Programmieren mit C++
– Einführungskurs –

Karsten Baensch, Thomas Müller

3. Dezember 1997

Inhaltsverzeichnis

Start

- Ziel des Kurses „A“:
  - Verstehen der Grundbegriffe der ooP
  - Erlernen der (Syntax und) Semantik von C++ anhand einfacher Beispiele, Unterschiede zu C
  - Verstehen der grundlegenden Konzepte von C++ zur objektorientierten Programmierung
  - Anwenden der iostream-Bibliothek
- Ziel des Kurses „B“:
  - detaillierte Kenntnisse der oo Konzepte mit C++
  - Erarbeiten weiterführender Beispiele anhand praktischer Erfordernisse
  - Anwenden der STL
  - Heranführen an Methoden zur oo Modellierung, Phasen der objektorientierten Softwareentwicklung
- nicht im Vordergrund:
  - Umgang mit spezifischen Produkten: C++ Entwicklungssystemen (z.B. Borland C++, Visual C++), Bibliotheken etc.
- Durchführung:
  - Vortrag/Seminar
  - Praktikum
- Inhalt des Kurses „A“:
  - Begriffe und Methodik der ooP
  - Grundlegende Konzepte
  - Klassen als Abstrakte Datentypen
    - Klassenbegriff, Definition von Klassen
    - Konstruktion und Destruktion
    - Operatorfunktionen, Grundlagen
  - C++-Templates
    - Grundlagen
  - Vererbung und Aufbau von Klassenhierarchien
    - Einfachvererbung
    - virtuelle Ableitung
    - polymorphes Verhalten, abstrakte Konzepte
  - Fehlerbehandlung mit Exception Handling
    - Grundlagen
  - I/O mit Streams

- Inhalt des Kurses „B“:
  - Weiterführende Konzepte
  - C++-Klassen bis ins Detail
    - friends
    - Selbstreferenz
    - Operatorfunktionen
    - Konvertierungen
      - implizit und explizit
      - Konstruktoren
      - Konvertierungsoperatoren
    - Klassenvariablen und -funktionen
  - C++-Templates
    - Container und Iterationen, generische Algorithmen
  - Vererbung und Aufbau von Klassenhierarchien
    - Mehrfachvererbung
  - Run-Time-Type-Informations (RTTI)
  - C++ - Namespaces
  - Fehlerbehandlung mit Exception Handling, Vertiefung
  - I/O mit Streams
    - Manipulation von Streams
  - Aufbau und Verwendung der Standard Template Library (STL)
  - Phasen der oo Softwareentwicklung, oo Modellierung
    - Patterns
    - Unified Modeling Language (UML)

Hinweise zur Durchführung des Praktikums, Teil „A“

Das Praktikum besteht aus dem Durcharbeiten und Verstehen der mitgelieferten Beispiele und dem Lösen der gestellten Praktikumsaufgaben. Folgender Index soll Ihnen zur Übersicht über die Praktikumsaufgaben dienen:

Die Lösungen zu den gestellten Praktikumsaufgaben werden getrennt zu den übrigen Beispielprogrammen geliefert. Die Lösungen (eigene ODER Referenz) sollten unter den jeweils vorgegebenen Namen im Verzeichnis <***kursdir***>`/MODULE/BEISPIELE_A/PROTECTED` gespeichert werden, da sie im Verlaufe des Kurses an dieser Stelle ggf. von anderen Programmtexten referenziert werden.

Index der Beispielprogramme und Kurzbeschreibung:

Kapitel	File	Kurzbeschreibung
Grundlagen	HelloCoutGrA01.cc	Hello, World! (`cout`)
	HelloPrintfGrA02.cc	Hello, World! (`printf`)
	LebensdauerGrA03.cc	Lebensdauer von Variablen
	ZketteFeldIdxGrA04.cc	ASCII-Tabelle mit Feldindizierung
	ZketteZeigIdxGrA05.cc	ASCII-Tabelle mit Zeigerrechnung
	SwapOrNotGrA06.cc	Argumentübergabe
	ConstConvGrA07.cc	Konvertierungen und `const`
	FakultaetGrA08.cc	Berechnung der Fakultät
	LogOpGrA09.cc	logische Operatoren
	AbsBetragGrA10.cc	absoluter Betrag
	ExplCastingGrA11.cc	explizite Konvertierungen
	TypedefGrA12.cc	`typedef`'s
	SaveStringGrA13.cc	dynamischer Speicher
	FirstIdxGrA14.cc	Zeichen in String suchen, vorwärts
	LastIdxGrA15.cc	Zeichen in String suchen, rückwärts
	SwitchGrA16.cc	Redundante Whitespaces entfernen

Vererbung	PunktVeA01.h	Klasse `Punkt`
	Rechteck2VeA02.h	Klasse `Rechteck`,
	Rechteck2VeA03.cc	durch 2 Punkte def.
	mainRechteck2VeA04.cc	Klasse `Rechteck`, Verw.
	RechteckRundVeA05.h	Klasse `RechteckRund`, Def.
	RechteckRundVeA06.cc	Klasse `RechteckRund`, Impl.
	mainRechteckRundVeA07.cc	Klasse `RechteckRund`, Verw.
	PROTECTED/DreieckVeA08.h	Aufgabe 8:
	PROTECTED/DreieckVeA09.cc	Klasse `Dreieck`
	mainDreieckKreisVeA10.cc	Klassen `Dreieck`, `Kreis`, Verw.
	PROTECTED/KreisVeA11.h	Klasse `Kreis`, Def.
	PROTECTED/KreisVeA12.cc	Klasse `Kreis`, Impl.
	mainRechteck3VeA13.cc	Klassen `Rechteck`, `-Rund`, V.
	Rechteck3VeA14.h	Klasse `Rechteck`, virt. Meth.
	GeoObjVeA15.h	abstrakte Klasse `GeoObj`
	Rechteck4VeA16.h	Klasse `Rechteck`, abgeleitet
	PROTECTED/Dreieck2VeA17.h	Aufgabe 9:
	PROTECTED/Dreieck2VeA18.cc	Klasse `Dreieck`, abgeleitet
	PROTECTED/KreisVeA19.h	Klasse `Kreis`, abgeleitet, Def.
	PROTECTED/KreisVeA20.cc	Klasse `Kreis`, abgeleitet, Impl.
	ZGeoObjVeA21.h	Klasse `ZGeoObj`, abgel., Def.
	ZGeoObjVeA22.cc	Klasse `ZGeoObj`, abgel., Impl.
	mainZGeoObjVeA23.cc	Klasse `ZGeoObj`, Verw.
	Rechteck4VeA24.cc	Klasse `Rechteck`, Impl.
Exceptions	xmsgExA01.h	Klasse `xmsg`
	xallocExA02.h	Klasse `xalloc`
	mainExceptExA03.cc	`new` mit Except.
	PROTECTED/mainExceptExA04.cc	Aufgabe 11
Streams	Streams1StA01.cc	Streamstatus
	Streams2StA02.cc	Klasse `Complex`, `ostream`
	Streams3StA03.cc	Klasse `Complex`, `istream`
	Streams4StA04.cc	Breite und Füllzeichen
	Streams5StA05.cc	Ausrichtung
	Streams6StA06.cc	Integer
	Streams7StA07.cc	Gleitzahlen
	Streams10StA08.cc	File kopieren
	Streams11StA09.cc	persistente Objekte `Complex`
	PROTECTED/StackStA10.h	Aufgabe 12:
	PROTECTED/mainStackStA11.cc	persistenter `Stack`

Literatur

[1] zur Zusammenfassung und Standardisierung siehe auch:
`http://www.tu-chemnitz.de/home/kba/oop/C++/rahmen.html`

Einführende Literatur, Standardwerke:

[2] Bjarne Stroustrup: Die C++ Programmiersprache;
(The C++ Programming Language);
2. Auflage 1992, 4. überarbeiteter Nachdruck 1994, Addison-Wesley, 1992

[3] Bjarne Stroustrup, M. A. Ellis: The Annotated C++ Reference Manual;
Addison-Wesley, 1990

[4] Bjarne Stroustrup: Design und Entwicklung von C++;
(The design and evolution of C++);
Addison-Wesley, 1994

[5] P. J. Plauger: The Draft Standard C++ library;
Englewood Cliffs, Prentice Hall, 1995

Weiterführende Literatur:

[6] Scott Meyers: Effektiv C++ programmieren;
(Effective C++);
Bonn, Addison-Wesley, 1995

