Programmieren mit C++
– Vertiefungskurs –

Karsten Baensch

16. Dezember 1997

Inhaltsverzeichnis

Start

Hinweise zur Durchführung des Praktikums, Teil „B“

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:

Aufgabe	**Seite**	**Kurzbeschreibung**
1	18	Kopieren und Zuweisen
2	33	Linking mit C-Funktionen
3	59	virtuelle Ableitung und virtuelle Methoden
4	73	Implementation des COMPOSITE-Patterns und RTTI
5	78	Implementation des COMPOSITE-Patterns ohne RTTI
6	85	Manipulation von Streams mit Methoden
7	107	Impl. der ZGeoObj::ReadFrom()-Methode
8	111	Einlesen komplexe ZGeoObj-Objekthierarchie
9	114	Verletzung der Precondition

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_B/PROTECT` 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**
	Name.h	Klasse Name, Definition
	Name.cc	Klasse Name, Implementation
	NamTitel.h	Klasse NamemitTitel, Definition
Grundlagen	NamTitel.cc	Klasse NamemitTitel, Implementation
	mName.cc	Klasse NamemitTitel, Verwendung
	ArrInit.h	Klassen zur Array-Initialisierung
	ArrInitT.h	Funktionstemplates zur Ausgabe
	ArrInit.cc	Tests zur Array-Initialisierung
	PROTECT/LinkCPP.h	Prakt. 2: Definition
	PROTECT/LinkCPP.cc	Prakt. 2: Implementation
	PROTECT/LinkC.c	Prakt. 2: C-Funktion
	mLinkC.cc	Prakt. 2: Test C-C++ Linking
Klassen	PoiToMem.h	Klassendef. Receiver und SpecialReceiver
	PoiToMem.cc	Test Pointer to Member
	DynRef.cc	Test Referenzen und dynamische Objekte
	PoToSMem.h	Receiver mit Klassenmethode
	PoToSMem.cc	Test Zeiger auf Klassenmethode
Namespaces	NamSpace.cc	Test Namespaces

Kapitel	**File**	**Kurzbeschreibung**
Vererbung	MultiVer.cc	Mehrfachvererbung, Ambiguity
	MultBase.h	Mehrfachvererb., mehrf. ent. Subobjekte
	MultBase.cc	Mehrfachvererb., mehrf. ent. Subobjekte
	AmbBase.cc	Mehrfachvererb., k. eindeut. Basisklasse
	VirtBase.h	Virt. Mehrfachvererbung, Klassen
	VirtBase.cc	Virt. Mehrfachvererbung, Test
	MultCall.h	Virt. Mehrfachver., ungew. mehrf. Aufruf
	MultCall.cc	Virt. Mehrfachver., ungew. mehrf. Aufruf
	SingCall.h	Virt. Mehrfachvererb., einm. Aufruf
	SingCall.cc	Virt. Mehrfachvererb., einm. Aufruf
	PROTECT/SiOpCall.h	Prakt. 3: polym. ostream-Ausg.
	SiOpCall.cc	Prakt. 3: polym. ostream-Ausg.
Konvertierung	KonvOp.cc	Test Konvertierungsoperator
	UCastImp.cc	implizite Upcasts
	DCastExp.cc	explizite Downcasts
	PunktV.h PunktV.cc GeoObjV.h GeoObjV.cc RchteckV.h RchteckV.cc RchtckRV.h RchtckRV.cc DreieckV.h DreieckV.cc KreisV.h KreisV.cc ZGeoObjV.h ZGeoObjV.cc mGeoV.cc	Prakt. 4: Vorlagen
	PunktR.h PunktR.cc PROTECT/GeoObjR.h PROTECT/GeoObjR.cc PROTECT/RchteckR.h PROTECT/RchteckR.cc PROTECT/RchtckRR.h PROTECT/RchtckRR.cc PROTECT/DreieckR.h PROTECT/DreieckR.cc PROTECT/KreisR.h PROTECT/KreisR.cc PROTECT/ZGeoObjR.h PROTECT/ZGeoObjR.cc mGeoR.cc	Prakt. 4: Implementation COMPOSITE Basis RTTI

Kapitel	**File**	**Kurzbeschreibung**
Konvertierung	PunktN.h PunktN.cc PROTECT/GeoObjN.h PROTECT/GeoObjN.cc PROTECT/RchteckN.h PROTECT/RchteckN.cc PROTECT/RchtckRN.h PROTECT/RchtckRN.cc PROTECT/DreieckN.h PROTECT/DreieckN.cc PROTECT/KreisN.h PROTECT/KreisN.cc PROTECT/ZGeoObjN.h PROTECT/ZGeoObjN.cc mGeoN.cc	Prakt. 5: Implementation COMPOSITE Basis eigene TypeStrings
Streams	Streams_B01.cc	Beispiel Manipulation ohne Args
	PROTECT/ManName.h	Prakt. 6: Manipulation mit Methoden
	NaivGeoO.h NaivGeoO.cc NaivRctk.h NaivRctk.cc NaivRtRn.h NaivRtRn.cc NaivDrik.h NaivDrik.cc NaivKris.h NaivKris.cc NaivZGeo.h NaivZGeo.cc	Streaming für GeoObjs, Interface Implementation der Operatoren hier die Probleme des naiven Entwurfes
	iNaiv.cc oNaiv.cc	1. Test zum naiven Entwurf
	iNaivZGe.cc oNaivZGe.cc	2. Test zum naiven Entwurf
ERZEUGER Pattern	AbstFcry.h AbstFcry.cc	ABSTRACT FACTORY Klassendefinition ABSTRACT FACTORY Implementation
	IOFactry.h IOFactry.cc	CONCRETE FACTORY Klassendefinition CONCRETE FACTORY Implementation
	GeoObj.h GeoObj.cc Rechteck.h Rechteck.cc RchtckRn.h RchtckRn.cc Dreieck.h Dreieck.cc Kreis.h Kreis.cc	

Kapitel	**File**	**Kurzbeschreibung**
	ZGeoObj.h	Prakt. 7:
	ZGeoObj.cc	Impl. Methode `ReadFrom()`, Vorlage
	PROTECT/ZGeoObj.cc	Benutzung ABSTRACT FACTORY
ERZEUGER	mFctTstC.cc	Test `IOFactory`: cin-Eingabe
Pattern	mFctTstF.cc	Test `IOFactory`: File-Eingabe
	mGeoFacD.cc	zwei Streams zur Objekt-Eingabe
	mGeoFacG.cc	Einlesen über einen gemeinsamen Stream
	precond.h	Precondition Klassendefinition
Precondition	precond.cc	Precondition Implementation
	mprecond.cc	Prakt. 9: Verletzung der Precondition

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`
STL: `http://www.sgi.com/Technology/STL/`
GNU Coding Standards:
`http://www.delorie.com/gnu/docs/standards/standards_toc.html`

News:

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

1 C++ Grundlagen

1.1 Zusammenfassung

Im Einführungskurs (Teil „A“) wurden die grundlegenden Begriffe und Konzepte der objektorientierten Programmierphilosophie erläutert. Es folgte eine Erarbeitung der für die Programmierung in C++ wesentlichen, aber zum großen Teil durchaus schon von der Programmiersprache „C“ her bekannten Grundlagen. Hierbei wurde vor allem auf die Kennzeichnung der Unterschiede C $\rightarrow$ C++ sowie auf die Gegenüberstellung und den Vergleich der logisch zusammengehörigen Aspekte der Syntax und Semantik (z.B. *Sichtbarkeit* $\Longleftrightarrow$ *Lebensdauer*) Wert gelegt. Folgende Bilder aus dem Kapitel „Grundlagen“ des Kursteiles „A“ dienten der kompakten Darstellung dieser Aspekte:

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**	***dynamisch***: **zur Laufzeit ermittelt**
- ***Bindung***:	
beim Compilieren klar, welche Methode beim Aufruf angesprungen wird	**konkret angesprungene Methode erst zur Laufzeit festgelegt ("spätes Binden")**
- ***Speicherbelegung***:	
Objekte über gesamte Laufzeit existend	**Objekt zur Laufzeit explizit konstruieren und destruieren**
- ***Typkonvertierung***:	
Weg der Konvertierung zur Compilezeit festgelegt	**Weg der Konvertierung und Resultat zur Laufzeit berechnet**
implizit: **vom Compiler entschieden**	***explizit***: **vom Programmierer angeordnet**
- ***Typkonvertierung***:	
selbständige Konvertierung durch den Compiler anhand eingebauter oder programmierter Regeln (z.B. Casting-Konstruktoren)	**Konvertierung durch expliziten Casting-Operator**
Initialisierung	***Zuweisung***
bei Konstruktion eines Objektes	**beliebig zur Laufzeit**
`int a = 3; float b(5.8);`	`a = 3; f = 5.8;`
Zeiger	***Referenzen***
- `T*` **ist**	**-** `T&` **ist**
"Zeiger auf Objekt des Typs `T`**"**	**"Referenz auf Objekt des Typs** `T`**"**
- Definition/Initialisierung:	
`T oname;`	`T oname;`
`T* pname = &oname;`	`T& rname = oname;`
'`&oname`**': "Adresse des Objektes** `oname`**"**	
- Verwendung/Zuweisung:	
`*pname = wert;` **(modifiziert** `oname`**)**	`rname = wert;` **(modifiziert** `oname`**)**
'`*pname`**': "Dereferenzierung"**	
"Inhalt der Adresse `pname`**"**	
`pname = adresse;` **(modifiziert** `pname`**)**	**nicht möglich**

Nach den „Grundlagen“ folgten im Kursteil „A“ mehrere Kapitel, in denen die wesentlichsten und gebräuchlichsten Elemente der Programmiersprache C++ behandelt wurden. So wurde der Klassenbegriff von C++ erörtert und die verschiedenen, durch und mit C++ nutzbaren Programmierstile (Konkrete Klassen, Generische Programmierung, Objektorientierte Klassenhierarchien) erklärt. Ein Kapitel zur Fehlerbehandlung in C++ (Exception Handling) sowie eines zur Erläuterung des Programm-I/O mit Streams komplettierten das Programm des Kursteiles „A“.
Im vorliegenden Kursteil „B“ werden nun die o.g. Kapitel vertieft und das weggelassene ergänzt. Weitere Kapitel zu den neueren Eigenschaften der Pro-

grammiersprache C++ folgen. Ein Kapitel zu C++-Templates und Aufbau und Verwendung der Standard-Template-Library (STL) dient zur Erweiterung der Fähigkeiten im *Generic Programming.* Eine Einführung in die recht moderne Philosophie der *Patterns und Idioms* wird auch Überleitung zu anspruchsvolleren Beispielen sein, die bestimmte C++-Idioms betreffen.

