ausgewertet werden; wird es nicht aufgefangen, folgt der Programmabbruch

- Ausnahmen (Exceptions) in C++ sind synchron und vom Programm selbst generiert und daher nicht mit Ereignissen wie Interrupts bzw. Signalen vergleichbar, die außerhalb des Programmes erzeugt wurden. Letztere können aber (zumindest teilweise) durchaus auf C++-Ausnahmen und deren Behandlung abgebildet werden.

- Schlüsselwörter `try`, `throw` und `catch`

- Syntax:

```
try
{
  try_some_code();
}
catch(Exception1& Ex)
{
  // Handler fuer Exceptions vom Type Exception1
  // Ex auswerten und Fehler behandeln
}
catch(Exception2& Ex)
{
  // Handler fuer Exceptions vom Type Exception2,
  // sofern sie nicht durch den (die) vorherigen
  // Handler aufgefangen werden konnten
}
catch(...)
{
```

```
  // Handler fuer Exceptions, die nicht durch die
  // vorherigen Handler aufgefangen werden konnten
}

// nach dem Behandeln der ausgeworfenen Exception
// wird hier fortgesetzt
```

8.2.2 Vorteile gegenüber traditioneller Techniken

- Behandlung von Fehlern wird dem Anwendungsprogrammierer überlassen; er hat die Möglichkeiten, den Benutzer zu informieren
- Exception Handling entspricht der oo Philosophie: Hierarchien von Ausnahmeklassen können rel. unabhängig vom eigentlichen Code entwickelt werden
- Programm wird robuster, eine ausgeworfene Exception muß aufgefangen und entsprechend behandelt werden, sonst erfolgt Programmabbruch; Vermeidung unkontrollierter Programmzustände nach Auftreten eines Fehlers

8.2.3 Deklarieren von Exceptions

- ein Exception-Objekt überträgt Informationen von der Stelle, an der sie ausgelöst wurde zu der Stelle, wo sie aufgefangen wird
- Exception-Objekte werden aus regulären C++-Klassen instanziiert
- es kann Ableitungshierarchien von Exception-Klassen geben
- Exception-Objekte können sich auch polymorph verhalten (virtuelle Funktionen!)
- werden Exception-Objekte per Value aufgefangen, sollten sie einen Copykonstruktor besitzen
- Beispiel: File xmsg.h

```
#ifndef __XMSG_H
#define __XMSG_H

#include "String_6.h"

class xmsg
{
private:
    const String str;

public:
    xmsg(const String& _str)
```

```
    : str(_str){}

    const String& why() const
    {
      return str;
    }
};

#endif
```

8.2.4 Auslösen einer Exception

Eine Exception wird an der Fehlerstelle durch `throw` *exception-object* ausgelöst. Die Informationen über den Fehler werden dem Konstruktor von *exception-object* übergeben.

8.2.5 Behandeln einer Exception

- Behandlungsroutinen schließen sich unmittelbar an den `try`-Block an
- Behandlungsroutinen sind durch das Schlüsselwort `catch` gekennzeichnet. `catch` kann nach dem `try`-Block oder nach einem anderen `catch` vorkommen
- ein Statement `catch(T Ex)` wird ein Exception-Objekt `Ex` akzeptieren, wenn folgendes gilt:
  - der Typ von `Ex` ist `T`
  - der Typ `T` ist eine zugreifbare Basisklasse des Types von `Ex` (Ist-Relation)
  - der Type `T` ist ein Zeigertyp und `Ex` ist von einem Zeigertyp, der zu `T` durch Standardzeigerkonvertierung konvertiert werden kann
- vom Ort des Auswurfes des Exception-Objektes aus wird solange der Stack abgebaut, bis ein geeigneter Handler für das ausgeworfene Objekt gefunden wird; dabei werden für die automatischen Objekte die Destruktoren gerufen
- wird kein geeigneter Handler gefunden, erfolgt ein Aufruf von `terminate()`, die in der voreingestellten Version ihrerseits `abort()` aufruft
  Beispiel (Hinweis: Klasse `xalloc` ist von `xmsg` abgeleitet, siehe auch Files xalloc.h, except1.cc und except2.cc):

```
#include "xalloc.h"
size_t _last_allocation_size;
void *operator new[] (size_t size) {
  void* p;
  _last_allocation_size = size;
  while((p = malloc(size)) == 0) {
```

```
    if(__new_handler)
      (*__new_handler)();
    else
      return 0;
  }
  return p;
}

void ex_new_handler() throw(xalloc) {
  // try_to_find_some_memory();
  // if(found_some()) return;
  throw xalloc(_last_allocation_size);
}

int main()
{
  char *p;
  void (*old_handler)() = set_new_handler(&ex_new_handler);
  try {
    p = new char[SEGMENT_SIZE_1];
  }
  catch(xmsg& ex) {
    printf("Exception catched:\n\t%s\n", ex.why());
    p = NULL;
  }
  printf("Allocation result for size=%u: p=%p\n", SEGMENT_SIZE_1, p);
  delete[] p;
  set_new_handler(old_handler);
}
```

8.2.6 Exception-Spezifikationen

- die Spezifikation der Ausnahmen, die eine Funktion auswerfen kann, kann zum Bestandteil der Deklaration gemacht werden, Beispiel:
  `void ex_new_handler() throw(xalloc);`

- die Spezifikation der Exceptions in der Deklaration einer Funktion ist eine Garantie dieser Funktion gegenüber dem aufrufenden Kontext, daß die angegebenen und keine anderen Ausnahmen von dieser Funktion verbreitet werden

- werden dennoch andere Ausnahmen (als die spezifizierten) ausgeworfen, dann erfolgt ein Aufruf von `unexpected()`, die in der voreingestellten Version ihrerseits `terminate()` aufruft

8.2.7 Unerwartete und unbehandelte Exceptions

- falls eine Funktion eine nicht deklarierte Ausnahme auswirft, wird `unexpected()` aufgerufen; `unexpected()` ruft die letzte mittels `set_unexpected()` gesetzte Funktion auf
- falls eine ausgeworfene Ausnahme nicht aufgefangen wird (ernste Situation!), wird `terminate()` aufgerufen; `terminate()` ruft die letzte mittels `set_terminate()` gesetzte Funktion auf
- Voreinstellung: `unexpected()` ruft `terminate()` auf und `terminate()` tt abort()