[7] Scott Meyers: More Effective C++;
Bonn, Addison-Wesley, 1995

[8] Bruce Eckel: C++ inside & out;
Berk: Osb. McGraw-Hill, 1993

[9] Bruce Eckel: In C++ denken;
(Thinking in C++);
Englewood Cliffs, NJ (Prentice Hall), 1995

[10] *das "LSD-Buch" (violett und horizonterweiternd) ;-) :*
James O. Coplien: Advanced C++ programming styles and idioms;
Reading: Addison-Wesley, 1992

Design Patterns:

[11] *das Buch der "Gang of Four" (GoF) ;-) :*
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Entwurfsmuster; Elemente wiederverwendbarer objektorientierter Software;
(Design Patterns; Elements of Reusable Object-Oriented Software);
Addison-Wesley, 1996

[12] *das "POSA-Buch" der "Gang of fiVe" (GoV) ;-) :*
Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: Pattern-Oriented Software Architecture, A system of patterns; Cichester, Wiley, 1996

Analyse & Modellierung:

[13] Grady Booch: Objektorientierte Analyse und Design;
Bonn, Addison-Wesley, 1994

[14] Robert C. Martin: Objektorientierte C++ Anwendungen: Die Booch-Methode;
(Designing Object-Oriented C++ Applications using the Booch Method);
Prentice-Hall, 1996

Projektmanagement:

[15] Adele Goldberg, Kenneth S. Rubin: Succeeding with Objects; Decision Frameworks for Project Management;
Addison-Wesley, 1995

[16] Grady Booch: Object solutions; Managing the Object-Oriented Project;
Addison-Wesley, 1996

WWW:

[17] *Literatur:* `http://www.cs.uidaho.edu/~frincke/misc/booklists/ood_books.html`
Standardisierung: `http://www.csci.csusb.edu/dick/c++std/`
Patterns: `htpp://www.enteract.com/~bradapp/docs/patterns-intro.html`

News:

[18] `comp.lang.c++`
`comp.std.c++`
`de.comp.lang.c++`

1 Einführung

1.1 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: ANSI (American National Standards Institute)-Komitee X3J16 einberufen (J: 'Programmiersprachen', J16: das 16. zugelassene J-Komitee);
  Namensgebung für X3 jetzt:
  NCITS (National Committee for Information Technology Standards);
  damit heißt das C++-Komitee jetzt: NCITS J16
- Standardisierungsarbeiten des ANSI und der anderen nationalen Institute sind Grundlage für ISO (International Standards Organization) -Standardisierungsbemühungen;
  das technische Komitee für C++ in der ISO ist SC22/WG21 (Steering Committee 22, Working Group 21)
- 12/1996: Beginn der 'public comment period' für ISO/IEC CD-2 (Committee Draft)
- 07/1997: Beginn der Auswertung der 'comments' zum CD-2
- 03/1998: CD-2 wird DIS (Draft International Standard)
- 08/1998: DIS wird IS (International Standard)

1.1.1 C++ - Compilerentwicklung

- zuerst als Precompiler (Translator) realisiert C++ $\rightarrow$ 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.
  $\Rightarrow$ Vorsicht bei der Anwendung, falls portabel programmiert werden soll!

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 Einordnung

- 'Computer' können nur solche (Rechen-)Aufgaben erfolgreich bearbeiten, die ihnen in einer geeigneten Weise dargestellt wurden. Ein Problem muß richtig aufgearbeitet, **formuliert** werden, um von einem 'Computer' verstanden zu werden. Viele Möglichkeiten der Darstellung bzw. Formulierung sind denkbar. Der Maschinencode ist die niedrigste Darstellungsform, derer man sich zur Programmierung eines fertigen Prozessors bedienen kann. Die Evolution der Allzweck-Programmiersprachen von niedriger zu höherer Abstraktion wird (neben dem temporären Effizienzverlust) zunehmend durch den Verlust an Flexibilität begrenzt. Bei weiterer Erhöhung des Abstraktionsniveaus muß die Allzweck-Eigenschaft aufgegeben werden; es entstehen Darstellungsformen, die an bestimmte Problemklassen angepaßt sind (datenbankorientierte, sog. 4GL's (auch objektorientierte); VRML für VR-Simulationen; POSES++ (Mehrfarben-Petrinetze) für zeitdiskrete Simulationen; etc.).

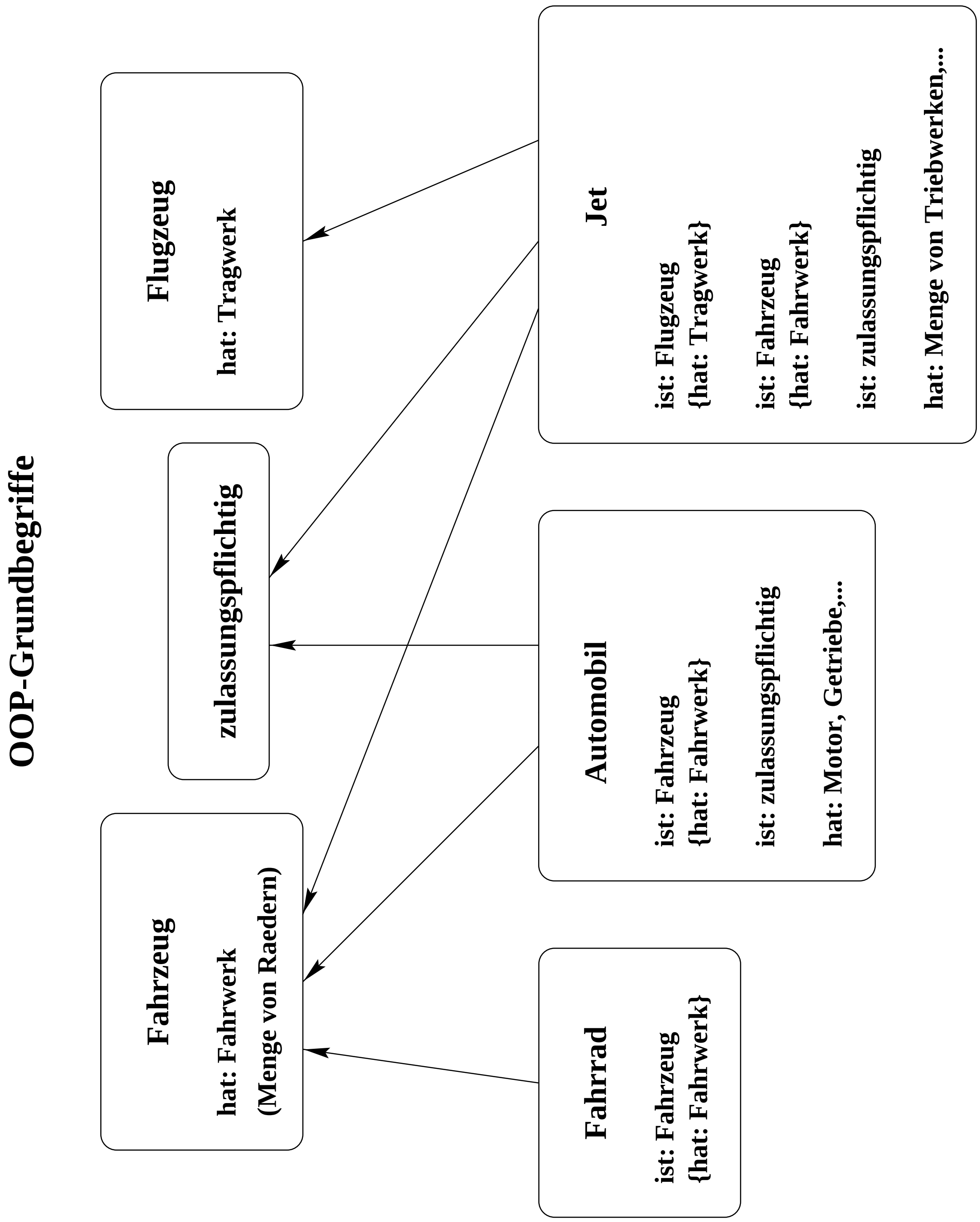
- Allen verwendbaren Darstellungsformen ist gemeinsam, daß es Werkzeuge (Compiler, Interpreter) geben muß, um ein so beschriebenes Problem durch Umsetzung in die 'natürliche Sprache' des Prozessors durch diesen bearbeitungsfähig zu machen.