1.2 Initialisieren, Kopieren und Zuweisen

1.2.1 Kopier-Konstruktor und Zuweisungsoperator

Auf die besondere Bedeutung des Kopier-Konstruktors
`X::X(const X&)`
und des Zuweisungs-Operators
`const X& X::operator=(const X&)`
für die Verwendung von Objekten der Klasse X beispielsweise in folgenden Zusammenhängen

```
X f(X x)    // passing x by value, calling X::X(const X&)
{
  return x; // returning x by value, calling X::X(const X&)
}

X x;
X y(x);     // calling X::X(const X&)
X z = y;    // calling X::X(const X&)

x = z;      // calling const X& X::operator=(const X&)
```

wurde schon im Teil „A" hingewiesen. Es soll hier bezüglich der Entwicklung der Sprache C++ erwähnt werden, daß es erhebliche Unterschiede bei der Handhabung der Fälle „kein Kopierkonstruktor explizit definiert" und „kein Zuweisungs-Operator explizit definiert" durch die Compiler der verschiedenen Generationen gibt. Compiler früherer Generationen (Cfront 2) erzeugten als Standard-Operation für den nicht explizit definierten Kopier-Konstruktor bzw. Zuweisungsoperator eine bitweise Kopie, d.h. eine direkte Kopie des vom Quellobjekt belegten Speicherbereiches auf den Speicherbereich des Zielobjektes. Neuere Compiler (ANSI-C++) setzen für die nicht explizit definierten Kopier-Konstruktoren und Zuweisungsoperatoren standardmäßig eine *elementweise* Kopieroperation bzw. eine *elementweise* Zuweisung ein. Sind also für die Elemente eines Objektes `x` (aggregierte Objekte) der Klasse `X` oder auch für die Objekte der Basisklassen von `X` Kopier-Konstruktoren bzw. Zuweisungs-Operatoren definiert, dann werden diese innerhalb der Abarbeitung des vom Compiler standardmässig erzeugten Kopier-Konstruktors bzw. Zuweisungs-Operators korrekt aufgerufen. Ein Problem und damit die Notwendigkeit zur expliziten Definition entsteht erst, wenn Objekten der Klasse `X` andere Objekte als Elemente indirekt (über Pointer) aggregiert sind, die Klassendefinition von `X` also Pointer zu weiteren Objekten enthält. Die vom Compiler standardmässig eingesetzten elementweisen Kopieroperationen kopieren den Pointer als Wert. Es wird eine

shallow copy erzeugt; die Elemente werden korrekt kopiert, jedoch nicht die indirekt aggregierten Objekte. Wünscht der Programmierer dagegen eine *deep copy*, muß er die Kopier-Konstruktion und die Zuweisung explizit definieren.
Beispiel:
Aufbauend auf unsere String-Klasse (Files `String.h` und `String.cc`) aus dem Teil „A" definieren wir eine Klasse `Name` folgendermaßen:

```
// File: Name.h

#ifndef Name_h
#define Name_h 1

#include <String.h>

class Name {
private:
  String Vorname;
  String Familienname;

public:
  Name(const char*, const char*);
  virtual ~Name(){}

  virtual ostream& printOn(ostream&) const;

friend ostream& operator<<(ostream&, const Name&);
};

#endif
```

Die Methoden werden implementiert:

```
// File: Name.cc

#include <Name.h>

Name::Name(const char* _Vorname, const char* _Familienname)
: Vorname(_Vorname), Familienname(_Familienname)
{}

ostream& Name::printOn(ostream& ros) const
{
  return (ros << Vorname << " " << Familienname);
}

ostream& operator<<(ostream& ros, const Name& rName)
{
  return (rName.printOn(ros));
```

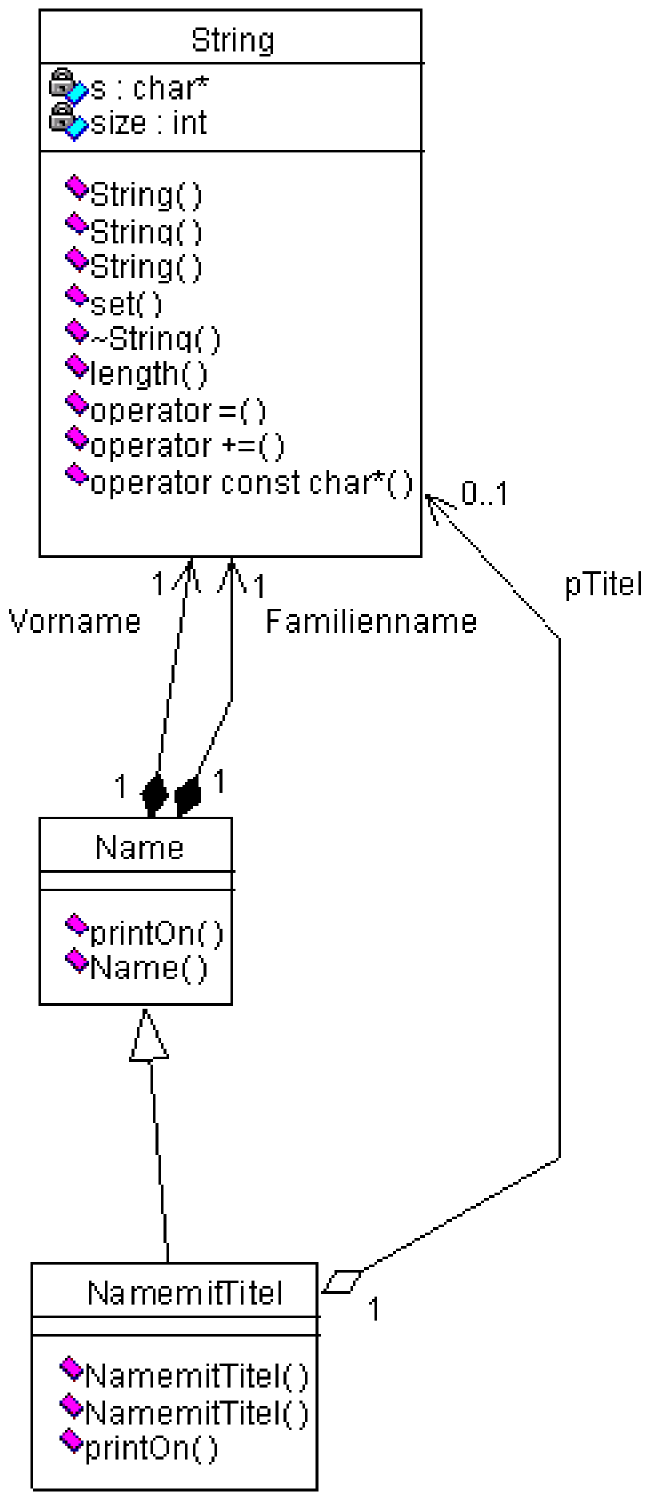
```
}
```

Beachten Sie bitte, daß die Methode `printOn` `virtual` definiert ist, also in polymorphen Zusammenhängen verwendet werden kann! Beachten Sie bitte weiterhin, daß die Klasse `Name` keinen explizit definierten Kopier-Konstruktor besitzt! Die daher vom Compiler eingesetzte elementweise (flache) Kopie (*shallow copy*) stellt in diesem Fall kein Problem dar, da die Strings durch den in der `String`-Klasse definierten Kopierkonstruktor korrekt kopiert werden können. Zur Ausgabe von Objekten der Klasse `Name` wurde ein Ausgabeoperator definiert, der die `printOn`-Methode nutzt. Da diese an der `public`-Schnittstelle sichtbar ist, ist es hier nicht notwendig, den `operator<<` als `friend` der Klasse `Name` zu deklarieren.

Durch Ableitung von `Name` definieren wir eine Klasse `NamemitTitel`:

```
// File: NamTitel.h

#include <Name.h>

class NamemitTitel: public Name
{
private:
  String *pTitel;

public:
  NamemitTitel(const char*, const Name&);
  NamemitTitel(const char* _Titel, const char*, const char*);

  virtual ~NamemitTitel();

  virtual ostream& printOn(ostream&) const;
};
```

Zwei verschiedene Konstruktoren dienen der unterschiedlichen Initialisierung eines Objektes von `NamemitTitel`. Einer der Konstruktoren wird zur Initialisierung durch ein `char`-Array (für die neu hinzugekommene Eigenschaft „Titel“) und ein bereits existierendes Objekt vom Typ `Name` verwendet; der zweite zur Initialisierung durch Übergabe dreier `char`-Arrays. Der erste erwähnte Konstruktor setzt auf den oben beschriebenen und als problemlos eingestuften impliziten Kopier-Konstruktor auf.


String
s : char*
size : int
String()
String()
String()
set()
~String()
length()
operator =()
operator +=()
operator const char*()
0..1
1
1
Vorname
Familienname
pTitel
1
1
Name
printOn()
Name()
NamemitTitel
NamemitTitel()
NamemitTitel()
printOn()
1

Die Methoden sind folgendermaßen implementiert:

```
// File: NamTitel.cc

#include <NamTitel.h>

NamemitTitel::NamemitTitel(const char* _Titel, const Name& _Name)
: Name(_Name)  // Name::Name(const Name&) not explicitely defined
{
  pTitel = new String(_Titel);
}

NamemitTitel::NamemitTitel(const char* _Titel,
      const char* _Vorname, const char* _Familienname)
: Name(_Vorname, _Familienname)
{
  pTitel = new String(_Titel);
}

NamemitTitel::~NamemitTitel()
{
  delete pTitel;
}

ostream& NamemitTitel::printOn(ostream& ros) const
{
  ros << (*pTitel) << " ";
  return Name::printOn(ros);
}
```

Beachten Sie bitte, daß wir zwar eine `printOn`-Methode definieren, allerdings keinen `operator<<`! Die `printOn`-Methode benutzt die gleichnamige Methode der Basisklasse.
Verwenden wollen wir die Klassen wie im folgenden File:

```
// File: mName.cc

#include <NamTitel.h>

int main()
{
  Name *pDerStinomann = new Name("Otto", "Normal");
  Name *pDerStinoHerr = new NamemitTitel("Herr", *pDerStinomann);
  NamemitTitel *pDieStinoFrau
                 = new NamemitTitel("Frau", "Lotte", "Normal");

  cout << (*pDerStinomann) << endl
       << (*pDerStinoHerr) << endl
```

```
        << (*pDieStinoFrau) << endl;

  Name *pAnOtherStino = new NamemitTitel(*pDieStinoFrau);

  delete pDerStinoHerr;
//  delete pDieStinoFrau; !!!

  cout << (*pAnOtherStino) << endl;

  return 0;
}
```

Die *shallow copy* des durch den Pointer `pDerStinomann` referenzierten Objektes zum Zwecke der Initialisierung des durch den Pointer `pDerStinoHerr` referenzierten Objektes funktioniert bei modernen Compilern tatsächlich wie erwartet; das Problem ist die Benutzung des implizit definierten Kopier-Konstruktors für die Klasse `NamemitTitel`. Durch ein evtl. Löschen des über `pDieStinoFrau` referenzierten Objektes wird auch der darin über `pTitel` referenzierte String im Destruktor gelöscht. Eine Ausgabe des neu entstandenen und über `pAnOtherStino` referenzierten `NamemitTitel`-Objektes bringt damit undefinierte Resultate.
Beachten Sie bitte, dass obwohl `pDerStinoHerr` vom Typ `Name*` ist, durch `cout << (*pDerStinoHerr)` das komplette darüber referenzierte `NamemitTitel`-Objekt ausgegeben wird!

Praktikumsaufgabe Nr. 1:
Testen Sie bitte obiges Programm! Stellen Sie fest, ob alle von Ihnen verwendeten Compiler auf den Ihnen wichtigen Plattformen tatsächlich standardmäßig elementweises (statt bitweises) Kopieren bzw. Zuweisen für nicht definierte Kopierkonstruktoren bzw. Zuweisungs-Operatoren einsetzen!

1.2.2 Array-Initialisierung

Das Thema „Array-Initialisierung“ wurde im Kursteil „A“ stark vereinfacht dargestellt. Es soll nun eine vollständige Erläuterung der in Abhängigkeit der Definition der Klassenelemente und Konstruktoren existierenden Möglichkeiten zur Initialisierung von Objekten in Arrays folgen.
Betrachten wir zunächst folgende Typdefinitionen:

```
// File: ArrInit.h

#include <iostream.h>

struct PublicS
{
  int i;
  char *s;
  float f;
};
```

```

class PublicC
{
public:
  int i;
  char *s;
  float f;
};

struct PrivateS
{
private:
  int i;
  char *s;
  float f;

friend ostream& printOn(ostream&, const PrivateS&, const char*);
friend ostream& printOn(ostream&, const PrivateS*, const char*, int);
};

class PrivateC
{
  int i;
  char *s;
  float f;

friend ostream& printOn(ostream&, const PrivateC&, const char*);
friend ostream& printOn(ostream&, const PrivateC*, const char*, int);
};

class NoDefaultC
{
public:
  int i;
  char *s;
  float f;

  NoDefaultC(int _i, char *_s, float _f)
  : i(_i), s(_s), f(_f){}
};

class DefaultC
{
public:
  int i;
  char *s;
  float f;
```

```

  DefaultC(int _i = 0, char *_s = "", float _f = 0.0)
  : i(_i), s(_s), f(_f){}
};
```

Damit sind verschiedene Typen definiert, deren Objekte im Speicher ein identisches Layout haben werden (gleiche Anzahl, Typen und Reihenfolge der Datenelemente). Die C++-Typen unterscheiden sich aber durch ihre Verwendbarkeit im Programmtext. Die Unterschiede werden durch ungleiche Element-Zugriffsspezifikation sowie durch das Vorhandensein verschiedener Arten von Konstruktoren verursacht.