8.2.8 Ein komplettes Beispiel

- in der traditionellen C-Programmierung wird in der Entwicklungsphase eines Programmes oft vom `assert`-Mechanismus Gebrauch gemacht
- ein Aufruf von `void assert(int expression);` führt im Falle `expression == 0` zu einer Ausgabe der Form
  `Assertion failed: expression, file xyz, line nnn`

Begriff der *Invariante*:

- der Wert der Elemente eines Objektes heißt *Zustand* des Objektes
- Initialisierung sollte ein Objekt in einen 'wohldefinierten' Zustand bringen
- 'wohldefinierter' Zustand eines Objektes soll nach (im strengeren Sinne sogar während) der auf dieses Objekt angewandten Operationen aufrechterhalten werden bis zur Löschung des Objektes (Funktionen, die auf Daten eines Objektes arbeiten und aufgerufen werden können, während der Zustand des Objektes nicht 'wohldefiniert' ist, sollten nicht an der öffentlichen Schnittstelle (`public, protected`) erscheinen)
- die Eigenschaft, die im Zusammenhang mit Design und Implementierung den Zustand eines Objektes wohldefiniert macht, heißt *Invariante*

Praxis:

- ... den Zustand wohldefiniert machen, kann nur der Code des Objektes selbst
- die Invariante kann also zumindest als Code realisiert werden, der den Zustand eines Objektes auf 'Wohldefiniertheit' prüft bzw. vor Abweichungen davon schützt
- häufige Realisierung: Pre- und Postconditions; Precondition prüft Zustand des Objektes und Wert der Argumente beim Eintritt in den Code einer Methode; Postcondition prüft den Zustand des Objektes vor Verlassen der Methode

- Beispiel: File precondition.h

```
#ifndef __PRECONDITION_H
#define __PRECONDITION_H

#include "xmsg.h"

class precondition : public xmsg
{
private:
  static const String MakeString(const char *type,
                                 const char *txt,
                                 const char *file,
                                 int line);
public:
  precondition(const char *txt,
               const char *file,
               int line)
               : xmsg(MakeString("Precondition",txt,file,line)){}
};

#define PRECONDITION(p) PRECONDITIONX(p,#p)

#if __DEBUG < 1
#define PRECONDITIONX(p,s)   ((void)0)
#else
#define PRECONDITIONX(p,s)   \
    if(!(p)) {throw precondition(s,__FILE__,__LINE__);}
#endif

#endif
```

Die `PRECONDITION(p)`-Anweisung ist als Makro realisiert, welches erst wirklich Code erzeugt, wenn die Präprozessorvariable `__DEBUG` einen Wert ≥ 1 hat. Wird Code erzeugt, dann expandiert das Makro an der originalen Quelltextzeile zu einer Überprüfung der Bedingung (Precondition) und wirft bei Verletzung eine `precondition`-Exception aus. Dem Konstruktor des Exception-Objektes wird das Bedingungsstatement als String und zusätzlich der Name des Quelltextfiles und die Nummer der aktuellen Quelltextzeile übergeben.

- File AssocArrayPre.cc

```
#include <typeinfo>
#include <iostream.h>
#include "precondition.h"
#include "AssocArray.h"
```

```
AssocArray::AssocArray(int s)
 : anzahl(s < 1 ? 16: s), frei(0)
{
   vek = new Paar[anzahl];
}
AssocArray::~AssocArray()
{
   delete[] vek;
}

int& AssocArray::operator[](const String& s1)
{
//      PRECONDITION((frei <= anzahl) && s1.length());
   PRECONDITION((frei <= anzahl) && (s1 != "xxx"));

   Paar* pp;

   for(pp = &vek[frei - 1]; vek <= pp; pp--)
      if (s1 == pp->wort)
         return pp->haeufigkeit;
   if (frei == anzahl) {
      Paar* nvek = new Paar[anzahl*2];
      for (int i= 0; i<anzahl; i++)
         nvek[i] = vek[i];
      delete [] vek;
      vek = nvek;
      anzahl *= 2;
   }
   pp = &vek[frei++];
   pp->wort = s1;
   pp->haeufigkeit = 0;
   return pp->haeufigkeit;
}

void AssocArray::print_all()
{
   for (int i=0; i < frei; i++)
      cout << vek[i].wort << ": " << vek[i].haeufigkeit << "\n";

}
```

Die erste (auskommentierte) PRECONDITION-Anweisung testet die Konsistenz der Elemente des AssocArray-Objektes und das String-Argument des Subscription-Operators.
Zu Testzwecken wurde die Precondition dahingehend abgewandelt, das eine Verletzung derselben bereits bei Eingabe einer 'xxx'-Sequenz eintritt.

- File precondition.cc

```
#include <stdio.h>
#include <stdlib.h>
#include <typeinfo>
#include <iostream.h>

#include "precondition.h"
#include "AssocArray.h"
#include "String_6.h"

const String precondition::MakeString(const char *type,
             const char *txt, const char *file, int line)
{
  static char str[0xff];

  sprintf(str, "%s violated in file %s, line %d:\n\t(%s)\n",
          type, file, line, txt);

  return str;
}

int main ()
{
  String buf;
  AssocArray avek(512);
  try
  {
    while (cin >> buf)
      avek[buf]++;
  }
  catch(precondition& ex)
  {
    cout << "Exception catched:\n\t" << ex.why() << endl;
  }
  avek.print_all();
}
```

Die `main()`-Funktion zeigt das Auffangen der `precondition`-Exception. Ausgaben des Programmes könnten z.B. sein:

```
../BEISPIELE > ./precondition
Trabi Wartburg Auto ?
Renault Renault Auto !
Trabi: 1
Wartburg: 1
Auto: 2
?: 1
```

```
Renault: 2
!: 1
../BEISPIELE > ./precondition
Jetzt
die
Exception: xxx
Exception catched:
        Precondition violated in file AssocArrayPre.cc, line 15:
        ((frei <= anzahl) && (s1 != "xxx"))

Jetzt: 1
die: 1
Exception:: 1
```

9 I/O mit Streams

9.1 Grundlagen

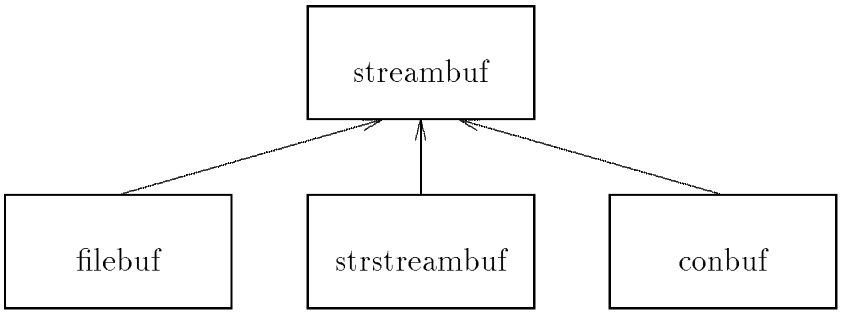
9.1.1 Was ist ein Stream?