- Wir behandeln im Kurs einen Vertreter aus dem Bereich der Allzweck-Programmiersprachen. In diesem Bereich (wie auch in den oben erwähnten anderen, eher problemangepaßten Bereichen) erfolgte in den letzten Jahren ein Wandel hin zu sog. Objektorientierung. Dies stellt einen erfolgversprechenden Versuch dar, durch bessere Strukturierung, Wieder- bzw. Weiterverwendung fertiger, getesteter und damit sicherer Komponenten und dadurch wiederum Erhöhung des Abstraktionsniveaus die Effizienz der Softwareentwicklung zu steigern. Es wird damit für den Programmierer nicht immer leichter! Der Aufwand zur Erlernung der komplexen und mächtigen Programmierumgebungen ist oft hoch.

- C++ vereint Möglichkeiten zur objektorientierten Programmierung mit der Flexibilität einer Allzweck-Programmiersprache und der Laufzeiteffizienz für die Systemprogrammierung und ist damit der derzeit wohl wichtigste Vertreter dieser Kategorie.

1.4 Was ist ooP?

(Begriffe und Methodik der ooP)



- Die Methodik der ooP erlaubt es, Begriffe (Ideen, Kategorien) aus der realen Welt direkt in die Software zu übernehmen, indem
  - die mit den Begriffen assoziierten, für das Softwareprojekt wesentlichen Eigenschaften analysiert und gespeichert werden
  - die Beziehungen zwischen den aus diesen Begriffen (Ideen, Kategorien) hervorgehenden (greifbaren) Gegenständen modelliert und für den Programmlauf ausgenutzt werden.
- Die Methodik der ooP ist damit stark an der menschlichen Denk- und Vorstellungswelt orientiert.
- Gewöhnlich denkt man in Begriffen und arbeitet mit realen Instanzen dieser.
  (⇒ <mein Auto> ist eine konkrete Instanz des Begriffes [ein Auto]; <das Auto des Nachbarn> ist eine weitere Instanz ...)
- In den Begriffen sind bestimmte Eigenschaften und Fähigkeiten enthalten bzw. assoziiert.
  (⇒ [ein Auto] hat [eine Farbe], [eine Menge von Raedern] ([ein Fahrwerk]); [ein Auto] ist 'zulassungspflichtig'.)
- Alle Instanzen eines Begriffes erhalten dessen Eigenschaften, bzw. wiederum konkrete Instanzen dieser. Verschiedene Instanzen desselben Begriffes haben zwar diesselben Eigenschaften; können aber verschiedene Werte für die Eigenschaften annehmen (Freiheitsgrad bei der Instanziierung).
  (⇒ <mein Auto> hat die Farbe <grau>; hat eine Instanz von [Menge von Raedern] <4 x R13> und eine konkrete Zulassung als Resultat der Eigenschaft 'zulassungspflichtig'.)
- Der Umgang mit Instanzen erfolgt durch Inanspruchnahme bestimmter Fähigkeiten; dazu findet Kommunikation statt:
  ⇒ Nachricht senden!
  (⇒ <mein Auto>.Starten(); <mein Auto>.Kurswechsel(<Wohin>); <konkretes Flugzeug>.Kurswechsel(<Wohin>))
- Neben bzw. vor der erwarteten Reaktion auf eine Nachricht kann es (sollte es) interne Überprüfungen geben.
  (⇒ <konkretes Flugzeug>.Kurswechsel(<RechtwinkligAbwaerts>))
  (Aktion nur, wenn Argumente korrekt und sonstige Bedingungen erfüllt.)
- Kapselung, *Information Hiding:* Nutzer sieht nur die Schnittstelle; die Technologie im Inneren sollte verborgen bleiben. Ggf. kommuniziert die angesprochene Instanz mit weiteren.
  (⇒ <mein Auto>.Starten()⇒<mein Motor>.Starten())
- Die Reaktion auf eine Nachricht wird unterschiedlich sein, auch, wenn die Nachricht über eine identische „Schnittstelle“ aufgenommen wurde (Individualität).
  (⇒ <mein Auto>.Starten(); <konkretes Flugzeug>.Starten())

- Individualität aber auch: Versteht die angesprochene Instanz die Nachicht überhaupt?
  (⇒ <mein Auto>.FahrwerkEinfahren()
  ⇒ Diese Aktion ist in der Regel nicht möglich.)

- Welche Nachrichten versteht eine Instanz?
  ⇒ Doku lesen oder Erfahrungen nutzen.
  ⇒ „Schnittstelle" einer Instanz leitet sich ab aus der „Schnittstelle", wie sie durch die Kategorie (auch: Typ eine Instanz) definiert ist.
  (Dabei reicht es, wenn man die Doku des Autotyps (Kategorie) liest und kann dann jedes konkrete <Auto> dieses Typs (Instanz) bedienen (benutzen).)
  ⇒ Instanzen haben eine Schnittstelle, die ihre Benutzung ermöglicht.

- Wo hat eine Instanz seine Eigenschaften/Fähigkeiten her?
  ⇒ mehrere Möglichkeiten:

  1. Es besteht aus Teilinstanzen und kombiniert durch Zusammenwirken dieser deren Eigenschaften/Fähigkeiten zu neuen Eigenschaften/Fähigkeiten (<mein Auto> besteht aus etlichen Einzelteilen).
  2. Weiterentwicklung bestehender Kategorien (bewährte Teile/Eigenschaften erhalten, neue ergänzen)
     ([ein Fahrzeug]←[ein Auto])
     ⇒ **Spezialisierung!**
  3. Zusammenführen der Eigenschaften mehrerer Kategorien

- bei der Entwicklung neuer Kategorien (Typen):

  - vollständige Neuentwicklung
  - ausgehend von vorhandenen Kategorien, Modifizierungen durch:
    - ergänzen
    - ersetzen
    - weiter kombinieren

Zentrale Begriffe der ooP

ooP	herkömmliche Bedeutung/Erläuterung
Klasse	Begriff, Typ, Kategorie (z.B. [ein 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
formale Argumente (Liste)	Anzahl, Typ und Reihenfolge der zur Ausführung einer Aktion **benötigten** Informationen / Werte / Objekte
aktuelle Argumente	zur Ausführung einer Aktion **gelieferte** 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 Trabi] **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 Raedern>.“, und aus „[ein Trabi] **ist** [ein Auto].“ folgt: „[ein Trabi] **hat** <eine Menge von Raedern>.“

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
<mein Auto>.Starten();	<konkretes Flugzeug>.Starten()
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. (Ein Objekt muß mindestens vom Typ [ein Fahrzeug] sein, um die Nachricht .Fahren(<Wohin>) verstehen zu können; konkret kann es aber z.B. eine Instanz von [ein Auto] oder auch [ein Fahrrad] sein!)

Erläuterungen zu einigen Begriffen aus dem Softwaredesign

Begriff	Erläuterung
Architekturmuster (architectural pattern)	beschreibt Schema zur fundamentalen strukturellen Organisation eines Softwaresystemes; beschreibt die Dekomposition in Subsysteme, die Verantwortlichkeiten dieser und die Organisation der Beziehungen dazwischen Beispiele: *Interaktive Systeme:* Model-View-Controller *Adaptierbare Systeme:* Microkernel-Architektur
Entwurfsmuster (design pattern)	beschreibt oft wiederkehrende Struktur miteinander in Beziehung stehender Komponenten eines Softwaresystems, die ein allgemeines Design-Problem in einem bestimmten Kontext löst Beispiele: *zur Arbeitsteilung:* Master-Slave-Struktur *zur Zugriffskontrolle:* Proxy-Struktur *zur Kommunikation:* Client-Dispatcher-Server
Idiom	ist ein Muster auf dem Niveau der Programmiersprache und auch für diese bestimmte Programmiersprache spezifisch; löst bestimmte Probleme mit den Features der Programmiersprache Beispiel: *Management dynamischer Speicherobjekte; Sharing eines Objektes zwischen mehreren referenzierenden Instanzen:* Handle-Counted Body Idiom

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 → Abstrakte Datentypen (ADT's)
  Jedes Objekt ist Exemplar einer Klasse (→ 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
  ⇒ 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!