- `PublicS` besitzt nur `public`-Datenelemente und keinen explizit definierten Defaultkonstruktor. Der Compiler definiert einen Defaultkonstruktor implizit.
- Für `PublicC` trifft das für `PublicS` genannte ebenfalls zu. `PublicC` und `PublicS` sind quasi identische Typdefinitionen.
- `PrivateS` hat nur `private`-Datenelemente, einen vom Compiler implizit definierten Defaultkonstruktor und zwei `friend`-Operatoren, die zur Ausgabe von Objekten dieses Typs sowie Arrays von Objekten dieses Typs dienen.
- `PrivateC` und `PrivateS` sind wiederum quasi identische Typdefinitionen.
- Der Typ `NoDefaultC` besitzt durch die Definition eines speziellen Konstruktors keinen implizit definierbaren Defaultkonstruktor.
- Der Typ `DefaultC` hat mit der speziellen Konstruktordefinition (Definition von Defaultargumenten für den Konstruktor) ebenfalls einen definierten Defaultkonstruktor.

Folgende zwei Funktionstemplates sollen die Ausgabe der Objekte bzw. Arrays von Objekten der o.g. Typen übernehmen:

```
// File: ArrInitT.h

#include <iostream.h>

template<class T>
ostream& printOn(ostream& ros, const T& rT, const char* tn)
{
  ros << "[" << tn << "] "
      << "i = " << rT.i
      << "; s = \"" << rT.s << "\"; f = " << rT.f << endl;
  return ros;
}

template<class T>
```

```
ostream& printOn(ostream& ros, const T* aT, int nElems, const char* tn)
{
  if(aT)
  {
    for(int i = 0; i < nElems; i++)
    {
      ros << "[" << tn << "][" << i << "] "
          << "i = " << aT[i].i
          << "; s = \"" << aT[i].s << "\"; f = " << aT[i].f << endl;
    }
  }
  return ros;
}
```

In der main-Funktion sollen die Möglichkeiten zur Initialisierung von Objekten o.g. Typen getestet werden:

```
// File: ArrInit.cc

#include <ArrInit.h>
#include <ArrInitT.h>

int main()
{
                      // only public elements:
  PublicS PuS;        //   implicit default Constructor
  PuS.s = "";         //   aggregate initialization:
  PublicC PuC = {11, "111", 11.11};
                      // private elements:
                      //   no aggregate initialization:
// PrivateC PrC = {22, "222", 22.22};
//  ‘PrC’ must be initialized by constructor, not by ‘{...}’

                      // constructor, but no implicit default:
// NoDefaultC NoDefC1;
//  no matching function for call to ‘NoDefaultC::NoDefaultC ()’
//  candidates are: NoDefaultC::NoDefaultC(const NoDefaultC &)
//                  NoDefaultC::NoDefaultC(int, char *, float)
                      //   no aggregate initialization:
// NoDefaultC NoDefC2 = {33, "333", 33.33};
//  ‘NoDefC2’ must be initialized by constructor, not by ‘{...}’
                      //   constructor initialization:
  NoDefaultC NoDefC3(33, "333", 33.33);

                      // constructor, explicit default too:
// DefaultC DefC1 = {44, "444", 44.44};
//  ‘DefC1’ must be initialized by constructor, not by ‘{...}’
  DefaultC DefC2;
```

```
  DefaultC DefC3(55, "555", 55.55);

  printOn(cout, PuS,     "PublicS");
  printOn(cout, PuC,     "PublicC");
  printOn(cout, NoDefC3, "NoDefaultC");
  printOn(cout, DefC2,   "DefaultC");
  printOn(cout, DefC3,   "DefaultC");
  cout << endl;

                      // only public elements:
  PublicS APuS[2];    //   implicit default Constructor
  APuS[0].s = ""; APuS[1].s = "";
                      //   aggregate initialization:
  PublicC APuC[] = {{66, "666", 66.66}, {77, "777", 77.77}};
                      // private elements:
                      //   no aggregate initialization:
// PrivateC APrC[] = {{88, "888", 88.88}, {99, "999", 99.99}};
//  subobject of type ‘PrivateC’ must be initialized by constructor,
//  not by ‘{88, "888", 0x00000000000000000000c00028f58f5cb1c24005}’
//  subobject of type ‘PrivateC’ must be initialized by constructor,
//  not by ‘{99, "999", 0x000000000000000000007800ae14e147c7fa4005}’

                      // constructor, but no implicit default:
// NoDefaultC ANoDefC1[2];
// no matching function for call to ‘NoDefaultC::NoDefaultC ()’
// candidates are: NoDefaultC::NoDefaultC(const NoDefaultC &)
//                 NoDefaultC::NoDefaultC(int, char *, float)
                      //   no aggregate initialization:
// NoDefaultC ANoDefC2[] = {{10, "100", 10.10}, {11, "110", 11.11}};
//  subobject of type ‘NoDefaultC’ must be initialized by constructor,
//  not by ‘{10, "100", 0x00000000000000000000980099999999a1994002}’
//  subobject of type ‘NoDefaultC’ must be initialized by constructor,
//  not by ‘{11, "110", 0x00000000000000000000c00028f58f5cb1c24002}’
                      //   constructor initialization:
  NoDefaultC ANoDefC3[] = {NoDefaultC(12, "120", 12.12),
                           NoDefaultC(13, "130", 13.13)};
                      // constructor, explicit default too:
// DefaultC ADefC1[] = {{14, "140", 14.14}, {15, "150", 15.15}};
//  subobject of type ‘DefaultC’ must be initialized by constructor,
//  not by ‘{14, "140", 0x000000000000000000004000d70a70a3e23d4002}’
//  subobject of type ‘DefaultC’ must be initialized by constructor,
//  not by ‘{15, "150", 0x00000000000000000000680066666666f2664002}’

  DefaultC ADefC2[2];
  DefaultC ADefC3[] = {DefaultC(16, "160", 16.16),
                       DefaultC(17, "170", 17.17)};

```

```
    printOn(cout, APuS,      2, "PublicS");
    printOn(cout, APuC,      2, "PublicC");
    printOn(cout, ANoDefC3,  2, "NoDefaultC");
    printOn(cout, ADefC2,    2, "DefaultC");
    printOn(cout, ADefC3,    2, "DefaultC");

    return 0;
};
```

- Mit Objekten der Typen `PublicS` und `PublicC` kann offensichtlich wie mit Strukturen der Programmiersprache „C“ umgegangen werden; Default-Konstruktion, Zuweisung an Datenelemente und *aggregate initialization* sind möglich.

- Nicht so mit Objekten von `PrivateC`. Wegen der vorhandenen `private`-Elemente ist der Aufruf eines Konstruktors notwendig, *aggregate initialization* daher verboten.

- Objekte vom Typ `NoDefaultC` können nicht durch Defaultkonstruktion erzeugt werden. Auch *aggregate initialization* ist nicht möglich.

- Objekte vom Typ `DefaultC` können wegen des speziellen Konstruktors durch Defaultkonstruktion sowie durch Konstruktion mit Argumenten erzeugt werden. Aber auch hier ist *aggregate initialization* nicht möglich.

Es folgen Tests zur Initialisierung von Arrays von Objekten der definierten C++-Typen.

- Arrays von Objekten von `PublicS` sowie auch von `PublicC` können durch Defaultkonstruktion und auch durch *aggregate initialization* erzeugt werden.

- *aggregate initialization* ist nicht möglich für Arrays von Objekten von `PrivateC`.

- Von Objekten von `NoDefaultC` können Arrays weder durch Default-Konstruktion noch durch gewöhnliche *aggregate initialization* erzeugt werden. Hier ist eine spezielle Syntax notwendig: expliziter Aufruf des Konstruktors für jedes im Array zu erzeugende Subobjekt.

- Zur Erzeugung von Arrays von Objekten vom Typ `DefaultC` kann Defaultkonstruktion wie auch die oben beschriebene spezielle Initialisierungssyntax verwendet werden.

Folgender Output wird vom Programm erzeugt:

```
[PublicS] i = 1074319255; s = ""; f = 0
[PublicC] i = 11; s = "111"; f = 11.11
[NoDefaultC] i = 33; s = "333"; f = 33.33
[DefaultC] i = 0; s = ""; f = 0
```

```
[DefaultC] i = 55; s = "555"; f = 55.55

[PublicS][0] i = -1073743128; s = ""; f = 2.26046
[PublicS][1] i = 93; s = ""; f = 2.12223
[PublicC][0] i = 66; s = "666"; f = 66.66
[PublicC][1] i = 77; s = "777"; f = 77.77
[NoDefaultC][0] i = 12; s = "120"; f = 12.12
[NoDefaultC][1] i = 13; s = "130"; f = 13.13
[DefaultC][0] i = 0; s = ""; f = 0
[DefaultC][1] i = 0; s = ""; f = 0
[DefaultC][0] i = 16; s = "160"; f = 16.16
[DefaultC][1] i = 17; s = "170"; f = 17.17
```

Zusammenfassung

- *aggregate initialization* ist nur dann möglich, wenn der Typ des zu initialisierenden Objektes ausschließlich `public`-Datenelemente enthält und einen vom Compiler implizit definierten Defaultkonstruktor (bzw. keinen explizit definierten). Diese Einschränkungen gelten für einzelne Objekte wie auch für zu initialisierende Arrays von Objekten.
- Default-Konstruktion von Objekten in Arrays ist nur bei implizit oder explizit definiertem Defaultkonstruktor möglich.
- Die Initialisierung von Arrays durch expliziten Aufruf des Konstruktors für jedes Subobjekt im Array ist immer (?) möglich.

2 Patterns und Idioms: wiederverwendbare Konzepte

Beim Erarbeiten dieses kurzen Kapitels und der patternorientierten Beispiele diente mir das Buch der *GoF* ([11]) als Grundlage.
Muster sind Problem- und Lösungsbeschreibungen zugleich. Ein Muster beschreibt ein oft wiederkehrendes Problem, den Kontext seines Auftretens und den Kern der Lösung dafür.
Ein Muster hat vier grundlegende Elemente:

1. Name eines Musters
   Der Name ist das Stichwort zur Benennung und Speicherung eines Musters sowie zur Erinnerung an ein solches und Kommunikation über ein solches.

2. Problembeschreibung
   Ein Muster enthält die Beschreibung des Problems, für welches es anzuwenden ist bzw. Lösungen bietet und den Kontext des Auftretens. Es beschreibt also, wann die vorgeschlagene Lösung in Betracht gezogen werden kann.

3. Lösungsabschnitt
   Im Lösungsabschnitt werden der Entwurf der vom Muster vorgeschlagenen Lösung, die Elemente des Entwurfs, deren Beziehungen, Abhängigkeiten, Zuständigkeiten und Interaktionen erörtert. Die vorgeschlagene Lösung ist, sofern es sich nicht um ein sprachspezifisches Idiom handelt, unabhängig von einer bestimmten Programmiersprache; kann aber Fragmente mit konkreter programmiersprachlicher Syntax als Beispielimplementation enthalten.

4. Konsequenzenabschnitt
   Hier werden die Konsequenzen der Anwendung des Musters, oft durch Auflistung der Vor- und Nachteile beschrieben. Eine Checkliste für diesen Punkt wäre:

   - Ressorcenverbrauch (Speicherplatz und Rechenzeit)
   - Implementierungsaspekte
   - Einfluß auf das gesamte Projekt
     - Ausführungsgeschwindigkeit
     - Flexibilität, Erweiterbarkeit, Portabilität
     - Robustheit
     - neue Probleme
     - darauf folgende Entwürfe

Muster können anhand der Granularität in folgende Kategorien unterteilt werden:

Kategorie	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-Observer *Adaptierbare Systeme:* - Microkernel-Architektur • Subsysteme können wiederum Implementationen von Mustern sein Beispiel: *Verhaltensmuster:* - Observer
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: *Strukturierungsmuster:* - Composite - Proxy-Struktur (*Zugriffskontrolle*) *Verhaltensmuster:* - Observer (*Beobachter*) - Iterator ...
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 *Automatische Löschung unreferenzierter dynamischer Speicherobjekte:* - Smart Pointer - Garbage Collection

Eine weitere Möglichkeit zur Strukturierung eines Musterkataloges ist die Gliederung nach Problemklassen:

- Erzeugermuster
  - Abstract Factory
  - Factory Method
  - Prototype
  - etc.
- Strukturmuster
  - Composit
  - Adapter
  - Proxy
  - etc.
- Verhaltensmuster
  - Observer
  - Interpreter
  - Iterator
  - State

2.1 Beispiel: Composite Pattern

ZWECK
Einheitliche Behandlung von Objekten (*Items*) und Kompositionen (*Compositions*) von Objekten als Komponenten wiederum in Kompositionen; *Whole-Part*-Hierarchien.