Die Sprache C++ besitzt keine eigenen Sprachelemente für Input und Output. Die I/O-Funktionalität kann mit der Sprache selbst elegant genug implementiert werden. Dazu wurde eine Bibliothek geschaffen und standardisiert, die *standard stream input/output library*.
Abstrakt gesehen ist ein Stream ein Datenfluß von einer Quelle (*Produzenten*) zu einem Ziel (*Konsumenten*). In der Begriffswelt von C++ ist ein Ausgabestream ein Objekt, welches mit dem Ziel verbunden ist und ein Eingabestream ein Objekt, welches mit der Quelle verbunden ist. Entsprechend erfolgt die Ausgabe von Daten (Objekten) durch Übergeben dieser an ein Ausgabeobjekt; die Eingabe erfolgt durch Holen von Daten von einem Eingabeobjekt. Die Ausgabe eines Objektes auf einen Stream nennt man auch *Serialisierung des Objektes* zu einer Folge von Textzeichen. Die Eingabe eines Objektes von einem Stream nennt man auch *Wiederherstellung* des Objektes aus seiner serialisierten Darstellung. Die Stream-Bibliothek enthält Klassen (z.B. `ostream`), die als allgemeine Schnittstellen für die Arbeit mit verschiedenen Streams (z.B. `ofstream`, `ostrstream`) verwendet werden können.

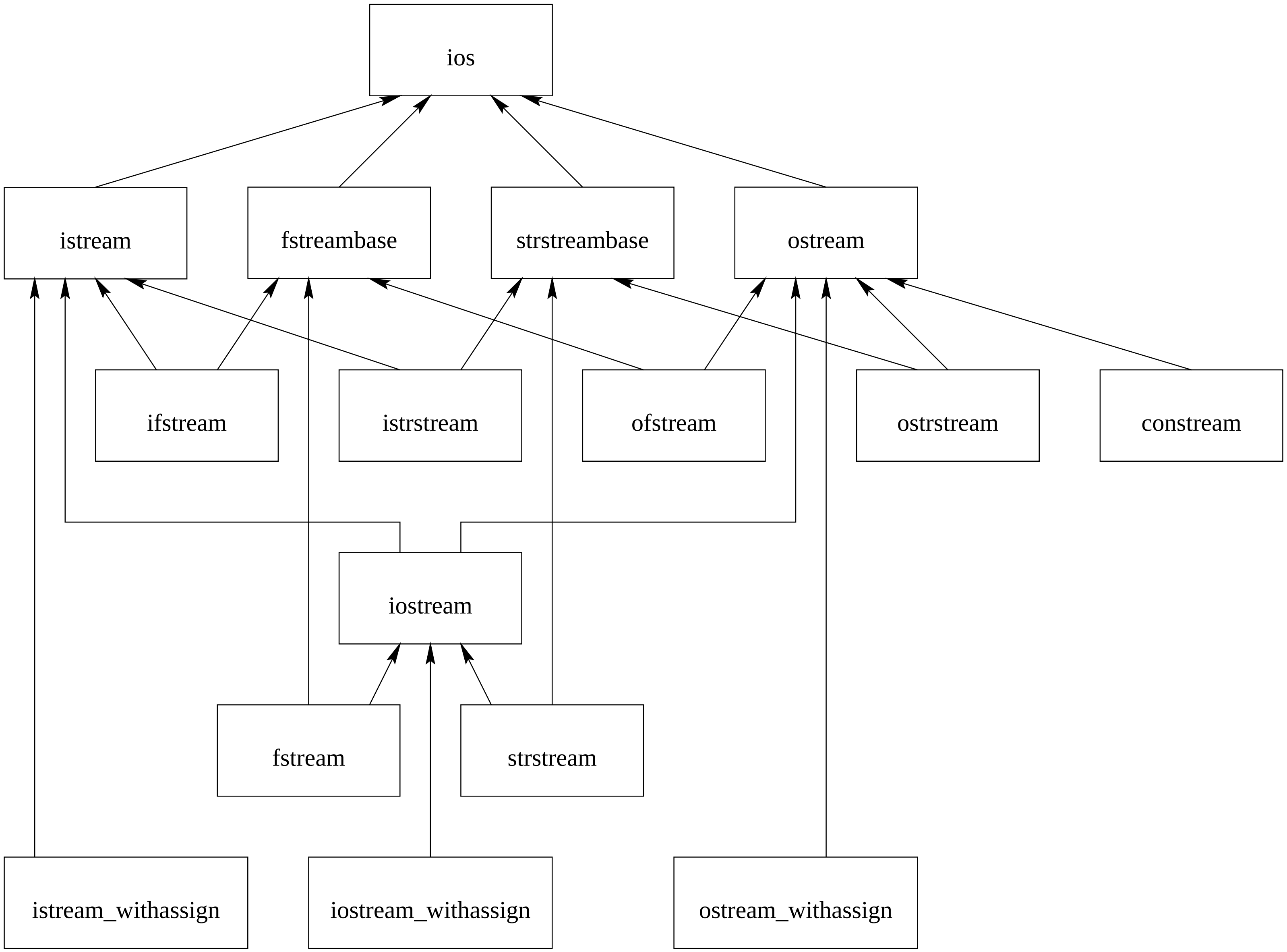
9.2 I/O-Stream-Klassen

Die Stream-Bibliothek besteht aus zwei parallelen Klassenfamilien, der `streambuf`-Hierarchie und der `ios`-Hierarchie. Objekte von Klassen der `streambuf`-Hierarchie werden oft den Objekten der Klassen der `ios`-Hierarchie aggregiert; d.h. `ios`-Klassen haben oft Objekte von `streambuf`-Klassen als direkte oder indirekte (Pointer auf diese) Elemente.