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";`

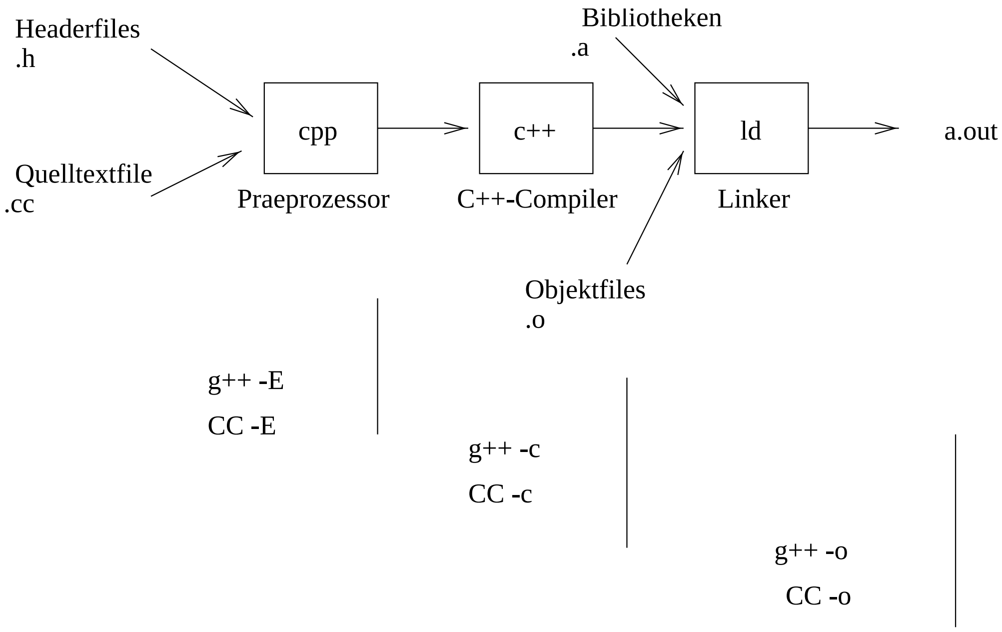
- Was macht das Pogramm?
- Ein C++ - Programm besteht aus Funktionen! (C-Historie)
- Eigentlich wird nur eine Funktion benötigt: `main`
  Dort geht die Ausführung los. (Es muß eine solche Funktion geben.)
- Offenbar findet eine Ausgabe statt: E/A gehört nicht zur Sprache C++, das Objekt `cout` ist Bestandteil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

Beispiel 2:

```
// File: HelloPrintfGrA02.cc

#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
  ⇒ 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,
  signature (?), 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`	$-2^{15} \ldots 2^{15}-1$	$-2^{31} \ldots 2^{31}$-1
`unsigned int`	$0 \ldots 2^{16}-1$	$0 \ldots 2^{32}-1$
`short`	$-2^{15} \ldots 2^{15}-1$	$-2^{15} \ldots 2^{15}$-1
`unsigned short`	$0 \ldots 2^{16}-1$	$0 \ldots 2^{16}-1$
`long`	$-2^{31} \ldots 2^{31}$-1	$-2^{31} \ldots 2^{31}$-1
`unsigned long`	$0 \ldots 2^{32}-1$	$0 \ldots 2^{32}-1$

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 (Sonderfall für `const`):
`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;      // Deklaration
  unsigned n = 1;         // Definition
  ```
  aber:
  ```
  extern const int c = 1; // Definition
  ```

- wenn nicht nur der Name einer Klasse deklariert wird
  Bsp.:

`class X;`	⇒ Typdeklaration, *Forward-Deklaration*
`class X { int x; public: x(int xx); void print(); };`	⇒ Typdefinition

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

- 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:

```
// File: HelloCoutGrA01.cc

#include <iostream.h>

int main()          // Beispiel 1: Erstes C++-Programm
{
     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.
- In C++ können auch Funktionsaufrufe Lvalues sein:
```
int& f();
f() = 2;
```

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***$>$)

    $<anweisung_1>$
  `else`
    $<anweisung_2>$

- Wahrheitswert:
  numerischer Wert von <ausdruck> verschieden von Null $\Rightarrow$ True
  numerischer Wert von <ausdruck> gleich Null $\Rightarrow$ 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

- Lebensdauer (Schlüsselwort `static`)

```
// File: LebensdauerGrA03.cc

#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 << " b = " << b << " 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
*/
```

- 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“.

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“

```
// File: ZketteFeldIdxGrA04.cc

#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
*/
```

- 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` $\equiv$ `&string[0]`
  `*string` $\equiv$ `string[0]`
  `*(string+1)` $\equiv$ `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

```
// File: ZketteZeigIdxGrA05.cc

#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?

```
// File: SwapOrNotGrA06.cc

#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;
}
```

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 i;
const int ci = 1;
const double PI = 3.141592653589;

int* pi = &i;
//int* pe = &ci;             // ? (n)
```

```
const int* pci = &ci;        //   (l)
const int* pce = &i;         // ? (j)

int *const cpi = &i;
//int *const cpe = &ci;      // ? (n)
const int *const cpci = &ci; //   (l)
const int *const cpce = &i;  // ? (j)
```

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;       // ? (n)
*pci = 2;          // ? (n)
pci = &i;          // ? (j)
*pi = pci;         // ? (n)
*cpi = 5;          // ? (j)
cpi = &ci;         // ? (n)
```

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
```

Praktikumsaufgabe Nr. 1:
Das Beispielprogramm `ConstConvGrA07.cc` *enthält einen schwerwiegenden Fehler im Zusammenhang mit den* `const`*-Deklarationen. Bitte ermitteln Sie die Reaktionen des Compilers und ggf. des Betriebssystems darauf und beschreiben Sie die Fehlerursache! Wenn Ihnen ein Post-Mortem Debugger zur Verfügung steht, dann nutzen Sie diesen, um die Absturzstelle zu lokalisieren!*

- 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};
  ```
  $\uparrow$
  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.

Praktikumsaufgabe Nr. 2:
Gegeben sei eine Struktur:

```
struct Align
{
  char k1;
  double k2;
  char k3;
  int k4;
};
```

Bitte ermitteln Sie die Größe der Struktur bei der Ausführung eines eigenen Testprogrammes, welches durch Ihren Compiler mit den in Ihrer IDE eingestellten Compileroptionen compiliert wurde und erklären Sie den Unterschied zur Summe der Größen der einzelnen Elemente!

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

```
// File: FakultaetGrA08.cc

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

```
// File: LogOpGrA09.cc

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:

  $\mathit{ausdruck}_1$ `?` $\mathit{ausdruck}_2$ `:` $\mathit{ausdruck}_3$

- Wert des Gesamtausdrucks ist vom Wahrheitswert des Gesamtausdrucks abhängig und entspricht dem Wert von $\mathit{ausdruck}_2$ im Fall True bzw. dem Wert von $\mathit{ausdruck}_3$ 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.:

```
// File: AbsBetragGrA10.cc

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:

  - cast-Operator

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

diese Variante funktioniert für alle Typen, auch abgeleitete:

```
// File: ExplCastingGrA11.cc

#include<iostream.h>

struct S1 {
  int a;
  float b;
};

struct S2 {
  int x;
  float y;
  double d;
};

int main()
{
  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.d = -0.1;  s2p->x--;

// nicht moeglich (implizit):
//s1p = s2p;
//s2p = s1p;

// jedoch explizit:
  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
```

```
    cout << "Was steht wohl hinter s1v im Speicher?" << endl;
    cout << s2p->d << endl,

    s2p->d = 88.88;     // SEGV
  }
```

– 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:

```
// File: TypedefGrA12.cc

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:

```
// File: SaveStringGrA13.cc

#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

- 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]

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

- 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)

```
// File: FirstIdxGrA14.cc

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:

```
// File: LastIdxGrA15.cc

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_ausdruck1 :
      anweisungs_liste1
```

...
`case` $\mathit{konst\_ausdruck}_i$ `:`
  $\mathit{anweisungs\_liste}_i$
`default:`
  $\mathit{anweisungs\_liste}_d$
`case` $\mathit{konst\_ausdruck}_{i+1}$ `:`
  $\mathit{anweisungs\_liste}_{i+1}$
  ...
`case` $\mathit{konst\_ausdruck}_n$ `:`
  $\mathit{anweisungs\_liste}_n$
}

- $\mathit{anweisungs\_liste}_i$ 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):

```
// File: SwitchGrA16.cc

#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
  ```

2.9 Zusammenfassung Grundlagen

C - Grundlagen

Sichtbarkeit eines Namens: Name ist verwendbar ohne Zugriffsqualifizierung	*Lebensdauer* einer Variablen: `static`: Programmlaufzeit automatisch: während Sichtbarkeit dynamisch: explizit formuliert
Deklaration - von Namen für Konstanten, Variablen, Funktionen, Typen: führt Namen in Sichtbarkeits- bereich ein	*Definition* - von Konstanten, Variablen: Vereinbarung von Speicher - von Funktionen, Typen: Festlegen der Datenstruktur und Funktionalität Definition ist auch Deklaration