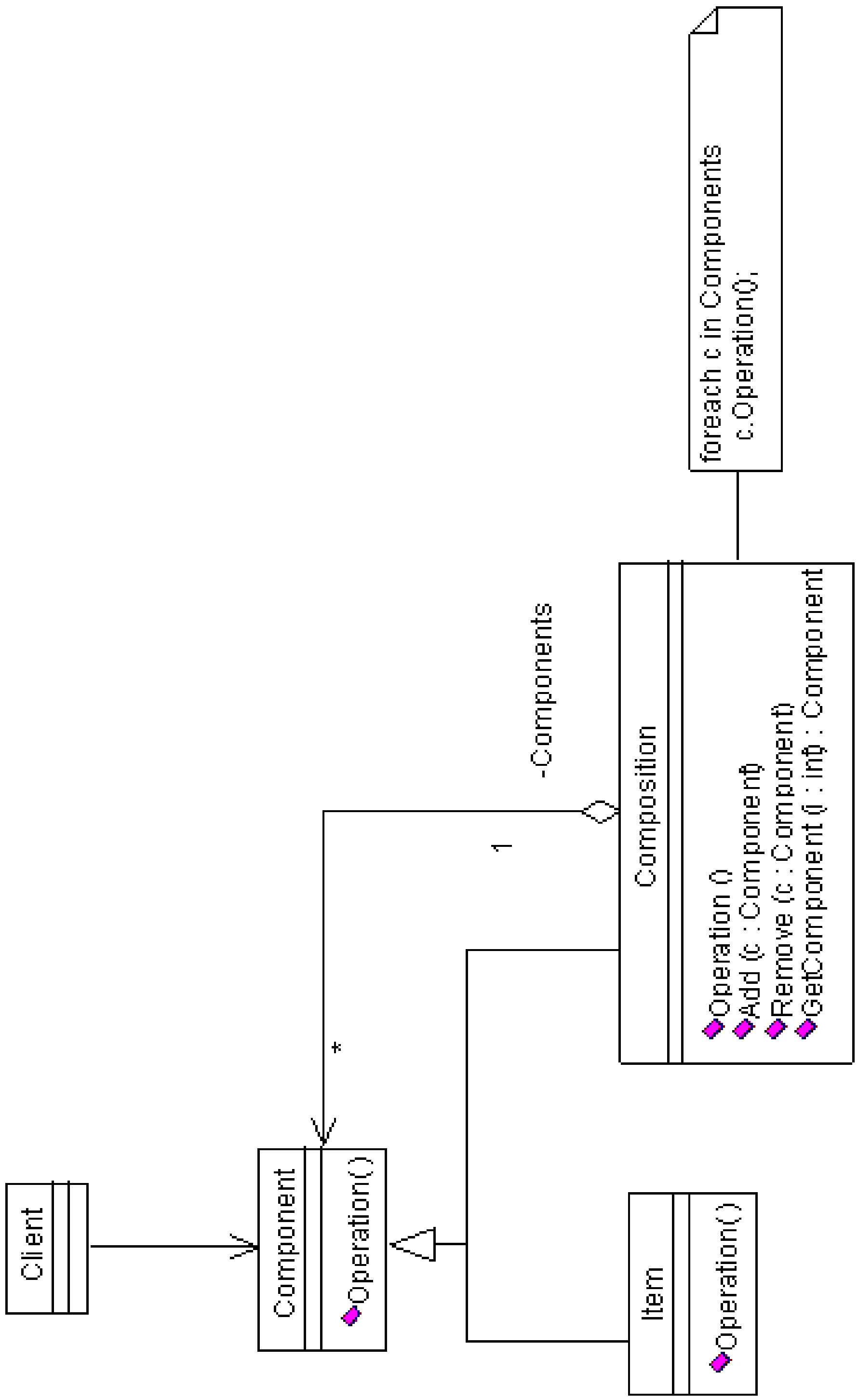
MOTIVATION
Beispiele:

- populäres Beispiel: Java Core API: package `java.awt` (Klassen `Component` und `Container`)
- Aufbau einer Grafik aus graphischen Objekten
- Aufbau eines Dialogfensters aus hierchisch strukturierten Dialogelementen

ANWENDBARKEIT

- Repräsentation von Whole-Part-Hierarchien von Objekten
- Verdeckung der Unterschiede von einfachen (elementaren) Objekten (*Items*) und zusammengesetzten Objekten (*Compositions*)
- einheitliche Behandlung von Objekten in der Struktur

Struktur


foreach c in Components
c.Operation();
-Components
1
*
Composition
Operation ()
Add (c : Component)
Remove (c : Component)
GetComponent (i : int) : Component
Client
Component
Operation()
Item
Operation()

Teilnehmer

- `Component`
  - Schnittstelle der Objekte in der Komposition (`Components`)
  - evtl. Implementation von Defaultverhalten dafür
  - (evtl.) Schnittstelle zum Verwalten der Kindobjekte (`Components`) und Defaultverhalten dafür
  - evtl. Schnittstelle zum Zugriff auf Elternobjekt (`Composition`)
- `Item`
  - elementares Objekt in der Komposition; keine Kindobjekte
- `Composition`
  - Verhalten von Komponenten, die selbst Komposition sein können (Kindobjekte haben können)
  - speichert `Components`
  - Implementation der Schnittstelle zum Verwalten der Kindobjekte (`Components`)
- `Client`
  - manipuliert Objekte in der Komposition durch die Schnittstelle von `Component`

Interaktionen

- `Clients` verwenden die Schnittstelle von `Component`, um Botschaften an die Objekte in der Hierarchie (`Components`: `Items` und `Compositions`) zu senden
- ist Empfänger ein `Item`, wird Anfrage direkt beantwortet (`Operation`)
- ist Empfänger eine `Composition`, wird Anfrage an die enthaltenen `Components` verteilt

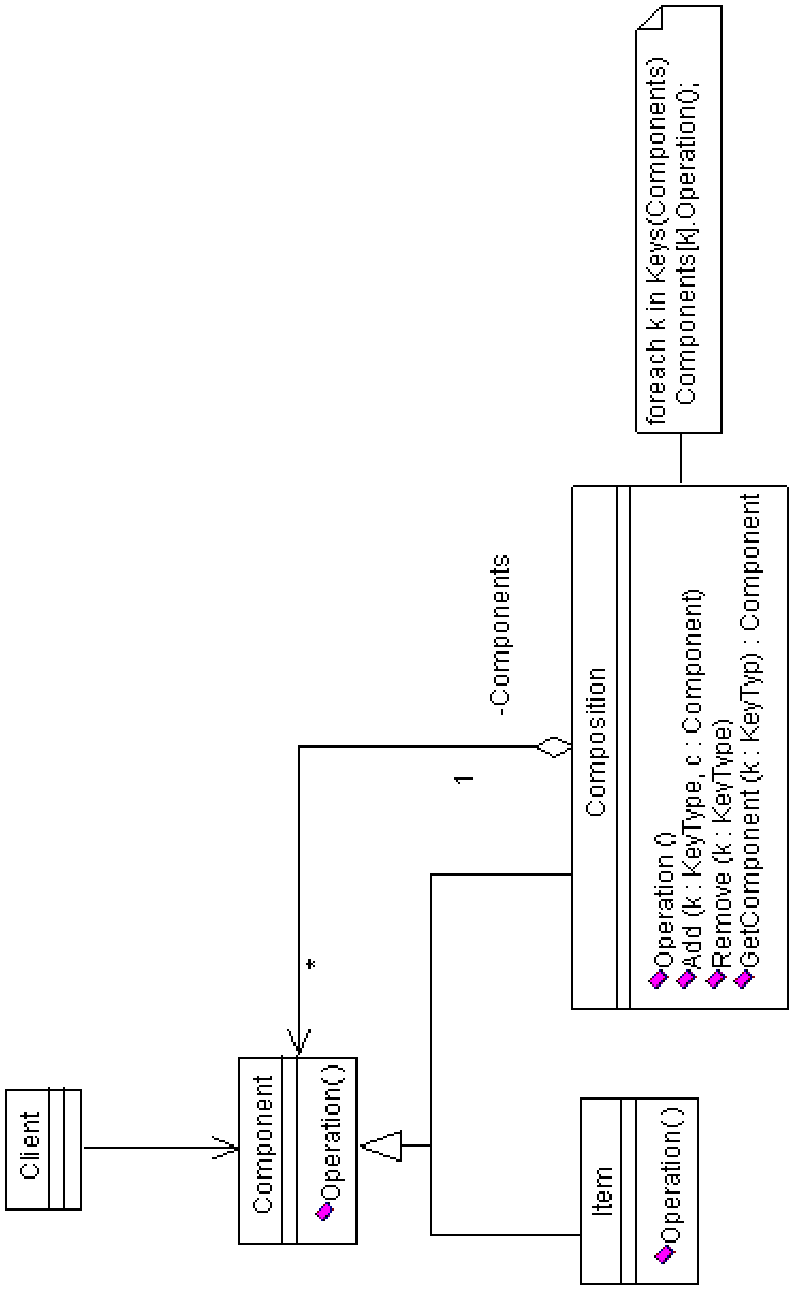
Konsequenzen

- Vereinfachung im `Client`
- Vereinfachung beim Definieren und Hinzufügen neuer `Clients`
- Gefahr der Über-Generalisierung: Keine Einschränkung bei den akzeptierten `Components`; evtl. dynamische Typüberprüfung (RTTI) notwendig
- Gefahr der unendlichen Rekursion

Implementation

- Speicherung der `Components`? Listen, Bäume, Arrays, Hash-Tabellen, Ordnung notwendig?

- evtl. Ordnung der Kindobjekte: Verwaltungsschnittstelle ändern:



- evtl. Referenzen auf das Elternobjekt
  Invariante: Alle Kindobjekte (`Components` einer Komposition (`Compositions`) müssen dieselbe Referenz auf das direkte Elternobjekt (`Compositions`) speichern

- Fragwürdigkeit der Definition der Schnittstelle zum Verwalten der Kindobjekte (`Components`) in `Components`; evtl. erst in `Composition` oder Test auf `Composition` (RTTI!)

- Verantwortlichkeit des `Composition`: Löschen der enthaltenen `Components` zum Ende der eigenen Laufzeit?

Siehe auch Praktikumsaufgabe 4 auf Seite 73.