9.2.1 `streambuf`-Hierarchie



- ein `streambuf` ist die Schnittstelle zwischen Hauptspeicher und den Peripheriegeräten
- `streambuf` bietet Methoden zum Puffern der I/O-Daten
- für spezialisierte Streams werden spezialisierte Puffer-Klassen von `streambuf` abgeleitet


ios
istream
fstreambase
strstreambase
ostream
ifstream
istrstream
ofstream
ostrstream
constream
iostream
fstream
strstream
istream_withassign
iostream_withassign
ostream_withassign

- Klasse `ios` und abgeleitete KLassen enthalten Zeiger auf ein `streambuf`-Objekt

- über die `streambuf`-Objekte führen `ios`-Objekte formatierte Ein- und Ausgabe durch

9.2.3 Vordefinierte Streams

Folgende Streamobjekte für die Standard-I/O-Streams sind vordefiniert und in einem C++-Programm global sichtbar:

```
extern istream_withassign cin;  // stdin,  file desc. 0
extern ostream_withassign cout; // stdout, file desc. 1
extern ostream_withassign cerr; // stderr, file desc. 2
extern ostream_withassign clog; // buffered cerr, file desc. 2
```

9.2.4 Stream-Status

- ein Stream hat einen zugehörigen Status, der gewisse Fehlerzustände oder auch das Ende der Datenquelle anzeigt

- Statusbits werden in der Klasse `ios` definiert

- Status kann mittels der Elementfunktionen der Klasse `ios` abgefragt werden:

```
class ios {
  ios& operator=(ios&);  /* Not allowed! */
// ...
  public:
    enum io_state {
goodbit = _IOS_GOOD,
eofbit  = _IOS_EOF,
failbit = _IOS_FAIL,
badbit  = _IOS_BAD };
// ...
    void clear(iostate state = 0);
    int good() const { return _state == 0; }
    int eof()  const { return _state & ios::eofbit; }
    int fail() const { return _state & (ios::badbit|ios::failbit); }
    int bad()  const { return _state & ios::badbit; }
    iostate rdstate() const { return _state; }
    operator void*() const { return fail() ? (void*)0 : (void*)(-1); }
    int operator!() const { return fail(); }
// ...
};
```

Beispiel (streams_1.cc):

```
#include <iostream.h>

void printStateOn(ostream& ros, ios& rios) {
    ros << "Streamstatus: " << hex << rios.rdstate() << dec << endl;
    ros << "Stream ist";
    if(rios.good()) ros << " good";
    if(rios.eof())  ros << " eof";
    if(rios.fail()) ros << " fail";
    if(rios.bad())  ros << " bad";
    ros << endl;
}
int main() {
    char c;
    int z = 0;

    cout << "erster Leseversuch von cin:"  << endl;
    while(cin >> c) {
        cout << c;
        z = z + 1;
    }
    cout << endl << "Anzahl bis hierhin kopierter Zeichen: " << z << endl;
    printStateOn(cout, cin);

    cout << "zuruecksetzen:" << endl;
    cin.clear();
    printStateOn(cout, cin);

    cout << "nochmaliger Leseversuch von cin:" << endl;
    while(cin >> c) {
        cout << c;
        z = z + 1;
    }
    cout << endl << "Anzahl insgesamt kopierter Zeichen: " << z << endl;
    printStateOn(cout, cin);
}
```

9.3 Stream-Ausgabe

9.3.1 ostream-Klassen

```
class ostream : virtual public ios {
// ...
  public:
// ...
    ostream& flush();
    ostream& put(char c);
    ostream& write(const char *s, streamsize n);
```

```
    ostream& seekp(streampos);
    ostream& seekp(streamoff, _seek_dir);
    streampos tellp();

    ostream& operator<<(char c);
    ostream& operator<<(const char *s);
    ostream& operator<<(int n);
    ostream& operator<<(long n);
    ostream& operator<<(short n);
    ostream& operator<<(bool b);
    ostream& operator<<(float n);
    ostream& operator<<(double n);
    ostream& operator<<(__omanip func) { return (*func)(*this); }
    ostream& operator<<(__manip func) {(*func)(*this); return *this;}
// ...
};
```

Zur Streamausgabe werden Objekte von von `ostream` abgeleiteten Klassen benutzt.

- die `seekp` und `tellp`-Funktionen dienen der Positionierung bzw. Abfrage der Position zum Schreiben in einen Stream

- für eingebaute Typen existieren `operator<<`-Funktionen als Memberfunktionen innerhalb `ostream`

- für selbstdefinierte Typen muß der globale `operator<<` überladen werden

9.3.2 Ausgabe eingebauter Typen

- `ostream` verfügt über eine Menge von `operator<<`-Funktionen mit folgender Syntax:
  `ostream& operator<<(<T>);`
  wobei entsprechend viele überladene Operatoren existieren, sodaß für `<T>` sämtliche eingebauten Typen, sowie deren `signed` und `unsigned`-Varianten eingesetzt werden können

- die `operator<<`-Funktionen geben jeweils eine Referenz auf den `ostream` zurück, sodaß der `<<`-Operator auch verkettet eingesetzt werden kann, Beispiel:
  `char c = 'A'; cout << c << int(c);`
  wird interpretiert als
  `char c = 'A'; (cout.operator<<(c)).operator<<(int(c));`

9.3.3 Ausgabe selbstdefinierter Typen

- Ausgabe selbstdefinierter Typen wird auf Ausgabe eingebauter Typen (der Elemente) zurückgeführt

- hierbei wird der globale Operator `operator<<` entsprechend überladen; folgende Notation kommt zum Einsatz:
  `ostream& operator<<(ostream&, const <T>&);`
  wobei `<T>` für den realen, selbstdefinierten Typ steht