OOP - Grundbegriffe

Klasse: Typdefinition; Begriff, "Layout eines Objektes"; definiert Datenstruktur und Funktionalität	*Objekt*: Instanz eines Typs; Menge strukturierter Daten und wohldefinierter Operationen darauf; nimmt Speicherplatz ein (irgendwo)
Ableitung: Erweiterung eines Typs um Daten, Funktionen; Spezialisierung der Kategorie; einfach oder mehrfach; *Is-A-Relation*	*Aggregation*: Kapselung mehrerer Objekte; homogen: Array von Objekten eines Typs innerhalb eines Objektes; heterogen: Komponenten verschiedenen Typs *Has-A-Relation*

C++ - Grundlagen

statisch: zur Compilezeit bekannt - *Bindung*: beim Compilieren klar, welche Methode beim Aufruf angesprungen wird - *Speicherbelegung*: Objekte über gesamte Laufzeit existend - *Typkonvertierung*: Weg der Konvertierung zur Compilezeit festgelegt	*dynamisch*: zur Laufzeit ermittelt konkret angesprungene Methode erst zur Laufzeit festgelegt ("spätes Binden") Objekt zur Laufzeit explizit konstruieren und destruieren Weg der Konvertierung und Resultat zur Laufzeit berechnet
implizit: vom Compiler entschieden - *Typkonvertierung*: selbständige Konvertierung durch den Compiler anhand eingebauter oder programmierter Regeln (z.B. Casting-Konstruktoren)	*explizit*: vom Programmierer angeordnet Konvertierung durch expliziten Casting-Operator
Initialisierung bei Konstruktion eines Objektes `int a = 3; float b(5.8);`	*Zuweisung* beliebig zur Laufzeit `a = 3; f = 5.8;`

Zeiger
- `T*` ist
"Zeiger auf Objekt des Typs `T`"
- Definition/Initialisierung:
`T oname;`
`T* pname = &oname;`
'`&oname`': "Adresse des Objektes `oname`"
- Verwendung/Zuweisung:
`*pname = wert;` (modifiziert `oname`)
'`*pname`': "Dereferenzierung"
"Inhalt der Adresse `pname`"
`pname = adresse;` (modifiziert `pname`)
	Referenzen
	- `T&` ist
	"Referenz auf Objekt des Typs `T`"
	
	`T oname;`
	`T& rname = oname;`
	
	`rname = wert;` (modifiziert `oname`)
	
	nicht möglich

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:

```
// File: String1KlA01.h

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:
    hier Methoden implementieren: Funktionsdefinitionen
    bei den Implementationen muß die Bezugnahme Klasse - Methode hergestellt werden:

```
// File: String1KlA02.cc

#include "String1KlA01.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:

```
// File: mainString1KlA03.cc

#include <iostream.h>
#include "String1KlA01.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 <u>keine</u>(!) 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`)

```
// File: String2KlA04.h

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)

```
// File: String2KlA05.cc

#include <iostream.h>
#include <string.h>
#include "String2KlA04.h"

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

void String::print() {
   cout << "[" << length() << "] ";
   if (s) cout << s; else cout << "(NULL)";
   cout << "\n";
}
```

- jetzt funktionieren folgende Dinge:

```
// File: mainString2KlA06.cc

#include <iostream.h>
#include "String2KlA04.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
```

Praktikumsaufgabe Nr. 3:

- *Testen Sie bitte obiges Programm!*
- *Entwickeln Sie bitte eine Klasse* `Rechteck`*!*
- *Entwickeln Sie bitte eine Klasse* `Stack` *zum Speichern für* `char`*'s!*
  Hinweis:
  Speichern Sie bitte die Klassendefinition `CharStack` *im File* `PROTECTED/CharStackKlA10.h`*, die Implementation der Methoden in* `PROTECTED/CharStackKlA11.cc` *und die* `main`*-Funktion in* `PROTECTED/mainCharStackKlA12.cc` *ab!*

Lösung zu `Rechteck`**:**

```
// File: Rechteck1KlA07.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: Rechteck1KlA08.cc

#include "Rechteck1KlA07.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: mainRechteck1KlA09.cc

#include <iostream.h>
#include "Rechteck1KlA07.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";
}
```

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: StringsUsageKlA13.h

#include  "String2KlA04.h"

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

```
// File: StringsUsageKlA14.cc

```

```
#include  <iostream.h>
#include  "StringsUsageKlA13.h"

String globalString("Hallo"); // Definition

void S1print()
{
    cout << "StringsUsageKlA14.cc: S1print()\n";
    globalString.print();
}
```

```
// File: mainString3KlA15

#include <iostream.h>
#include "StringsUsageKlA13.h"

void S2print()
{
    cout << "mainString3KlA15.cc: 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 StringsUsageKlA13.h macht den Bezeichner `globalString` in allen Files bekannt, die das File StringsUsageKlA13.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: mainString4KlA16.cc

#include <iostream.h>
#include  "String2KlA04.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:

```
// File: mainString5KlA17.cc

#include <iostream.h>
#include "String2KlA04.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

```
// File: mainString6KlA18.cc

#include <iostream.h>
```

```
#include "String2KlA04.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

```
// File: mainString7KlA21.cc

#include <iostream.h>
#include "String3KlA19.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 - 1;
    }
    ```

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

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

```
// File: NameKlA22.h

#include "String3KlA19.h"

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

```
// File: NameKlA23.cc

#include <iostream.h>
#include "NameKlA22.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

```
// File: mainNameKlA24.cc

#include "NameKlA22.h"

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

  nikolaus.print();
}
```

Praktikumsaufgabe Nr. 4:
Untersuchen Sie bitte im Beispiel `mainName` *die Aufrufreihenfolge der Konstruktionen und Destruktoren mit dem Trace-Modus Ihres Debuggers und erläutern Sie die Beobachtungen! Tauschen Sie bitte die Initialisierungen der Strings im Konstruktor von* `Name` *in der Reihenfolge aus und wiederholen Sie den Trace-Betrieb!*

- Felder von Objekten

```
// File: mainFeldKlA25.cc

#include "String3KlA19.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`

3.2 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, alle 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

Praktikumsaufgabe Nr. 5:
Definieren Sie bitte für die Klasse Name *einen* -> *Operator, der ausschließlich den Zugriff auf das Element* familienname *gestattet, d.h. folgende Verwendung:*

```
// File: mainElemZugriffKlA28.cc

#include "PROTECTED/ElemZugriffKlA26.h"

int main()
{
  Name* pnikolaus = new Name("Vaeterchen", "Frost");
// Unterschied:
  pnikolaus->print();    // eingebauter Operator ->
  (*pnikolaus)->print(); // String* Name::operator->()
  delete pnikolaus;
}
```

Hinweis:
Speichern Sie bitte die neue Definition der Klasse Name *im File* PROTECTED/ElemZugriffKlA26.h *und die Implementation des* -> *Operators in* PROTECTED/ElemZugriffKlA27.cc *ab! Linken Sie bitte zur Erzeugung des ausführbaren Programmes die Module* mainElemZugriffKlA28, PROTECTED/ElemZugriffKlA27, NameKlA23 *und* String3KlA20*!*

- Hinweis:
  - Weiteres Beispiel ist ein `Pointer`-Objekt zum Referenzieren dynamischer Objekte mit automatischer Löschung (auch als sog. Smart Pointers mit Reference Counting bezeichnet; auch verschiedene Idioms: *Handle Idiom*, *Reference Counting Idiom* oder *Counted Pointer Idiom*; auch auf Template-Basis); siehe Teil „B“.
- 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`

```
// File: String4KlA29.h

#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;
    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.

```
// File: String4KlA30.cc

#include <string.h>
#include "String4KlA29.h"

const 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;     // Wieso geht das?
   return is;
}
```

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

- Anwendung

```
// File: mainString8KlA31.cc

#include <iostream.h>
#include "String4KlA29.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
                        // (linkes Argument waere das eigene Objekt)
        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 konkrete 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 oder implizite Konvertierungen über programmierte Regeln (entsprechende Konstruktoren und Operatoren; auch über das Anlegen temporärer anonymer Objekte) 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 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 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:

```
// File: String5KlA32.h

#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):

```
// File: String5KlA33.cc

#include <string.h>
#include "String5KlA32.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

3.2.4 Aufgabe

Praktikumsaufgabe Nr. 6:
Erweitern Sie bitte die Klasse `String` *um Operatorfunktionen für die Operatoren* `+` *und* `+=`*! Beachten Sie dabei, daß folgende Dinge anschließend funktionieren:*