3 C++-Klassen, Vertiefung

3.1 Pointer und Referenzen

3.1.1 Selbstreferenz

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

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

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

  X x1;
  X x2;

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

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

- für diese Bezugnahme existiert in jeder Methode ein (verborgener) Zeiger, der über das Schlüsselwort `this` angesprochen werden kann
- dieser Zeiger stellt die Bezugnahme zu dem Objekt her, für das die Methode gerufen wurde
  in jeder Methode einer Klasse `X` ist also ein Zeiger

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

  oder

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

implizit deklariert

- man könnte also auch schreiben:

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

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

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

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

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

Praktikumsaufgabe Nr. 2:
Diese Praktikumsaufgabe soll helfen, den Aufrufmechanismus von Methoden zu verstehen. Methoden werden beim Aufruf mit dem konkreten Objekt, für welches sie aufgerufen werden, durch Übergabe eines versteckten Zeigers darauf assoziiert. Die Übergabe des versteckten `this`*-Zeigers kann durch den Compiler derart optimiert werden, daß dieser nicht mehr auf dem Stack auszumachen ist. Wir wollen nun durch Implementieren einer (Pseudo-)methode in der Programmiersprache C und expliziter Übergabe der Adresse des Zielobjektes als erstes Argument an diese den Methodenaufruf simulieren. Die C-Funktion soll ein Datenelement des Objektes modifizieren. Definieren Sie bitte hierfür eine C++-Klasse* `LinkCPP` *mit einem Datenelement und zwei Methoden* `int get() const;` *und* `void set(int);`*! Speichern Sie bitte die Klassendefinition im File* `PROTECT/LinkCPP.h`*, die Implementation der Methoden im File* `PROTECT/LinkCPP.cc`*! Rufen Sie nun anstelle der Zuweisung an das* `private`*-Datenelement in* `LinkCPP::set` *eine C-Funktion* `LinkCPP_set` *auf und Übergeben Sie an diese zwei Argumente: 1. die Adresse des konkreten Objektes der Klasse* `LinkCPP` *und 2. das Argument zur Modifikation des* `private`*-Datenelementes. Die Zuweisung an das* `private`*-Datenelement soll innerhalb der C-Funktion erfolgen. Speichern Sie bitte die Implementation der C-Funktion in einem File* `PROTECT/LinkC.c`*! Hinweis:*

Zum Linken der C-Funktion muß diese im C++-Quelltext als externe Funktion mit C-Linkage gekennzeichnet werden. Benutzen Sie dazu bitte die Notation:

```
extern ''C'' {
Funktionsdeklaration
}
```

Folgende `main`*-Funktion soll die Verwendung verdeutlichen:*

```
// File: mLinkC.cc

#include <PROTECT/LinkCPP.h>

int main()
{
  LinkCPP lc;
  // Zuweisung an Element in C-Funktion:
  cout << "Old: " << lc.set(111) << endl;
  cout << "New: " << lc.get() << endl;

  // Zuweisung an Element in C-Funktion:
  cout << "Old: " << lc.set(222) << endl;
  cout << "New: " << lc.get() << endl;

  return 0;
}
```

3.1.2 Pointer zu Member

Soll eine Zeigervariable deklariert werden, die auf ein Datenelement oder auf eine Methode einer Klasse zeigt (allg. *Pointer to Member*), so muß neben dem vollständigen Typen des Members auch die zugehörige Klasse spezifiziert werden. Soll der Zeiger dereferenziert werden, so kann dies nur durch Angabe eines konkreten Objektes dieser Klasse erfolgen. Es muß also das Objekt angegeben werden, für welches die referenzierte Methode aufgerufen oder in welchem das referenzierte Element enthalten ist. Zeiger auf Datenelemente enthalten oft nur den Offset des Datenelementes innerhalb der Klasse in Bezug auf den `this`-Pointer. Siehe nachfolgendes Beispiel. Das folgende Programm soll verdeutlichen, daß der Methodenaufruf in C++ nicht nur als einfacher Funktionsaufruf, sondern eher als Nachricht an ein konketes Objekt zu begreifen ist. Wir definieren eine Klasse (in diesem Falle `struct`) `Receiver` und eine Klasse `SpecialReceiver`, die von ersterer abgeleitet ist. In `Receiver` sind einige Datenelemente enthalten sowie zwei einfache Methoden, die jeweils ein `int`-Argument erwarten. Die virtuelle Methode `Receive1` wird in `SpecialReceiver` neu definiert:

```
// File: PoiToMem.h

```

```
#include <iostream.h>

struct Receiver
{
  char charField1;
  double doubleField;
  char charField2;
  int intField;

  virtual void Receive1(int Arg);
  virtual void Receive2(int Arg);
};

struct SpecialReceiver: public Receiver
{
  virtual void Receive1(int Arg);
};
```

Die Implementation der `Receive`-Methoden ist recht einfach. Es soll nur eine Ausgabe auf `cout` gemacht werden, die Typname und Adresse des eigenen Objektes, den Namen der angesprungenen Methode sowie den Wert des übergebenen Argumentes zum Inhalt haben soll. Interessant dagegen ist die Funktion `send`. Sie dereferenziert den übergebenen Zeiger auf eine Methode von `Receiver` und benutzt dabei das als Referenz übergebene `Receiver`-Objekt. Die dadurch definierte Methode eines konkreten Objektes wird aufgerufen und das Integer-Argument übergeben:

```
// File: PoiToMem.cc

#include <PoiToMem.h>

void Receiver::Receive1(int Arg)
{
  cout << "Receiver Object at " << hex << unsigned(this) << dec
       << " received message \"Receive1\" and Arg " << Arg << endl;
}

void Receiver::Receive2(int Arg)
{
  cout << "Receiver Object at " << hex << unsigned(this) << dec
       << " received message \"Receive2\" and Arg " << Arg << endl;
}

void SpecialReceiver::Receive1(int Arg)
{
  cout << "SpecialReceiver Object at " << hex << unsigned(this) << dec
       << " received message \"Receive1\" and Arg " << Arg << endl;
```

```
}

void send (Receiver& r, void (Receiver::* p)(int), int Arg)
{
  (r.*p)(Arg);
}

int main()
{
  void (Receiver::* Message1)(int) = &(Receiver::Receive1);
  void (Receiver::* Message2)(int) = &(Receiver::Receive2);

  char Receiver::* pcharField1 = &(Receiver::charField1);
  double Receiver::* pdoubleField = &(Receiver::doubleField);
  char Receiver::* pcharField2 = &(Receiver::charField2);
  int Receiver::* pintField = &(Receiver::intField);

  Receiver *pr1 = new SpecialReceiver;
  Receiver *pr2 = new Receiver;

  send(*pr1, Message1, 111);
  send(*pr2, Message2, 222);

  cout << "Offsets in [Receiver]:" << endl
       << "Receiver::charField1 - " << unsigned(pcharField1) << endl
       << "Receiver::doubleField - " << unsigned(pdoubleField) << endl
       << "Receiver::charField2 - " << unsigned(pcharField2) << endl
       << "Receiver::intField - " << unsigned(pintField) << endl;

  cout << "Adresses in <r1>:" << endl << hex
       << "pr1->charField1 - " << unsigned(&(pr1->*pcharField1)) << endl
       << "pr1->doubleField - " << unsigned(&(pr1->*pdoubleField)) << endl
       << "pr1->charField2 - " << unsigned(&(pr1->*pcharField2)) << endl
       << "pr1->intField - " << unsigned(&(pr1->*pintField)) << endl;

  delete pr1; // if Receiver needs a destructor here,
  delete pr2; // it should be a virtual one.

  return 0;
}
```

In der `main()`-Funktion soll weiterhin deutlich werden, dass in Zeigern auf Datenelemente einer Klasse tatsächlich nur die Offsets zum `this`-Pointer gespeichert werden. Die realen Adressen der Elemente entstehen erst durch Angabe eines konkreten Objektes. Beachten Sie bitte in nachfolgendem Programmoutput, daß für das `SpecialReceiver`-Objekt die korrekte (redefinierte) Methode angesprungen wird:

```
SpecialReceiver Object at 804aec0 received message "Receive1" and Arg 111
Receiver Object at 804b488 received message "Receive2" and Arg 222
Offsets in [Receiver]:
Receiver::charField1 - 0
Receiver::doubleField - 4
Receiver::charField2 - 12
Receiver::intField - 16
Adresses in <r1>:
pr1->charField1 - 804aec0
pr1->doubleField - 804aec4
pr1->charField2 - 804aecc
pr1->intField - 804aed0
```

3.1.3 Referenzen und dynamische Objekte

Referenzen können auch durch Adressen von dynamischen Objekten (Variablen auf Heap) initialisiert werden. Das kann z.B. sofort nach Allokation und Konstruktion durch Dereferenzierung des durch `new` gelieferten *typgerechten* Pointers erledigt werden (`*new classname(args)`). Siehe nachfolgendes Beispiel:

```
// File: DynRef.cc

#include <iostream.h>

class DynRef {
public:
  DynRef(){cout << "Constructor." << endl;}
  ~DynRef(){cout << "Destructor." << endl;}
};

int main() {
  DynRef& rdf = *new DynRef;

  delete &rdf;

  return 0;
}
```

Frage:
Was ist falsch an folgender Funktion:

```
DynRef NewDynRef() {
  cout "Creating a new DynRef." << endl;

  return (*new DynRef);
}
```

3.1.4 Pointer und Referenzen: Argumente, Returnwerte und Empfehlungen

Die Entscheidung, ob die Argumente für Funktionen und Methoden per Value, per Zeiger oder per Referenz übergeben bzw. geliefert werden sollen, fällt oft nicht leicht. Hier einige Empfehlungen:

Argumente

- Wenn eine Funktion (Methode) ein Argument nicht modifiziert (*IN-Argument*) und handelt es sich um ein Argument eines eingebauten Typs, dann sollte dieses per Value übergeben werden.

- Ist das IN-Argument ein Array, dann übergeben Sie dieses als `const typename*`.

- Ist das IN-Argument von selbstdefiniertem Typ (*Aggregat*), dann sollten Sie es per Reference to const (`const typename&`) übergeben.

- Modifiziert eine Funktion ein Argument von eingebautem Typ, dann übergeben Sie dieses bitte per Pointer (`typename*`). Dadurch wird dem Leser Ihres Quelltextes klar, daß das Argument (der Inhalt der Adresse) als *INOUT-Argument* zu betrachten ist.

- Soll ein INOUT-Argument von selbstdefiniertem Typ übergeben werden, verwenden Sie die per Reference Notation (`typename&`). Der Typ des Argumentes (die Klasse) enthält Informationen darüber, auf welche Elemente die Funktion zugreifen darf. Soll auf `private` member zugegriffen werden, dann muß die Funktion Methode oder `friend` der Klasse sein. Die Funktion kann also nicht beliebig auf Elemente des übergebenen Objektes zugreifen. Die Klasse hat die Kontrolle über den Zugriff.

Returnwerte

- Soll das Resultat der aufgerufenen Funktion nicht als *lvalue* verwendet werden (*OUT-Returnwert*) und ist der Returnwert von eingebautem Typ, dann sollte er per Value geliefert werden.

- Ist der OUT-Returnwert von selbstdefiniertem Typ und hat er eine Lebensdauer über das Ende der aufgerufenen Funktion hinaus, dann sollte er per Reference to const (`const typename&`) von dieser geliefert werden.

- Lebt das Objekt des Returnwertes nur bis zum Ende der aufgerufenen Funktion, muß der Returnwert von dieser per Value geliefert werden (Kopier-Konstruktor!).

- Soll die Funktion in verketteten Zusammenhängen verwendet werden (z.B. Operatoren), und verwendet die nachfolgend aufzurufende Funktion in der Kette das von der betrachteten aufgerufenen Funktion returnierte Objekt als INOUT-Argument, so muß dieses per Reference (`typename&`) geliefert werden. Hier wieder die Lebensdauer des Objektes beachten!

- Soll das Resultat der aufgerufene Funktion als *lvalue* verwendet werden (*INOUT-Returnwert*), so muß dieses ebenfalls per Reference (`typename&`) geliefert werden.

3.2 Klassenvariablen und Klassenmethoden

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

Instanzvariablen (*Datenelemente*)	Klassenvariablen
jedes Objekt besitzt eigene Instanzvariablen und belegt diese mit individuellen Werten	jedes Objekt teilt mit allen Objekten seiner Klasse Klassenvariablen, die folglich für jedes Objekt identische Werte besitzen (Klassenvariablen gibt es für jede konkrete Klasse genau einmal)
sind nur über Objekte zugreifbar	sind zugreifbar, ohne daß ein Objekt dieser Klasse existieren muß
werden durch Definition von Variablen bzw. Konstruktion von Objekten der Klasse erzeugt und initialisiert	müssen außerhalb der Klassendefinition definiert und initialisiert werden

- Beispiel:

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

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

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

void f() {
  A object;

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

- Klassenmethoden (*statische Elementfunktionen, statische Memberfunktionen*) unterscheiden sich von anderen Methoden in folgenden Punkten:

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

- Beispiel:

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

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

void f()
{
  A object;

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

3.2.1 Pointer auf Klassenmethoden

Klassenmethoden sind aufrufbar, ohne daß ein konkretes Objekt angegeben werden muß. Einzig der *scope* muß zur Qualifizierung des Namens angegeben werden, d.h. die Klasse, in welcher der Name der Klassenmethode bekannt ist. Klassenmethoden sind daher eigentlich normale "C"-Funktionen und sind dadurch eindeutig durch einen Zeiger auf eine solche Funktion referenzierbar. Betrachten wir noch einmal eine andere Definition einer `Receiver`-Klasse:

```
// File: PoToSMem.h

#include <iostream.h>

struct Receiver
{
  virtual void Receive1(int Arg);
  static void Receive2(int Arg);
};
```

Die Methode `Receive2` ist Klassenmethode zur Klasse `Receiver` und hat damit nicht ohne weiteres Zugriff auf Datenelemente eines Objektes dieser Klasse.