- die jeweilige globale Operatorfunktion wird in der zugehörigen Klasse für den Zugriff auf private Elemente oft als `friend` deklariert, Beispiel (streams_2.cc):

```
#include <iostream.h>

class Complex {
  double re, im;

public:
  Complex(double _re = 0, double _im = 0)
  : re(_re), im(_im){}

friend ostream& operator<<(ostream&, const Complex&);
};
ostream& operator<<(ostream& ros, const Complex& rc) {
  return ros << '(' << rc.re << ',' << rc.im << ')';
}

int main() {
    Complex z(1.1, 2.2);
    cout << z << endl;
}
```

9.4 Stream-Eingabe

9.4.1 istream-Klassen

```
class istream : virtual public ios {
public:
    istream& get(char& c);
    istream& get(char* ptr, int len, char delim = '\n');
    istream& read(char *ptr, streamsize n);
    int peek();
    istream& seekg(streampos);
    istream& seekg(streamoff, _seek_dir);
    streampos tellg();
    istream& putback(char ch);

    istream& operator>>(char*);
    istream& operator>>(char& c);
    istream& operator>>(int&);
    istream& operator>>(long&);
```

```
    istream& operator>>(short&);
    istream& operator>>(bool&);
    istream& operator>>(float&);
    istream& operator>>(double&);
    istream& operator>>( __manip func) {(*func)(*this); return *this;}
    istream& operator>>(__imanip func) { return (*func)(*this); }
// ...
};
```

Zur Streameingabe werden Objekte von von `istream` abgeleiteten Klassen benutzt.

- die `seekg` und `tellg`-Funktionen dienen der Positionierung bzw. Abfrage der Position zum Lesen aus einem Stream
- die `peek()`-Funktion liefert das nächste Zeichen aus einem Stream, ohne daß es aus ihm entfernt wird (nächste Leseoperation liefert dieses Zeichen wieder)
- die `putback()`-Funktion schreibt ein ungewolltes Zeichen zurück in den Stream, sodaß es bei der nächsten Leseoperation nochmals gelesen wird
- für eingebaute Typen existieren `operator>>`-Funktionen als Memberfunktionen innerhalb `istream`
- für selbstdefinierte Typen muß der globale `operator>>` überladen werden

9.4.2 Eingabe eingebauter Typen

- analog `ostream` verfügt auch `istream` über eine Menge von - diesmal - `operator>>`-Methoden mit folgender Syntax:
  `istream& operator>>(<T>&);`
  wobei entsprechend viele überladene Operatoren existieren, sodaß für `<T>` sämtliche eingebauten Typen, sowie deren `signed` und `unsigned`-Varianten eingesetzt werden können
- die `operator>>`-Funktionen geben jeweils eine Referenz auf den `istream` zurück, sodaß der `>>`-Operator ebenfalls verkettet eingesetzt werden kann

9.4.3 Eingabe selbstdefinierter Typen

- analog der Ausgabe wird die Eingabe selbstdefinierter Typen auf Eingabe eingebauter Typen (der Elemente) zurückgeführt, die einzelnen gelesenen Werte müssen in geeigneter Weise den Elementen zugewiesen werden
- hierbei wird der globale Operator `operator>>` entsprechend überladen; folgende Notation kommt zum Einsatz:
  `istream& operator>>(istream&, <T>&);`
  wobei `<T>` für den realen, selbstdefinierten Typ steht

- die jeweilige globale Operatorfunktion wird in der zugehörigen Klasse für den Zugriff auf private Elemente oft als `friend` deklariert, Beispiel (streams_3.cc):

```
#include <iostream.h>

class Complex
{
  double re, im;

public:
  Complex(double _re = 0, double _im = 0)
  : re(_re), im(_im){}

friend ostream& operator<<(ostream&, const Complex&);
friend istream& operator>>(istream&, Complex&);
};

ostream& operator<<(ostream& ros, const Complex& rc)
{
  return ros << '(' << rc.re << ',' << rc.im << ')';
}
istream& operator>>(istream& ris, Complex& rc)
{
  return ris >> rc.re >> rc.im;
}

int main()
{
    Complex z;

    cout << "Eingabe komplexer Zahl: <Realteil><Leerz.><Imag.-Teil>" << endl;
    cin >> z;

    cout << z << endl;
}
```

9.4.4 Binden von Streams

Die `tie()`-Elementfunktion von `ios` dient zum Verknüpfen von `istream`'s und `ostream`'s. Beispielsweise ist `cout` an `cin` geknüpft mittels:
`cin.tie(&cout);`
Die Verknüpfung bewirkt, daß `cout` immer erst geleert (*flushed*) wird, bevor das Programm auf eine nachfolgende Eingabe von `cin` wartet.

9.5 Formatierung

9.5.1 Klasse ios

```
class ios {
// ...
public:
  enum {
    skipws=_IO_SKIPWS,
    left=_IO_LEFT, right=_IO_RIGHT, internal=_IO_INTERNAL,
    dec=_IO_DEC, oct=_IO_OCT, hex=_IO_HEX,
    showbase=_IO_SHOWBASE, showpoint=_IO_SHOWPOINT,
    uppercase=_IO_UPPERCASE, showpos=_IO_SHOWPOS,
    scientific=_IO_SCIENTIFIC, fixed=_IO_FIXED,
    unitbuf=_IO_UNITBUF, stdio=_IO_STDIO };
  enum { // Masks.
    basefield=dec+oct+hex,
    floatfield = scientific+fixed,
    adjustfield = left+right+internal
  };
// ...
  int width() const;
  int width(int val);
  char fill() const;
  char fill(char newf);
  int precision() const;
  int precision(int newp);
  fmtflags flags() const;
  fmtflags flags(fmtflags new_val);
  fmtflags setf(fmtflags val) {
    fmtflags oldbits = _flags;
    _flags |= val; return oldbits; }
  fmtflags setf(fmtflags val, fmtflags mask) {
    fmtflags oldbits = _flags;
    _flags = (_flags & ~mask) | (val & mask); return oldbits; }
  fmtflags unsetf(fmtflags mask) {
    fmtflags oldbits = _flags;
    _flags &= ~mask; return oldbits; }
// ...
};
```

Die Klasse `ios` kontrolliert die Verbindung zwischen dem `i/ostream` und dem Ein- bzw. Ausgabepuffer und bestimmt somit die Art und Weise, wie die Zeichen auf den Puffer geschrieben oder aus dem Puffer gelesen werden. Zur Modifikation der Formatierungsweise kann der Programmierer auf einen Satz von Kontrollflags mittels der `setf()`-Funktionen oder Manipulatoren Einfluß nehmen.