```
// File: mainString9KlA36.cc

#include <iostream.h>
#include "PROTECTED/String6KlA34.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;
}
```

Hinweise:

- *Benennen Sie bitte das zu entwickelnde Header-File mit* `String6KlA34.h` *und das Quelltextmodule mit den Implementationen der Operatoren mit* `String6KlA35.cc` *(um die Konsistenz des weiteren Praktikum-Verlaufes zu erhalten)!*
- *Zum Erzeugen des ausführbaren Files linken Sie bitte die Module* `mainString9KlA36`, `String3KlA20`, `String4KlA30`, `String5KlA33` *und* `PROTECTED/String6KlA35`*!*
- *Funktion der Standardbibliothek*

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

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

3.3 Besonderheiten

3.3.1 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

3.3.2 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 - 1;
}
```

- 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

3.3.3 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:

```
// File: mainAssocArrayKlA37.cc

#include <iostream.h>
#include "AssocArrayKlA38.h"
#include "PROTECTED/String6KlA34.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:

```
// File: AssocArrayKlA38.h

#include "PROTECTED/String6KlA34.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 `Paar`en
`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[](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

```
// File: AssocArrayKlA39.cc

#include <iostream.h>
#include "AssocArrayKlA38.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--) // rueckwaerts
      if (s1 == pp->wort)
```

```
            return pp->haeufigkeit;
      if (frei == anzahl) {                       // voll
         Paar* nvek = new Paar[anzahl*2];
         for (int i = 0; i < anzahl; i++)         // copy
            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.
Um einen *Map*-Container für alle denkbaren Kombinationen von Assoziationstypen bereitzustellen, müssen der (generische) Assoziationstyp selbst (Teil „A“) wie auch der Map-Container (Teil „B“) als C++-Templates realisiert werden.

4.0.4 Nachteile des bisherigen Designs

- wird dieselbe Containerfunktionalität (Abbildung einer Menge von Keys auf eine Menge von Values) auch für andere Assoziationstypen benötigt, dann muß die Klasse kopiert und der Assoziationstyp stets gegen den neuen ausgetauscht werden

- Container basiert auf der Implementation als Low-Level-Array (Einfügung in geordnete Menge der Keys in einer `Map`?!)

- Container und auf den Container anzuwendende Operationen (z.B. `print_all`) innerhalb derselben Klasse definiert und damit voneinander stark abhängig:
  - werden andere/zusätzliche Operationen benötigt, muß die Containerklasse geändert (ergänzt) werden
  - wird eine Operation auch für eine andere Menge (anderer Container mit anderer Funktionalität) benötigt, dann muß diese für den weiteren Container definiert (kopiert und angepaßt) werden.

4.1 Templates: Grundlagen

Zumindest der erste 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 (Achtung: Codegröße!)

4.2 Beispiel: Assoc-Template

- wir definieren eine allgemeingültige Assoziation als Template mit zwei Typparametern:

```
// File: AssocTeA01.h

#ifndef _ASSOC_H_
#define _ASSOC_H_

#include <iostream.h>

// Klassentemplate:
template<class K, class V>
struct Assoc  // setzt Defaultkonstruktoren
{             // fuer K und V voraus
  K Key;      // (sonst muessten Key und
  V Value;    // Value explizit initialisiert
};            // werden)

// Funktionstemplate:
template<class K, class V>
ostream& operator<<(ostream& ros, const 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:

```
// File: mainAssocTeA02.cc

#include <iostream.h>
#include "String7TeA03.h"
#include "AssocTeA01.h"

int main ()
{
  Assoc<String,int> a1;
  Assoc<char,int> a2;

  a1.Key = "Youa"; a1.Value = 888;
  a2.Key = 'A'; a2.Value = 'A';

  cout << "a1 = " << a1 << endl;
  cout << "a2 = " << a2 << endl;
}
```

- der Output des Programmes ist:

```
a1 = <Youa: 888>
a2 = <A: 65>
```

Praktikumsaufgabe Nr. 7:
Entwickeln Sie bitte ein Template für einen allgemeinen Stack! Verwenden Sie dazu bitte den Ansatz der Aufgabe 3 auf Seite 77! Benutzen Sie bitte das Template im Zusammenhang mit zwei verschiedenen Typparametern (`main`*-Routine)!*

5 Vererbung - Aufbau von Klassenhierarchien

5.1 Ableiten von Klassen

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

```
// File: PunktVeA01.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: Rechteck2VeA02.h

#ifndef _Rechteck_H_
#define _Rechteck_H_

#include "PunktVeA01.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: Rechteck2VeA03.cc

#include <math.h>
#include "Rechteck2VeA02.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:

```
// File: mainRechteck2VeA04.cc

#include <iostream.h>
#include "Rechteck2VeA02.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: RechteckRundVeA05.h

#ifndef _RechteckRund_H_
#define _RechteckRund_H_

#include "Rechteck2VeA02.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: RechteckRundVeA06.cc

#include "RechteckRundVeA05.h"

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

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

- Verwendung:

```
// File: mainRechteckRundVeA07.cc

```

```
#include <iostream.h>
#include "RechteckRundVeA05.h"

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

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

- allgemein:

```
class name1 : private|protected|public name2 {
      ...
};
```

  - 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

• Weitere geometrische Figuren:

Praktikumsaufgabe Nr. 8:

Entwickeln Sie bitte nach dem Muster der obigen `Rechteck`*-Klasse die Klassen* `Dreieck` *und* `Kreis`*! Beachten Sie bitte folgende Verwendung:*

```
// File: mainDreieckKreisVeA10.cc

#include <iostream.h>
#include "PROTECTED/DreieckVeA08.h"
#include "PROTECTED/KreisVeA11.h"

int main()
{
     Dreieck d(Punkt(1., 1.), Punkt(11.,  1.), Punkt(1., 11.));
     Kreis k(Punkt(11., 11.), 10.);

     cout << "Flaeche von d: " <<  d.flaeche() << "\n";
     cout << "Flaeche von k: " <<  k.flaeche() << "\n";
}
```

Hinweis:
Speichern Sie bitte die Definition der Klasse `Dreieck` *im File* `PROTECTED/DreieckVeA08.h` *und die Implementation in* `PROTECTED/DreieckVeA09.cc`*! Speichern Sie bitte die Definition der Klasse* `Kreis` *im File* `PROTECTED/KreisVeA11.h` *und die Implementation in* `PROTECTED/KreisVeA12.cc`*! Zum Erzeugen des ausführbaren Files linken Sie bitte die Module* `mainDreieckKreisVeA10`, `PROTECTED/DreieckVeA09` *und* `PROTECTED/KreisVeA12.cc`*!*

5.2 Objekte von abgeleiteten Klassen

- alle Möglichkeiten zur Instanziierung von Objekten
- Besonderheit: Zeiger

```
// File: mainRechteck3VeA13.cc

#include <iostream.h>
#include "RechteckRundVeA05.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 Polymorphes Verhalten durch virtuelle Funktionen

5.4.1 Definition und Verwendung

- weiter 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:

```
// File: Rechteck3VeA14.h

#ifndef _Rechteck_H_
#define _Rechteck_H_

#include "PunktVeA01.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

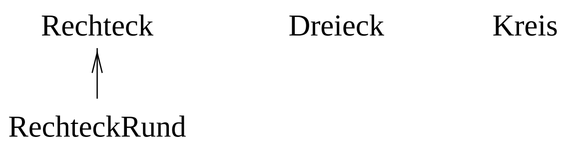
  `pr->flaeche();`

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

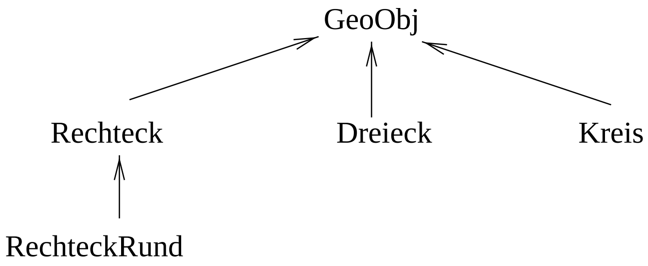
- dynamisches Binden bedeutet, daß zur Laufzeit entschieden wird, welche Methode an die Botschaft gebunden wird

5.4.2 Abstrakte Konzepte

- bisher folgende Klassen



- jede Klasse besitzt eine Methode `flaeche()`
- Idee: gemeinsame Oberklasse schaffen und Methode `flaeche()` herausziehen




```
// File: GeoObjVeA15.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

```
// File: Rechteck4VeA16.h

#ifndef _Rechteck_H_
#define _Rechteck_H_

#include "PunktVeA01.h"
#include "GeoObjVeA15.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);
   . . .
}
```