Die Berechnung der Adresse dieser Methode erfolgt wieder mit dem Adreß-Operator:

```
// File: PoToSMem.cc

#include <PoToSMem.h>

void Receiver::Receive1(int Arg) {
  cout << "Receiver Object at " << hex << unsigned(this) << dec
       << " received message \"Receive1\" and Arg " << Arg << endl;
}

void Receiver::Receive2(int Arg) {
  cout << "Receiver Class received message \"Receive2\" and Arg "
       << Arg << endl;
}

void sendO(Receiver& r, void (Receiver::* p)(int), int Arg) {
  (r.*p)(Arg);
}

#define sendC(Class, Ptr, Arg) (*Ptr)(Arg);

int main()
{
  void (Receiver::* Message1)(int) = &(Receiver::Receive1);
  void (*Message2)(int) = &(Receiver::Receive2);

  Receiver r1;

  sendO(r1, Message1, 111);
  sendC(Receiver, Message2, 222);

  return 0;
}
```

Beachten Sie, daß im obigen Beispiel der Aufruf einer Klassenmethode als Botschaft an die Klasse mit dem Präprozessor-Makro `sendC` simuliert wird. Der diesem Makro als Argument übergebene Klassenname dient ausschließlich der Anschaulichkeit.

Überlegung: *Kann eine Klassenmethoden eine konkrete Methode aufrufen? Wenn ja, warum? Wie kann eine Klassenmethode dazu gebracht werden, auf konkrete Datenelemente eines Objektes der Klasse zuzugreifen?*

4 Namespaces in C++

4.1 Motivation

Ein C++-Projekt besteht oft aus mehreren Quelldateien, die von mehreren Entwicklern bearbeitet werden. Die globalen Namen, die in den verschiedenen Quelldateien verwendet werden, gehören traditionell demselben Namensbereich `namespace` an. Werden die Module dann zum Endprodukt gelinkt, gibt es oft Linkerfehler aufgrund von Namenskonflikten (*multiple definition of 'symbol'*).
Durch die Einführung verschiedener Namensbereiche entsprechend dem Namensbereichmechanismus von C++ kann die Namensgebung geplant und Namenskonflikte somit vermieden werden. Bestehende Konflikte können sogar nachträglich noch behoben werden.
Folgende grundlegende Vorgehensweise:

1. Definieren eines Namensbereiches mit dem Schlüsselwort `namespace`
2. Zugriff auf Elemente eines Namensbereiches entweder explizit (`scope operator ::`) oder implizit nach `using`-Deklaration oder `using`-Anweisung (Unterschied)

4.2 Definieren von Namensbereichen

- ein Namensbereich wird durch eine `namespace`-Definition festgelegt:

```
namespace F {
  float x = 9;
}
```

- Namensbereiche können geschachtelt werden:

```
namespace G {
  using namespace F;
  float y = 2.0;
  namespace INNER_G {
    float z = 10.01;
  }
}
```

- die Definition eines Namensbereiches kann beliebig erweitert werden; Achtung: benötigte Erweiterungen vor der Benutzung des Namensbereiches vornehmen, der Compiler kennt bei einer `using`-Anweisung nur die bisher zum Namensbereich gehörigen Namen; Beispiel:

```
using namespace F; // xx wird hier nicht bekannt

namespace F {
  double xx = 99;
}
```

```
using namespace F; // hier wird xx bekannt
```

  - nach der zweiten Definition für `F` gehören dem Namensbereich `F` die zwei Symbole `float x` und `double xx` an
  - die Anweisung `using namespace F;` vor der zweiten Definition macht allerdings hier nur den Namen `float x` bekannt

- für definierte Namensbereiche können Aliase eingeführt werden:

```
namespace ALIAS_G = G;
namespace ALIAS_INNER_G = G::INNER_G;
```

- unbenannte Namensbereiche ersetzen die traditionellen `static`-Definitionen, um Namen nur innerhalb der Übersetzungseinheit bekannt zu machen:

```
#include <iostream.h>
extern void func(void);
namespace {
  float pi = 3.14;
}
int main() {
  cout << "pi = " << pi << endl;
  func();
}
```

  - innerhalb der in einer anderen Übersetzungseinheit definierten (`externen`) Funktion `func` kann auf das in diesem unbenannten Namensbereich vorhandene `pi` nicht zugegriffen werden
  - Die traditionelle `static`-Definition wird mit Einführung des Namensbereichmechanismus von C++ nicht mehr benötigt und sollte auch nicht mehr verwendet werden.

- eine Klassendefinition definiert zugleich auch einen Namensbereich, der allerdings nachträglich nicht mehr erweitert werden kann

4.3 Zugriff auf Elemente eines Namensbereiches

Für den Zugriff auf Elemente eines Namensbereiches gibt es folgende drei Möglichkeiten:

1. explizite Zugriffsqualifizierung, Beispiel:
   `G::INNER_G::z = 10.01;`

2. `using`-Deklaration, fügt den deklarierten Namen in den lokalen Namensbereich ein; Beispiele:

```
using G::INNER_G::z;
using A::a;
z = 10.01;
a = 111;
```

- using-Deklarationen fügen Aliase für die originalen Namen in den lokalen Sichtbarkeitsbereich ein; keine Beeinflussung des Typs, Linkage etc.
- redundante using-Deklarationen sind erlaubt

3. using-Anweisung, macht alle Namen des angegebenen Namensbereiches im lokalen verwendbar; Einfügung erfolgt erst mit Verwendung; Beispiele:

```
int main() {
  using namespace G;
  using namespace G::INNER_G;
  ...
}
```

- lokale Deklarationen haben stets Vorrang gegenüber denen durch using-Anweisungen eingefügten; siehe `float x` im nachfolgenden Beispiel
- using-Anweisungen können in Klassendefinitionen nicht verwendet werden

4.4 Beispiel

```
// File: NamSpace.cc
#include <iostream.h>
class A {
public:
  static int a;
  int print(){return a;}
};
namespace F {
  float x = 9;
};
namespace G {
  using namespace F;
  float y = 2.0;
  namespace INNER_G {
    float z = 10.01;
  };
};
int A::a = 0;
int main() {
//  using A::a; Das akzeptiert Watcom nicht. Auch O.K.
```

```
  using namespace G;
  using namespace G::INNER_G;
//  a = 111;    Muss daher dieses ersetzen durch:
  A::a = 111;
  float x = 19.1;
  A aa;
  cout << "x = " << x << endl;
  cout << "y = " << y << endl;
  cout << "z = " << z << endl;
  cout << "A::a = " << aa.print() << endl;
  cin.get(); return 0;
}
/* Ausgabe:
x = 19.1
y = 2
z = 10.01
A::a = 111 */
```

5 Weitere Aspekte der Vererbung in C++

5.1 Mehrfachvererbung

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

- Beispiel

```
class A {
  int x;
public:
  A(int _x): x(_x){}
  void printA()
    { cout << x << "\n"; }
};
```

```
class B {
  char y;
public:
  B(char _y): y(_y){}
  void printB()
    { cout << y << "\n"; }
};
```

```
class C: public A, public B {
  float z;
public:
  C(int _A, char _B, float _z)
  : A(_A), B(_B), z(_z){}
  void printC()
    { cout << z << "\n"; }
};
```

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

```
C c(1, 'A', 3.3);
```

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

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

- im Zusammenhang mit der Mehrfachvererbung stehen eine Reihe Probleme, die nachfolgend erörtert werden sollen

5.1.1 Problem 1: *Ambiguity*

Es entsteht ein Problem, wenn in den Basisklassen gleichnamige Methoden vereinbart sind:

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

```
class B {
  char y;
public:
  B(char _y): y(_y){}
  void print()
    { cout << y << "\n"; }
};
```

```
class C: public A, public B {
  float z;
public:
  C(int _A, char _B, float _z)
  : A(_A), B(_B), z(_z){}
  void printC()
    { cout << z << "\n"; }
};
```

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

```
C c(1, ’A’, 3.3);

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

Der Compiler kann nicht entscheiden, welche der beiden geerbten Methoden anzuwenden ist.
Lösung: Überschreiben der `print()`-Methode in `C`

```
class C: public A, public B {
  float z;
public:
  C(int _A, char _B, float _z)
  : A(_A), B(_B), z(_z){}
  void print()
    { A::print(); B::print(); printC(); }
  void printC()
    { cout << z << "\n"; }
};
```

5.1.2 Problem 2: *SubObjekt der indirekten Basisklasse mehrfach enthalten*

Das nächste Problem behandelt folgende sehr häufige Situation:

```
class S {
 int s;
public:
 S(int _s) : s(_s){}
 void printS()
   { cout << s << "\n"; }
};
```

```
class A: public S {
  int x;
public:
  A(int _x)
  : S(111), x(_x){}
  void printA()
    { cout << x << "\n"; }
};
```

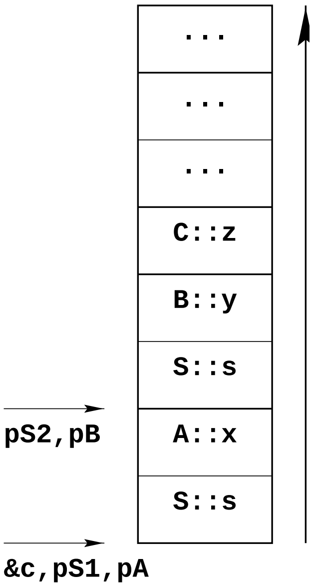
```
class B: public S {
  char y;
public:
  B(char _y)
   : S(222), y(_y){}
   void printB()
    { cout << y << "\n"; }
};
```

```
class C: public A, public B {
  float z;
public:
  C(int _A, char _B, float _z)
  : A(_A), B(_B), z(_z){}
  void printC()
  { printS(); printA(); printB();
    cout << z << "\n";
  }
};
```

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

```
C c(1, 'A', 3.3);
```

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



AmbBase.cc

Aufgrund der Reihenfolge der Konstruktoraufrufe werden diese Instanzvariablen folgendermaßen initialisiert:

```
A::S::s = 111
A::x    = 1
B::S::s = 222
B::y    = 'A'
C::z    = 3.3
```

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

```
MultBase.cc: In method 'void C::printC()':
MultBase.cc:37: request for method 'printS' is ambiguous
```

Lösung:

1. Spezifikation des „Erbweges“ beim Aufruf der Methode `printS()`:

```
class C: public A, public B {
  float z;
public:
  C(int _A, char _B, float _z)
  : A(_A), B(_B), z(_z){}
  void printC()
  { A::printS(); B::printS(); printA(); printB();
    cout << z << "\n";
  }
};
```

2. Verwendung virtueller Basisklassen (virtual base classes)

5.1.3 Problem 3: *Keine eindeutige Basisklasse*

```
// File: AmbBase.cc

#include <MultBase.h>

int main()
{
  C c(1, 'A', 3.3);
  c.printC();

  A* pA = &c;
  pA->printA();

  B* pB = &c;
  pB->printB();

  S* pS = &c;
  pS->printS();

  return 0;
}
```

```
AmbBase.cc: In function `int main()':
AmbBase.cc:16: type `S' is ambiguous base class for type `C'
```

Lösung:

1. Festlegung des Weges des *Upcasts* zur mehrfach vorhandenen Basisklasse durch expliziten (teilweisen) Upcast zum Zwecke der Eindeutigkeit.