Flag	Bedeutung	Maske	Bemerkung
`ios::width` `ios::fill`	Feldbreite Füllzeichen		nur folg. Ausg.

`ios::left`
`ios::right`
`ios::internal`
	Linksbündig
	Rechtsbündig
	Füllz. zw. Vorz. & Wert
		`ios::adjustfield`	

`ios::dec`
`ios::hex`
`ios::oct`
	Dezimal
	Hexadezimal
	Oktal
		`ios::basefield`	siehe
			auch
			`ios::showbase`

`ios::showbase`
`ios::showpos`
`ios::uppercase`
	Num. Basis ausgeben
	`+` mit ausgeben
	Großbuchstaben		

`ios::precision` `ios::showpoint`	Anzahl Gleit.-stellen Punkt stets ausgeben		Rundung!
`ios::scientific` `ios::fixed`	`1.234e5`-Format `1234.5`-Format	`ios::floatfield`	

Ausgabe-Breite und Füllzeichen

- die `width()`-Funktion legt die minimale Anzahl an Ausgabezeichen fest
- das an Leerstellen ausgegebene Füllzeichen kann mittels der `fill()`-Funktion gesetzt werden
- Beispiel (streams_4.cc):

```
#include <iostream.h>

int main() {
  int i = 123, j = -456;
  float f = 123.456, g = -1.23456;
  char s[] = "Text";

  cout << '|' << i << '|' << j << '|' << endl;
  cout << '|' << f << '|' << g << '|' << endl;

  cout.width(10);
  cout << '|' << i << '|' << j << '|' << endl;
  cout.width(10); cout.fill('X');
  cout << '|' << f << '|' << endl;
  cout.width(10);
  cout << '|' << s << '|' << g << '|' << endl;
  cout.width(10);
  cout << '|' << g << '|' << s << '|' << endl;
}
```

Programmausgabe:

```
|123|-456|
```

```
|123.456|-1.23456|
|       123|-456|
|XXX123.456|
|XXXXXXText|-1.23456|
|XX-1.23456|Text|
```

Ausrichtung

- `setf()`-Aufrufe setzen Flags innerhalb einer durch eine Maske bestimmte Flag-Gruppe
- die Ausrichtung wird durch die Flags der Gruppe `ios::adjustfield` gesetzt
- Beispiel (streams_5.cc):

```
#include <iostream.h>

int main() {
  int i = 123, j = -456;
  float f = 123.456, g = -1.23456;
  char s[] = "Text";

  cout << '|' << i << '|' << j << '|' << endl;
  cout << '|' << f << '|' << g << '|' << endl;

  cout.width(10); // ios::right ist Standard
  cout << '|' << i << '|' << j << '|' << endl;
  cout.width(10); cout.fill('X');
  cout.setf(ios::left, ios::adjustfield);
  cout << '|' << f << '|' << endl;
  cout.width(10);
  cout << '|' << s << '|' << g << '|' << endl;
  cout.width(10);
  cout.setf(ios::internal, ios::adjustfield);
  cout << '|' << g << '|' << s << '|' << endl; }
```

Programmausgabe:

```
|123|-456|
|123.456|-1.23456|
|       123|-456|
|123.456XXX
|TextXXXXXX|-1.23456|
|-XX1.23456|Text|
```

Integer-Ausgabe

- die Basis der Integer-Interpretation wird durch die Flags in der Gruppe `ios::basefield` kontrolliert

- zusätzlich kann mittels `ios::showbase` und `ios::showpos` bestimmt werden, ob die Basis bei der Ausgabe angegeben bzw. ob auch bei positiven Zahlen ein '+' vorangestellt werden soll
- Beispiel (streams_6.cc):

```
#include <iostream.h>
int main() {
  int i = 123, j = -456;
  float g = -1.23456, h = -1; char s[] = "Text";

  cout << '|' << i << '|' << j << '|' << endl;
  cout.setf(ios::oct, ios::basefield);
  cout << '|' << i << '|' << j << '|' << endl;
  cout.setf(ios::hex, ios::basefield);
  cout << '|' << i << '|' << j << '|' << endl;
  cout.setf(ios::oct, ios::basefield); cout.setf(ios::showbase);
  cout << '|' << i << '|' << j << '|' << endl;
  cout.setf(ios::hex, ios::basefield);
  cout.setf(ios::uppercase); cout.setf(ios::showpos);
  cout << '|' << i << '|' << j << '|' << endl;
  cout.setf(ios::dec, ios::basefield); cout.width(10); cout.fill('X');
  cout << '|' << i << '|' << j << '|' << endl;
  cout.width(10); cout.setf(ios::internal, ios::adjustfield);
  cout << '|' << h << '|' << g << '|' << s << '|' << endl; }
```

  Programmausgabe:

```
|123|-456|
|173|37777777070|
|7b|fffffe38|
|0173|037777777070|
|0X7B|0XFFFFFE38|
|XXXXXX+123|-456|
|-XXXXXXXX1|-1.23456|Text|
```

Gleitkomma-Ausgabe

- das Format der Gleitkommaausgabe wird kontrolliert durch die Flags der Gruppe `ios::floatfield`
- die Anzahl der ausgegebenen Stellen (Standard: n=6) wird durch den Aufruf von `precision(n)` gesetzt
- Beispiel (streams_7.cc):

```
#include <iostream.h>

int main()
```

```
{
  float f = 123.456, g = -1.23456, h = -1;

  cout << '|' << f << '|' << g << '|' << h << '|' << endl;
  cout.setf(ios::scientific, ios::floatfield);
  cout << '|' << f << '|' << g << '|' << h << '|' << endl;
  cout.setf(ios::fixed, ios::floatfield);
  cout << '|' << f << '|' << g << '|' << h << '|' << endl;
  cout.setf(0, ios::floatfield); // zurueck auf Default
  cout.setf(ios::showpos);
  cout.setf(ios::showpoint); cout.setf(ios::uppercase);
  cout << '|' << f << '|' << g << '|' << h << '|' << endl;
  cout.setf(ios::scientific, ios::floatfield);
  cout << '|' << f << '|' << g << '|' << h << '|' << endl;
  cout.setf(ios::fixed, ios::floatfield);
  cout << '|' << f << '|' << g << '|' << h << '|' << endl;
  cout.setf(0, ios::floatfield); // zurueck auf Default
  cout.precision(3);
  cout << '|' << f << '|' << g << '|' << h << '|' << endl;
  cout.setf(ios::scientific, ios::floatfield);
  cout << '|' << f << '|' << g << '|' << h << '|' << endl;
  cout.setf(ios::fixed, ios::floatfield);
  cout << '|' << f << '|' << g << '|' << h << '|' << endl;
}
```