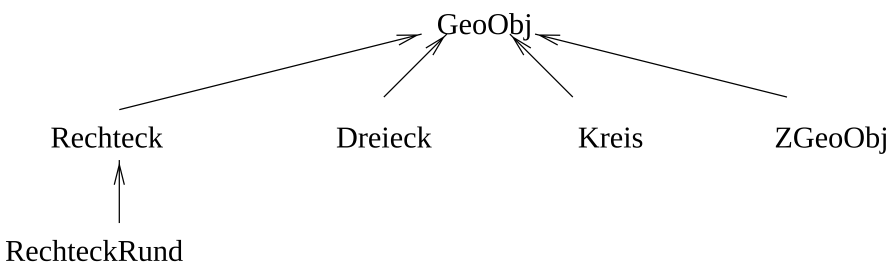
- auch für Referenzen

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

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

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

Praktikumsaufgabe Nr. 9:
Definieren und Implementieren Sie bitte zur Vorbereitung auf nachfolgendes Beispiel die noch fehlenden, von `GeoObj` *abgeleiteten Klassen* `Dreieck` *und* `Kreis`*! Benutzen Sie dabei die in der Aufgabe 8 auf Seite 107 entstandenen Quelltexte! Speichern Sie bitte die neue Klasse* `Dreieck` *in den Files* `PROTECTED/Dreieck2VeA17.h` *und* `PROTECTED/Dreieck2VeA18.cc`*, die neue Klasse* `Kreis` *in den Files* `PROTECTED/KreisVeA19.h` *und* `PROTECTED/KreisVeA20.cc` *ab!*

5.4.3 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:



- Definition der Klasse `ZGeoObj`:

```
// File: ZGeoObjVeA21.h

#include "GeoObjVeA15.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;
};
```

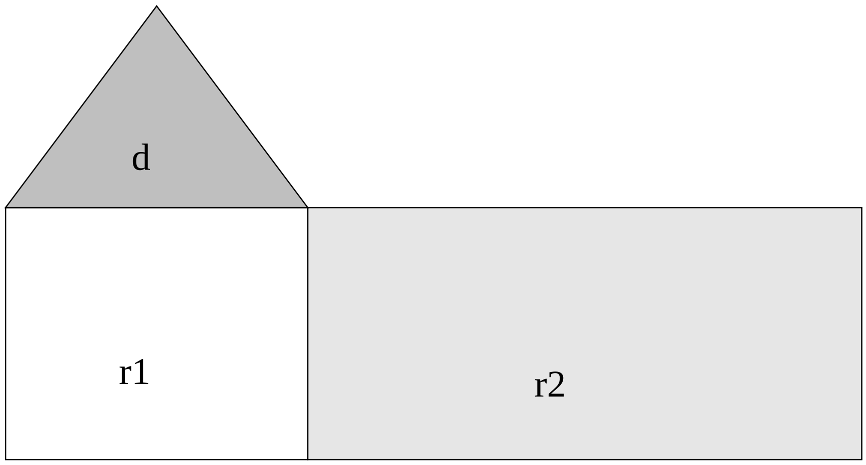
- Implementation `ZGeoObj`:

```
// File: ZGeoObjVeA22.cc

#include "ZGeoObjVeA21.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:

```
// File: mainZGeoObjVeA23.cc

#include <iostream.h>
#include "Rechteck4VeA16.h"
#include "PROTECTED/Dreieck2VeA17.h"
#include "ZGeoObjVeA21.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() << endl;
   cout << "Flaeche r2: " << r2.flaeche() << endl;
   cout << "Flaeche  d: " << d.flaeche() << endl;
   z1.set(r1);
   z1.set(d);
   cout << endl;
   cout << "Flaeche z1: " << z1.flaeche() << endl;
   z2.set(z1);
   z2.set(r2);
   cout << "Flaeche z2: " << z2.flaeche() << endl;
}

/* erwartete Ausgabe:
Flaeche r1: 30
Flaeche r2: 45
Flaeche  d: 12

Flaeche z1: 42
Flaeche z2: 87
*/
```



Praktikumsaufgabe Nr. 10:
Erweitern Sie bitte das Beispiel oben um Objekte Ihrer bei der Aufgabe 9 auf Seite 112 geschaffenen Klassen und weisen Sie auch deren polymorphes Verhalten in Bezug auf die virtuelle Methode `flaeche` *nach!*

6 Fehlerbehandlung

6.1 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?!)
   - 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

6.2 Ausnahmen und deren Behandlung in C++

6.2.1 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
```

6.2.2 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

6.2.3 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++-Typen instanziiert; können daher Objekte von eingebauten wie auch selbstdefinierten Typen (Klassen) sein
- 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: xmsgExA01.h

#ifndef __XMSG_H
#define __XMSG_H

#include <typeinfo>
#include "String7TeA03.h"

class xmsg {
  String desc;

```

```
public:
  xmsg(const String& what_arg): desc(what_arg) { }
  virtual ~xmsg(){}
  virtual String what() const { return desc; }
};

#endif
```

- Von der allgemeinen (Beispiel-)-Klasse `xmsg` wird nun `xalloc` abgeleitet, die auf Allocations-Fehler spezialisiert ist:

```
// File: xallocExA02.h

#ifndef __XALLOC_H
#define __XALLOC_H

#include <typeinfo>
#include "xmsgExA01.h"

class xalloc: public xmsg
{
private:
  size_t size;
  const String MakeString(const char *format, size_t _size);

public:
  xalloc(size_t _size)
  : xmsg(MakeString("Could not allocate %d bytes of memory!", _size)),
    size(_size){}

  size_t requested() const
  {
    return size;
  }
};

#endif
```

6.2.4 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.

6.2.5 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 (Bitte beachten: Ausgeworfen wird eine `xalloc`-Exception, aufgefangen wird jedoch 'nur' ein `xmsg`-Objekt!):

```
// File: mainExceptExA03.cc

#include <stdio.h>
#include <stdlib.h>
#include <new>
#include <typeinfo>

#include "xallocExA02.h"

const String xalloc::MakeString(const char *format, size_t size)
{
  static char str[0xff];

  sprintf(str, format, size); return str;
}

void *operator new[] (size_t size)
{
  void* p = malloc(size);

  if(!p)
    throw xalloc(size);

  return p;
```

```
}

void operator delete[] (void *p)
{
  free(p);
}

const size_t SegmentSize[2] =
{
  1024u * 1024u * 200u,
  1024u * 1024u * 10u
};

const int nSegments = 2;

int main()
{
  char *p;

  for(int i = 0; i < nSegments; i++)
  {
    try
    {
      p = new char[SegmentSize[i]];
    }
    catch(xmsg& ex)
    {
      cout << "Exception catched:\n\t" << ex.what() << endl;
      p = NULL;
    }
    printf("Allocation result for size=%u: p=%p\n", SegmentSize[i], p);
    delete[] p;
  }
}
```

6.2.6 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

6.2.7 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()
- die Funktionen `unexpected()` und `terminate()`, die Funktionen `set_unexpected()` und `set_terminate()` zum Setzen neuer Handlerfunktionen und die Typen der Zeiger auf derartige Handlerfunktionen sind folgendermaßen deklariert:

```
void unexpected (void);
void terminate (void);

typedef void (*unexpected_handler) ();
typedef void (*terminate_handler) ();