Programmausgabe:

```
|123.456|-1.23456|-1|
|1.234560e+02|-1.234560e+00|-1.000000e+00|
|123.456001|-1.234560|-1.000000|
|123.456|-1.23456|-1|
|1.234560E+02|-1.234560E+00|-1.000000E+00|
|123.456001|-1.234560|-1.000000|
|123|-1.23|-1|
|1.235E+02|-1.235E+00|-1.000E+00|
|123.456|-1.235|-1.000|
```

9.5.2 Manipulatoren

- Einstellung des Formatstatus über die `setf()`-Aufrufe ist unbequem; man wünscht sich eine streamorientiertere Notation, z.B.
  `cout << i << oct << i << hex << i << endl;`
- `oct`, `hex` und `endl` sind hierbei Manipulatoren für den Stream `cout`

Manipulatoren ohne Argumente

- einfache Manipulatoren sind Zeiger auf Funktionen mit folgenden Typdefinitionen:

```
typedef ios& (*__manip)(ios&);
typedef istream& (*__imanip)(istream&);
typedef ostream& (*__omanip)(ostream&);
```

- für diese Manipulator-Typen sind in den einzelnen Stream-Klassen folgende `operator>>`-Operatorfunktionen definiert:

```
ostream& ostream::operator<<(__manip func)
  {(*func)(*this); return *this;}
ostream& ostream::operator<<(__omanip func)
  { return (*func)(*this);}

istream& istream::operator>>(__manip func)
  {(*func)(*this); return *this;}
istream& istream::operator>>(__imanip func)
  { return (*func)(*this); }
```

- folgende Manipulatoren sind auf diese Art und Weise realisiert (iostream.h einbinden!):

```
extern istream& ws(istream& ins);     // whitesp. aus istream entf.
extern ostream& flush(ostream& outs); // Puffer eines ostream leeren
extern ostream& endl(ostream& outs);  // '\n' an ostream anhaengen
extern ostream& ends(ostream& outs);  // '\0' an ostream anhaengen

inline ios& dec(ios& i)
{ i.setf(ios::dec, ios::dec|ios::hex|ios::oct); return i; }
inline ios& hex(ios& i)
{ i.setf(ios::hex, ios::dec|ios::hex|ios::oct); return i; }
inline ios& oct(ios& i)
{ i.setf(ios::oct, ios::dec|ios::hex|ios::oct); return i; }
```

- Beispiel (streams_8.cc):

```
#include <iostream.h>
int main() {
  int i, j;
  cout << "Zwei Zahlen, getrennt durch Whitespaces, eingeben:" << endl;
  cin >> ws >> i >> ws >> j;

  cout << dec << '|' << i << '|' << j << '|' << endl;
  cout << oct << '|' << i << '|' << j << '|' << endl;
  cout << hex << '|' << i << '|' << j << '|' << endl;
}
```

Programmausgabe:

```
Zwei Zahlen, getrennt durch Whitespaces, eingeben:
8
```

```
16
|8|16|
|10|20|
|8|10|
```

Manipulatoren mit Argumenten

- Manipulationen mit Argumenten werden analog denen ohne Argumente indirekt über einen Funktionsaufruf realisiert. Die dazu gerufene Funktion nimmt jetzt neben der Referenz auf einen Stream zusätzlich noch das Manipulator-Argument, Beispiel:

```
ios& width(ios& ros, int n) {
  ros.width(n);
  return ros;
}
```

- da ein `operator<<` neben dem Stream nur noch ein Argument haben kann, muß die Manipulation mit einem Argument abgebildet werden auf die Streamausgabe eines temporären Objektes, welches oben beschriebene Funktion und das Manipulator-Argument speichert

- bei einer Manipulation wie z.B.
  `cout << setw(10);`
  könnte auf folgende Weise ein temporäres Objekt geschaffen werden:

```
ios& width(ios& ros, int n) {
  ros.width(n);
  return ros;
}
smanip<int> setw(int n) {
  return smanip<int>(&width, n);
}
```

  in diesem Beispiel ist das temporäre Objekt vom Typ `smanip<int>`, ein instanziiertes Template, wobei der Typparameter durch den Typ des Manipulatorargumentes ersetzt wurde

- das `smanip`-Template könnte folgendermassen definiert werden:

```
template <class TP> class smanip {
  ios& (*_f)(ios&, TP);
  TP _a;
public:
  smanip(ios& (*f)(ios&, TP), TP a) : _f(f), _a(a) {}

friend istream& operator>>(istream& i, const smanip<TP>& m);
friend ostream& operator<<(ostream& o, const smanip<TP>& m);
};
```

```
template<class TP>
inline istream& operator>>(istream& i, const smanip<TP>& m)
{ (*m._f)(i, m._a); return i; }

template<class TP>
inline ostream& operator<<(ostream& o, const smanip<TP>& m)
{ (*m._f)(o, m._a); return o;}
```

Objekte vom Typ `smanip<T>` speichern somit einen Zeiger auf eine Funktion, die die eigentliche Manipulation ausführt sowie das zugehörige Manipulationsargument vom Typ `T`. Die für den jeweiligen instanziierten `smanip`-Typ überladene `operator<<`-Funktion bringt dann die im temporären `smanip`-Objekt gespeicherte Funktion mit dem Manipulatorargument zur Ausführung.

- im Header <iomanip.h> werden solche Templates für die Manipulation von `istream`'s (`imanip`), von `ostream`'s (`omanip`) und allgemein für Objekte der Klasse `ios` (`smanip`) definiert
- folgende Manipulatoren mit einem Argument werden bereits in <iomanip.h> erzeugt:

```
smanip<int> setbase(int);
smanip<int> setfill(int);
smanip<int> setprecision(int);
smanip<int> setw(int);

smanip<ios::fmtflags> resetiosflags(ios::fmtflags);
smanip<ios::fmtflags> setiosflags(ios::fmtflags);
```

- Beispiel für eigendefinierte Manipulatoren mit zwei Argumenten (streams_9.cc):

```
#include <iostream.h>
// gleichzeitiges Setzen des Fuellzeichens und der Feldbreite

template <class TP1, class TP2> class mysmanip {
  ios& (*_f)(ios&, TP1, TP2);
  TP1 _a; TP2 _b;
public:
  mysmanip(ios& (*f)(ios&, TP1, TP2), TP1 a, TP2 b)
  : _f(f), _a(a), _b(b){}
friend ostream& operator<<(ostream& o, const mysmanip<TP1,TP2>& m);
};

template<class TP1, class TP2>
ostream& operator<<(ostream& o, const mysmanip<TP1, TP2>& m)
```

```
{ (*m._f)(o, m._a, m._b); return o;}

ios& filledwidth(ios& ros, char f, int w) {
  ros.fill(f); ros.width(w); return ros;
}

mysmanip<char,int> setfilledwidth(char f, int w) {
  return mysmanip<char,int>(&filledwidth, f, w);
}

int main() {
  int i = 123;
  cout << '|' << i << '|' << endl;
  cout << setfilledwidth('#', 10) << '|' << i << '|' << endl; }
```

Programmausgabe:

```
|123|
|#######123|
```

9.6 Streams und Files

Für Filestreams stehen die Klassen `ifstream` (input), `ofstream` (output) und `fstream` (in- und output; von `iostream` abgeleitet; eigene Positionen für Schreiben und Lesen) zur Verfügung.

9.6.1 Files öffnen

- Ein `ofstream` wird standardmässig zum Schreiben, ein `ifstream` zum Lesen geöffnet. Zusätzlich lassen sich die Modi mit den `ios::open_mode` weiter spezialisieren:

```
class ios {
public:
// ...
  enum open_mode {
    in = _IO_INPUT,            // Lesen
    out = _IO_OUTPUT,          // Schreiben
    ate = _IO_ATEND,           // ans Dateiende
    app = _IO_APPEND,          // anhaengen
    trunc = _IO_TRUNC,         // auf Laenge 0 setzen
    nocreate = _IO_NOCREATE,  // File muss bereits exist.
    noreplace = _IO_NOREPLACE,// File darf noch nicht ex.
    bin = _IOS_BIN };          // Binaermodus
// ...
};
```

- das Öffnen eines Files geschieht in der Regel beim Konstruieren des Filestreams unter Angabe des Filenamens und des optionalen Open-Modus, Beispiel: `ofstream fout(filename, ios::out|ios::bin)`
- konnte das File nicht mit dem angegebenen Modus geöffnet werden, wird im dennoch konstruierten Stream der Status entsprechend gesetzt, sodass dieser auf Gültigkeit überprüft werden kann, z.B. durch `if(!fout) throw` ...

9.6.2 Files schliessen

- das einem Stream zugeordnete und geöffnete File wird durch den Destruktor des Streams automatisch zum Ende der Lebensdauer des Streamobjektes geschlossen
- ein explizites Schliessen ist durch Aufruf der `close()`-Methode möglich

9.6.3 Zwei Beispiele

1. Kopieren von Files (streams_10.cc):

```
#include <fstream.h>

int main(int argc, char *argv[])
{
  if(argc != 3)
  {
    cerr << "Falsche Zahl an Argumenten!" << endl;
    return 3;
  }

  ifstream from(argv[1], ios::in | ios::bin);
  if(!from)
    cerr << "Konnte Quelle nicht oeffnen!" << endl;

  ofstream to(argv[2], ios::out | ios::bin);
  if(!to)
    cerr << "Konnte Ziel nicht oeffnen!" << endl;

  char c;
  while(from.get(c)) to.put(c);

  if(!from.eof() || !to)
    cerr << "Irgendein Fehler ist passiert!" << endl;
}
```

2. Sichern eines Objektes zur späteren Wiederverwendung (streams_11.cc):

```
#include <fstream.h>
```

```
class Complex
{
  double re, im;

public:
  Complex(double _re = 0, double _im = 0)
  : re(_re), im(_im){}

friend ostream& operator<<(ostream&, const Complex&);
friend istream& operator>>(istream&, Complex&);
};

ostream& operator<<(ostream& ros, const Complex& rc)
{
  if(&ros == &cout)
    return ros << '(' << rc.re << ',' << rc.im << ')';
  else
    return ros << rc.re << ' ' << rc.im;
}
istream& operator>>(istream& ris, Complex& rc)
{
  return ris >> rc.re >> rc.im;
}

int main()
{
    Complex z;

    ifstream in("complex.dat");
    if(!in)
    {
      cout << "complex.dat nicht vorhanden." << endl
           << "Neue komplexe Zahl eingeben: <Realteil><Leerz.><Imag.-Teil>"
           << endl;
      cin >> z;
      ofstream out("complex.dat");
      if(!out)
      {
        cerr << "Fehler beim Speichern!" << endl;
        return 3;
      }
      out << z;
      cout << "Zahl " << z << " in complex.dat gesichert." << endl;
    }
    else
    {
      cout << "complex.dat vorhanden. Lese Zahl ein." << endl;
```

```
      in >> z;
      cout << "gelesene Zahl: " << z << endl;
    }
}
```

Programmausgabe, erster Lauf:

```
lydia:/usr/projects/kba/kurs-c++/DOC/BEISPIELE > ./streams_11
complex.dat nicht vorhanden.
Neue komplexe Zahl eingeben: <Realteil><Leerz.><Imag.-Teil>
1.1 2.2
Zahl (1.1,2.2) in complex.dat gesichert.
```

Programmausgabe, zweiter Lauf:

```
lydia:/usr/projects/kba/kurs-c++/DOC/BEISPIELE > ./streams_11
complex.dat vorhanden. Lese Zahl ein.
gelesene Zahl: (1.1,2.2)
```