unexpected_handler set_unexpected (unexpected_handler);
terminate_handler set_terminate (terminate_handler);
```

Praktikumsaufgabe Nr. 11:
Exception-Objekte werden aus regulären C++-Typen instanziiert; können daher Objekte von eingebauten wie auch selbstdefinierten Typen (Klassen) sein. Entwickeln Sie bitte eine Funktion, in welcher ein initialisierter Integer (`int`*) als Exception ausgeworfen wird! Entwickeln Sie dann bitte eine* `main`*-Funktion, die die obige Funktion ruft und die Exception allerdings NICHT handelt. Verhindern Sie bitte die voreingestellte Beendigung des Programmes, indem Sie vor Aufruf der Funktion einen neuen Handler für* `terminate` *setzen! Im neuen Handler soll eine belanglose Bemerkung ausgegeben und danach in eine Endlosschleife eingetreten werden. Hinweis:*
Speichern Sie bitte das Programm im File `PROTECTED/mainExceptExA04.cc` *ab!*

7 I/O mit Streams

7.1 Grundlagen

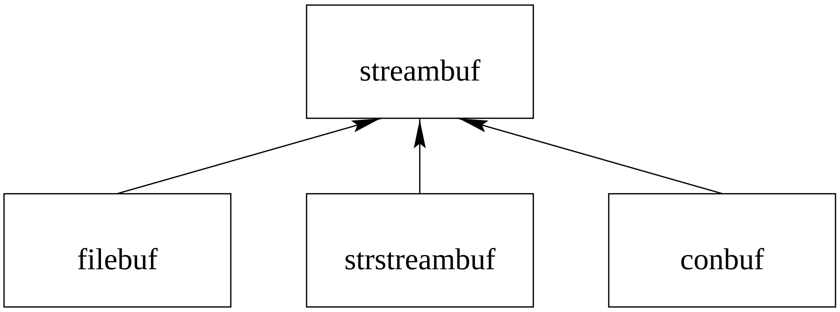
7.1.1 Was ist ein Stream?

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.

7.2 I/O-Stream-Klassen

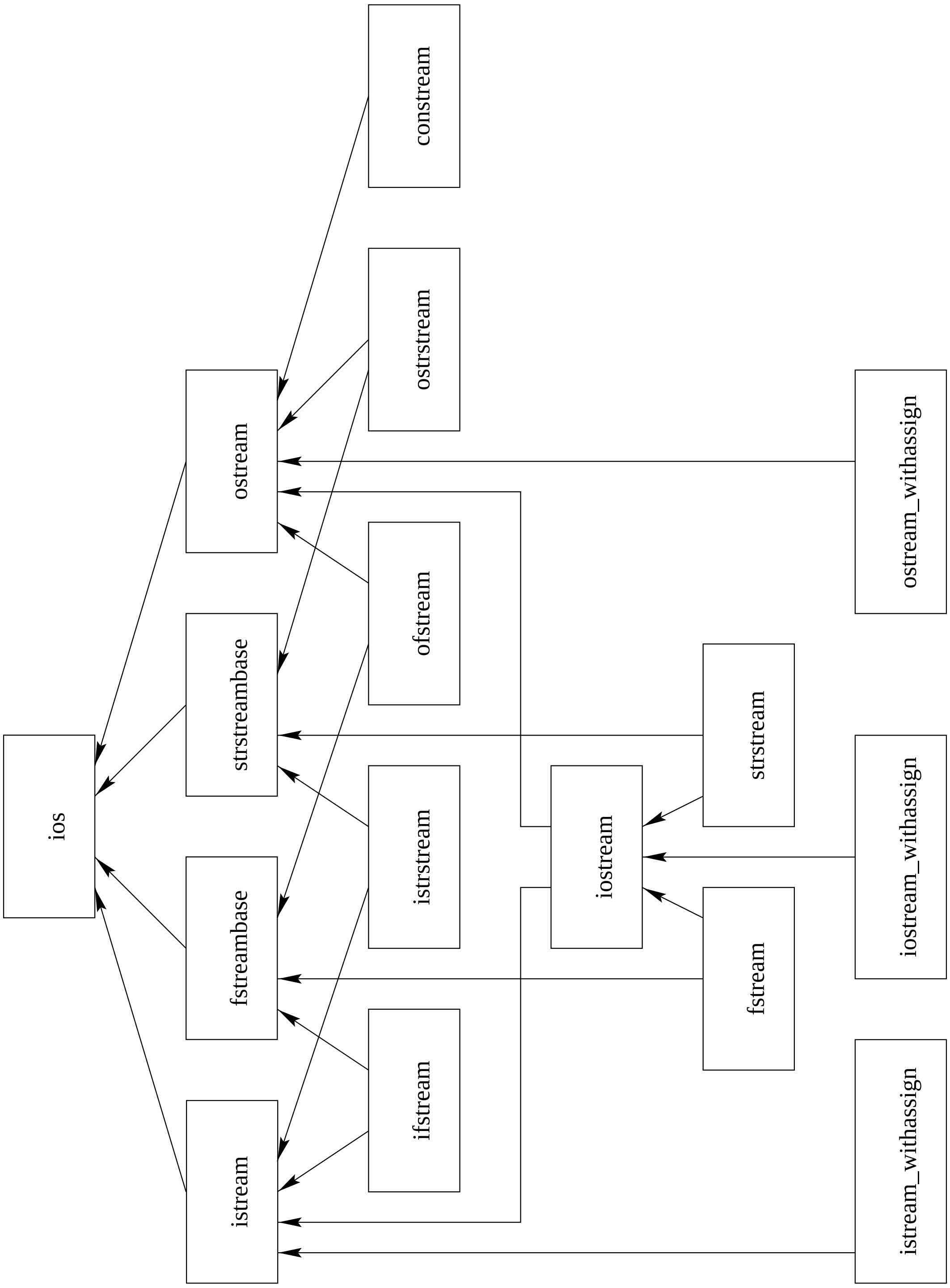
Die Stream-Bibliothek besteht aus zwei parallelen Klassenfamilien, der `streambuf`-Hierarchie und der `ios`-Hierarchie. Objekte von Klassen der `streambuf`-Hierarchie werden oft den Objekten der Klassen der `ios`-Hierarchie aggregiert; d.h. `ios`-Klassen haben oft Objekte von `streambuf`-Klassen als direkte oder indirekte (Pointer auf diese) Elemente.

7.2.1 `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

7.2.2 `ios`-Hierarchie



- Klasse `ios` und abgeleitete KLassen enthalten Zeiger auf ein `streambuf`-Objekt
- über die `streambuf`-Objekte führen `ios`-Objekte formatierte Ein- und Ausgabe durch

7.2.3 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
```

7.2.4 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:

```
// File: Streams1StA01.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);
    }
```

7.3 Stream-Ausgabe

7.3.1 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

7.3.2 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));`

7.3.3 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:

```
// File: Streams2StA02.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;
}
```

7.4 Stream-Eingabe

7.4.1 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

7.4.2 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

7.4.3 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:

```
// File: Streams3StA03.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;
}
```

7.4.4 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.

7.5 Formatierung

7.5.1 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:

```
// File: Streams4StA04.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:

```
// File: Streams5StA05.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:

```
// File: Streams6StA06.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:

```
// File: Streams7StA07.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|-1.00000|
|+1.234560E+02|-1.234560E+00|-1.000000E+00|
|+123.456001|-1.234560|-1.000000|
```

```
|+123.|-1.23|-1.00|
|+1.235E+02|-1.235E+00|-1.000E+00|
|+123.456|-1.235|-1.000|
```

7.6 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.

7.6.1 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 ...`

7.6.2 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

7.6.3 Zwei Beispiele

- Kopieren von benannten Files:

```
// File: Streams10StA08.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;
}
```

- Sichern eines Objektes zur späteren Wiederverwendung:

```
// File: Streams11StA09.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:

```
[kba@lydia BEISPIELE_A]$ ./Streams11StA09
complex.dat nicht vorhanden.
Neue komplexe Zahl eingeben: <Realteil><Leerz.><Imag.-Teil>
2 4
```

```
Zahl (2,4) in complex.dat gesichert.
```

- Programmausgabe, zweiter Lauf:

```
[kba@lydia BEISPIELE_A]$ ./Streams11StA09
complex.dat vorhanden. Lese Zahl ein.
gelesene Zahl: (2,4)
```

Praktikumsaufgabe Nr. 12:
Erweitern Sie das `Stack`*-Template von Beispiel 7 auf Seite 102 um einen* `operator<<` *und einen* `operator>>` *zum Schreiben auf einen* `ostream` *bzw. Lesen von einem* `istream`*! Wenden Sie dann bitte das Template als Stack für den Typ* `String` *(*`String7TeA03.h`*) in der Art und Weise des Beispieles (oben) an; d.h. im ersten Programmlauf wird nach Eingabe von Strings von der Tastatur das Stack-Objekt gesichert; wodurch im zweiten Programmlauf das Lesen der Daten aus einem File und Ausgabe der gespeicherten Strings (nach Stack-Art in umgekehrter Reihenfolge) möglich wird!*
Hinweis:
Speichern Sie bitte das neue Template im File
`PROTECTED/StackStA10.h` *und das Hauptprogramm im File*
`PROTECTED/mainStackStA11.cc` *ab!*
Folgende Verwendung:

- *Programmausgabe, erster Lauf:*

```
[kba@lydia BEISPIELE_A]$ ./mainStackStA11
stack.dat nicht vorhanden.
Neue Strings eingeben: <String><Enter><Ende: Ctrl-D>
Hallo Du !
Was
in
C++
nicht
alles
geht
!

Stack:
Hallo
Du
!
Was
in
C++
nicht
alles
geht
!
```

```
in stack.dat gesichert.
```

- *Programmausgabe, zweiter Lauf:*

```
[kba@lydia BEISPIELE_A]$ ./mainStackStA11
stack.dat vorhanden. Lese Stack ein.
gelesener Stack:
!
geht
alles
nicht
C++
in
Was
!
Du
Hallo
```