```
// File: AmbBase.cc

#include <MultBase.h>
int main()
{
  C c(1, 'A', 3.3);
  c.printC();

  A* pA = &c;
  pA->printA();

  B* pB = &c;
  pB->printB();

  S* pS1 = (A*)(&c);
  pS1->printS();
```

```
    S* pS2 = (B*)(&c);
    pS2->printS();

  // obwohl c on Stack:
    unsigned diff = unsigned(pS2) - unsigned(pS1);
    cout << hex << "&c  = " << (&c)  << endl
                << "pS1 = " << (pS1) << endl
                << "pS2 = " << (pS2) << endl
         << dec << "Differenz (dec) = " << diff << endl
                << "sizeof(c)       = " << sizeof(c) << endl;
    return 0;
  }
```

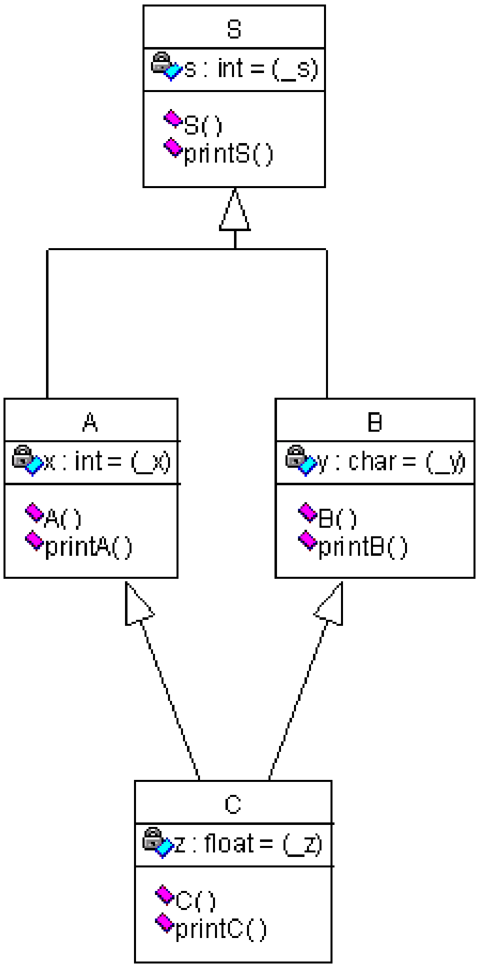
Programmausgabe:

```
111
222
1
A
3.3
1
A
111
222
&c  = 0xbffffaac
pS1 = 0xbffffaac
pS2 = 0xbffffab4
Differenz (dec) = 8
sizeof(c)       = 20
```

2. Verwendung virtueller Basisklassen (virtual base classes)

5.2 Virtuelle Vererbung

- virtuelle Basisklassen:




```
class S {
 int s;
public:
 S(int _s) : s(_s){}
 void printS()
   { cout << s << "\n"; }
};
```

```
class A: virtual public S {
  int x;
public:
  A(int _x)
```

```
class B: virtual public S {
  char y;
public:
  B(char _y)
```

```
  : S(111), x(_x){}
  void printA()
    { cout << x << "\n"; }
};
```

```
  : S(222), y(_y){}
  void printB()
   { cout << y << "\n"; }
};
```

```
class C: public A, public B {
  float z;
public:
  C(int _A, char _B, float _z)
  : S(333), A(_A), B(_B), z(_z){}
  void printC()
  { printS(); printA(); printB();
    cout << z << "\n";
  }
};
```

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

  1. die Konstruktoraufrufe `S(111)` bzw. `S(222)` aus `A` bzw. `B` werden ignoriert
  2. die Initialisierung der virtuellen Basisklasse (hier Klasse `S`) wird direkt von `C::C` aus vorgenommen, entweder explizit, z.B.:

     `C(int _A, char _B, float _z): S(333), A(_A), B(_B), z(_z){}`

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

- Mit virtuellen Basisklassen sind die Probleme Nr. 2 und 3 gelöst. Aber es tauchen neue Probleme auf, z.B.:

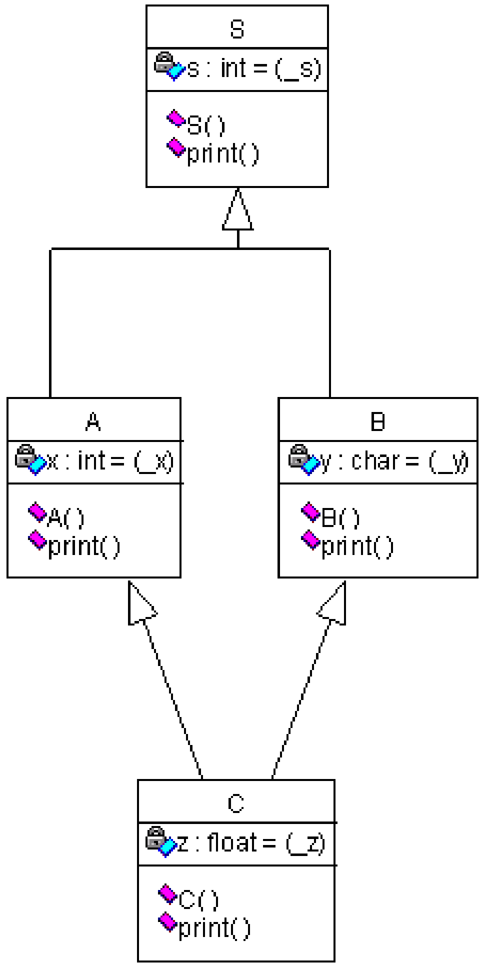
5.2.1 Problem 4: *Mehrfacher Aufruf derselben Methode der virtuellen Basisklasse durch die abgeleiteten Klassen*

Es ist wünschenswert, daß auch in einer Hierarchie mit virtueller Ableitung polymorphes Verhalten von Methoden gewährleistet ist. Betrachten Sie bitte dazu folgende Definitionen mit virtuellen Methoden:

```
// File: MultCall.h
#include <iostream.h>
class S {
  int s;
public:
  S(int _s) : s(_s){}
  virtual void print()
```

```
  { cout << s << "\n"; }
};
class A: virtual public S {
  int x;
public:
  A(int _x)
  : S(111), x(_x){}
  virtual void print()
  { S::print();
    cout << x << "\n"; }
};
class B: virtual public S {
  char y;
public:
  B(char _y)
  : S(222), y(_y){}
  virtual void print()
  { S::print();
    cout << y << "\n"; }
};
class C: public A, public B {
  float z;
public:
  C(int _A, char _B, float _z)
  : S(333), A(_A), B(_B), z(_z){}
  virtual void print()
  { A::print(); B::print();
    cout << z << "\n";
  }
};
```

Auf jeder Stufe der Hierarchie soll natürlich das gesamte Objekt, jeweils einschließlich des Subobjektes der virtuellen Basisklasse `S` ausgegeben werden. Die Klassen werden wie folgt verwendet:




```
// File: MultCall.cc

#include <MultCall.h>

int main()
{
  S& rS = *new C(1, 'A', 3.3);
  rS.print();
  delete &rS;

  return 0;
}
```

Analysiert man nun die Ausgabe des Programmes, dann fällt sofort auf, daß der im enthaltenen Subobjekt der virtuellen Basisklasse gespeicherte Wert `333` 2x ausgegeben wird:

```
333
1
333
A
3.3
```

Lösung:
Das kann nur verhindert werden, indem die virtuellen Methoden nicht auf der Basis der virtuellen Methoden der Basisklassen definiert werden, sondern ihre Funktionalität aus eigens dafür vorgesehenen, implementationsspezifischen Methoden „zusammensetzen“:

```
// File: SingCall.h

#include <iostream.h>
class S {
  int s;
protected:
  void _print()
  { cout << s << "\n"; }
public:
  S(int _s) : s(_s){}
  virtual void print()
  { _print(); }
};
class A: virtual public S {
  int x;
protected:
  void _print()
  { cout << x << "\n"; }
public:
  A(int _x)
  : S(111), x(_x){}
  virtual void print()
  { S::_print();
    _print(); }
};
class B: virtual public S {
  char y;
protected:
  void _print()
  { cout << y << "\n"; }
public:
  B(char _y)
  : S(222), y(_y){}
  virtual void print()
  { S::_print();
    _print(); }
```

```
};
class C: public A, public B {
  float z;
protected:
  void _print()
  { cout << z << "\n"; }
public:
  C(int _A, char _B, float _z)
  : S(333), A(_A), B(_B), z(_z){}
  virtual void print()
  { S::_print(); A::_print(); B::_print();
    _print(); }
};
```

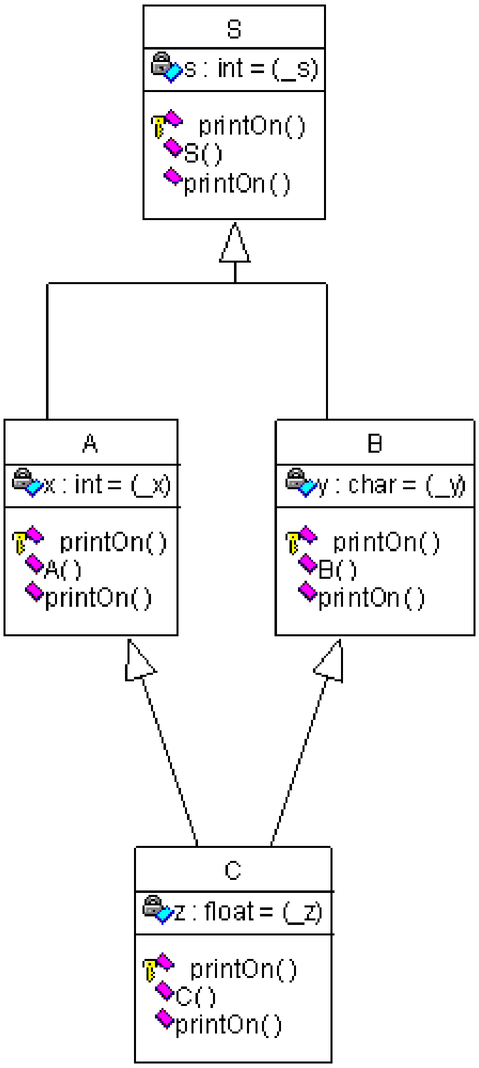
Folgende `main`-Funktion erzeugt dann korrekten Output:

```
// File: SingCall.cc

#include <SingCall.h>

int main()
{
  S& rS = *new C(1, 'A', 3.3);
  rS.print();
  delete &rS;

  return 0;
}
```

Ausgabe des Programmes:

```
333
1
A
3.3
```

Praktikumsaufgabe Nr. 3:



Bitte schreiben Sie `SingCall.h` *und* `SingCall.cc` *so um, daß die Programmausgabe auf beliebigen* `ostream`*-Objekten möglich wird! Definieren Sie dazu bitte nur einen (genau einen)* `ostream& operator<<(ostream& ros, S& rs)`*, typgenau für Objekte der virtuellen Basisklasse* `S`*. Dieser Operator soll durch polymorphen Aufruf anderer Methoden eine korrekte Ausgabe für Objekte beliebig abgeleiteter Klassen erzeugen. Speichern Sie bitte die Definition der neuen Klasse im File* `PROTECT/SiOpCall.h`*! Beachten Sie bitte folgende Verwendung:*

```
// File: SiOpCall.cc

#include <PROTECT/SiOpCall.h>
```

```
int main()
{
  S& rS = *new C(1, 'A', 3.3);
  cout << rS << endl;
  delete &rS;

  return 0;
}
```

Das Programm sollte z.B. folgenden Output erzeugen:

```
333 1 A 3.3
```

6 Konvertierungen und RTTI

C++ ist eine Programmiersprache mit strenger Typüberprüfung. Obwohl eine strenge Typisierung grundsätzlich für die Erstellung korrekter und robuster Programme von Vorteil ist (Entdeckung der Fehler durch falsche Konvertierungen hauptsächlich zur Compilezeit), sind häufig Konvertierungen notwendig, um bestimmte Effekte (einschließlich des polymorphen Verhaltens beim Methodenaufruf) zu erzielen. Konvertierungen werden häufig angewendet beim Aufruf von Funktionen, Methoden und Operatoren und bei Zuweisung und Initialisierung. Man unterscheidet hauptsächlich implizite und explizite, statische und dynamische Konvertierungen. Eine Konvertierung erzeugt nur dann einen *LValue*, wenn das Ergebnis eine Referenz ist.

6.1 Implizite Konvertierungen

Eine implizite Konvertierung liegt dann vor, wenn diese durch den Compiler anhand eingebauter oder anhand nutzerdefinierter Informationen selbständig eingefügt wurde. Standardkonvertierungen sind Konvertierungen zwischen eingebauten Typen. Für implizite benutzerdefinierte Konvertierungen können Konstruktoren und spezielle Konvertierungsoperatoren definiert werden.

6.1.1 Standardkonvertierungen

Eine Standardkonvertierung ist eine Konvertierung eines Wertes von einem eingebauten Typ in einen anderen eingebauten Typ. Standardkonvertierungen sind z.B. dann notwendig, wenn in einem binären Ausdruck (Vergleich, Zuweisung) die Operanden ungleiche Typen haben. In solchen Fällen wird der Compiler versuchen, die Typen der Operanden ohne Genauigkeitsverlust abzugleichen; der Wert mit der weniger genauen Darstellung wird zur genaueren Darstellung hin gewandelt.

6.1.2 Benutzerdefinierte implizite Konvertierungen

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

  Typ des Parameters → Klasse, zu der der Konstruktor gehört

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

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

  Objekt einer Klasse → Grunddatentyp (bzw. Zeiger darauf)

  anzugeben

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

`operator` *type*`()`

wobei *type* der gewünschte Zieltyp ist

Beispiel:

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

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

- Typkonvertierungsfunktionen haben selbst keinen Typ, da sich ihr Typ „automatisch" aus ihrem Namen ergibt
- diese Typkonvertierung stellt insofern keine Verletzung des `private`-Zugriffs dar, da das interne `char`-Array der Klasse `String` als `const char*` nach außen gegeben wird:

```
// File: KonvOp.cc

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

int main()
{
  String s("World!");
  char ac[100] = "Hello, ";

// cannot convert ‘s’ from type ‘String’ to type ‘char *’
//  strcpy(s, "BlaBla");

  cout << ac;
  strcat(ac, s);
  cout << s << endl;
  cout << ac << endl;
}
```

- Frage: was wird hier benutzt?

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

Möglich sind:

```
ostream& operator<<(ostream&, const String&);
ostream& operator<<(const char *s);
```

6.2 Explizite Konvertierungen

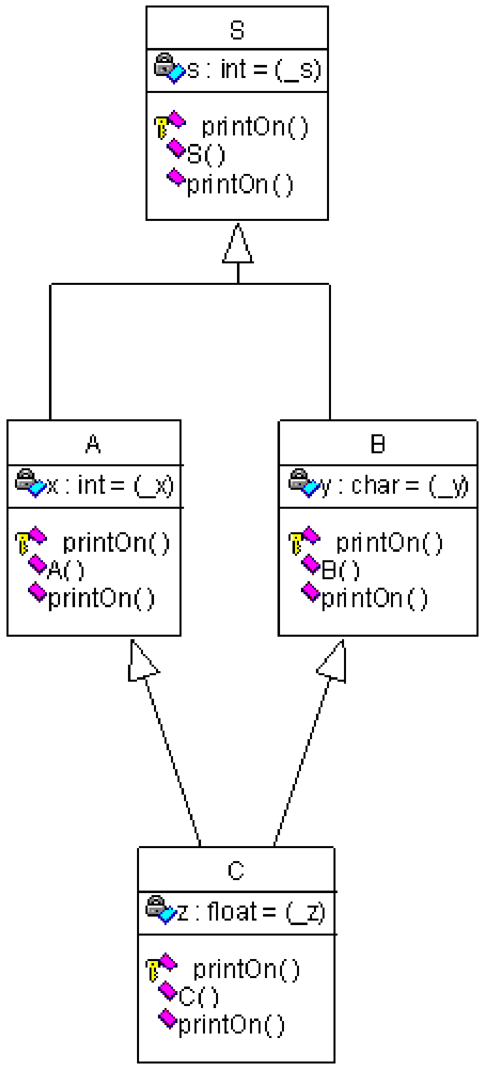
6.2.1 Upcasts und Downcasts

Betrachten wir noch einmal das Beispiel des Problems Nr. 3 von Seite 51.

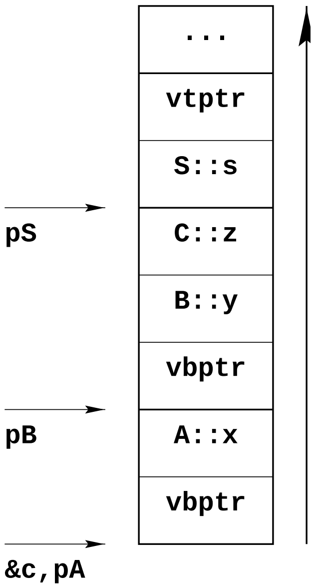
```
// File: AmbBase.cc

#include <MultBase.h>
int main()
{
  C c(1, 'A', 3.3);
  c.printC();

  A* pA = &c;
  pA->printA();

  B* pB = &c;
  pB->printB();

  S* pS1 = (A*)(&c);
  pS1->printS();
  S* pS2 = (B*)(&c);
  pS2->printS();

// obwohl c on Stack:
  unsigned diff = unsigned(pS2) - unsigned(pS1);
  cout << hex << "&c  = " << (&c)  << endl
              << "pS1 = " << (pS1) << endl
              << "pS2 = " << (pS2) << endl
       << dec << "Differenz (dec) = " << diff << endl
              << "sizeof(c)       = " << sizeof(c) << endl;
  return 0;
}
```

In diesem Beispiel waren *explizite Upcasts* notwendig, um danach implizit bis zu einer, durch den expliziten Upcast eindeutig bestimmten Basisklasse `S` weiterzukonvertieren. Wir sprechen von *Upcasts*, wenn in der Ableitungshierarchie „nach oben“ konvertiert werden soll. Umgekehrt sprechen wir von *Downcasts*, wenn in der Hierarchie in Richtung größerer Spezialisierung („nach unten“) konvertiert werden soll. Upcasts (entlang der IST-Relation) sind in der Regel implizit möglich, nämlich dann, wenn die entsprechende Basisklasse eindeutig und zugreifbar ist. Upcasts können auf Variablen für Objekte von Klassen wie auch auf Zeiger und Referenzen auf diese Objekte angewendet werden. Downcasts (umgekehrte IST-Relation) und Konvertierungen quer durch die Hierarchie sind nicht implizit möglich; sie müssen durch den Programmierer explizit verlangt werden. Downcasts können auf Zeiger und Referenzen auf Objekte von Klassen angewendet werden.

Um die Basisklasse `S` für Upcasts eindeutig werden zu lassen, führten wir virtuelle Ableitung ein.



Ein Objekt der Klasse `C` hat in etwa folgendes Layout:



UCastImp.cc

Das folgende Programm zeigt mögliche Konvertierungsoperationen in dieser Hierarchie:

```
// File: UCastImp.cc

#include <PROTECT/SiOpCall.h>
int main()
{
  C c(1, 'A', 3.3);
  cout << c << endl;
  A* pA = &c;              // Upcast
  cout << (*pA) << endl;
  B* pB = &c;              // Upcast
  cout << (*pB) << endl;
  S* pS = &c;              // Upcast
  cout << (*pS) << endl;

// Fuer den Compiler ist es "UP";
// fuer uns ist es "DOWN":
// cannot cast up from virtual baseclass 'S'
// C* pC = (C*)pS;         // Downcast
  C* pC = (C*)pA;          // Downcast
  cout << (*pC) << endl;
  B* pQ = (B*)(C*)pA;      // 'Quer'cast
  cout << (*pQ) << endl;
  unsigned diff = unsigned(pQ) - unsigned(pA);
  cout << hex << "&c   = " << (&c) << endl
             << "pA   = " << (pA) << endl
             << "pB   = " << (pB) << endl
             << "pS   = " << (pS) << endl
```

```
                 << "pC  = " << (pC) << endl
                 << "pQ  = " << (pQ) << endl
          << dec << "Differenz (dec) = " << diff << endl
                 << "sizeof(c)       = " << sizeof(c) << endl;
   A a(2);
   pA = &a;
   pC = (C*)pA;          // Error: Casting down to C
   cout << (*pC) << endl; // ??? virtuelle Meth., evtl. Glueck
   return 0;
}
```

Der Programmierer sollte bei Anwendung eines Downcasts sicher sein, daß das ursprünglich referenzierte Objekt auch mindestens dem Zieltyp der Konvertierungsoperation entspricht.
Das Programm (oben) liefert in etwa folgenden Output:

```
 333 1 A 3.3
 333 1 A 3.3
 333 1 A 3.3
 333 1 A 3.3
 333 1 A 3.3
 333 1 A 3.3
&c  = 0xbffffaa4
pA  = 0xbffffaa4
pB  = 0xbffffaac
pS  = 0xbffffab8
pC  = 0xbffffaa4
pQ  = 0xbffffaac
Differenz (dec) = 8
sizeof(c)       = 28
 111 2
```

6.3 Statische und dynamische Konvertierungen

Die bisher genannten impliziten und expliziten Konvertierungsoperationen waren durchweg *statisch*. Eine statische Konvertierung kann bereits vom Compiler vorgenommen werden, da hierfür nur das Wissen über die Ableitungshierarchie vorhanden sein muß. Eine implizite Konvertierung ist demnach immer auch statisch.
Wozu werden dann *dynamische* Konvertierungen benötigt?
Dynamische Konvertierungen sind Downcasts oder Konvertierungen quer durch die Hierarchie zur Laufzeit. Die dynamische Überprüfung einer solchen Konvertierung soll z.B. die o.g. Aussage zur Sicherheit eines Downcasts umsetzen helfen.
Stellen Sie sich folgende Situation vor:
Bei Verarbeitung der Objekte in einem Container konnten bis hierhin nur durch die Verwendung virtueller Funktionen typselektive Operationen durchgeführt werden. Manchmal ist es aber notwendig, den genauen Typ eines vom Container

gelieferten Objektes zu kennen, z.B. wenn ein solches Objekt kopiert werden soll und Speicher der korrekten Größe zur Verfügung gestellt werden muß. Ein Container kennt nur die Schnittstelle der enthaltenen Objekte, d.h. den Mindesttyp der Objekte, für die er gemacht wurde. In Umkehrung der oben erwähnten impliziten und statischen Upcasts sind oft Downcasts zu den Typen der konkret referenzierten Objekte notwendig; wobei eben diese Typen zur Compilezeit noch nicht bekannt sein müssen.
Ein Downcast kann auf zwei verschiedenen Wegen durchgeführt werden:

- traditionelle explizite Konvertierung:
  z.B. `C* pC = (C*)pA;`
  - ist gefährlich und ein grober Fehler, wenn durch `pA` nicht wirklich ein Objekt der Klasse `C` referenziert wird
- mit RTTI: dynamische explizite Konvertierung:
  z.B. `C* pC = dynamic_cast<C*>(pA);`
  - ist rel. sicher, da im Fehlerfalle eines Pointercasts ein NULL-Pointer zurückgeliefert und im Fehlerfalle eines Referenzcasts eine Ausnahme ausgeworfen wird

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

6.4 Verfügbarmachen von Typinformationen zur Laufzeit

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

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

6.5 Typinformationen abfragen und verwenden

Zur Abfrage der Typinformation dient der Operator `typeid`:

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

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

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

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

- Der `typeid`-Operator sollte mit eingebauten Basistypen genauso funktionieren wie mit selbstdefinierten C++-Typen (Vorsicht hier beim GNU-Compiler!).
- Wenn der Operand ein dereferenzierter Nullzeiger ist, dann wird in der Regel eine Exception (`bad_typeid`) ausgelöst.
- Das zurückgelieferte `typeinfo`-Objekt besitzt Methoden bzw. Operatorfunktionen zum Vergleich mit anderen `typeinfo`-Objekten sowie zur Ausgabe der Typbezeichnung als `char[]`.

6.6 Möglichkeiten der Typumwandlung mit RTTI

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

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

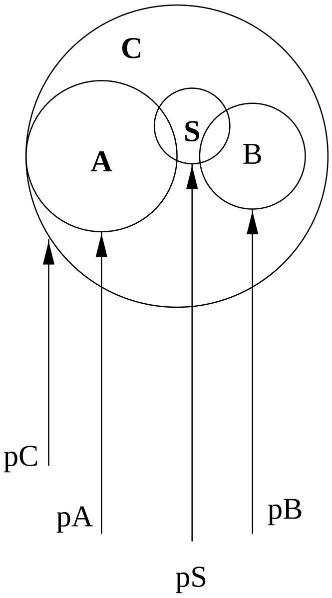
  - arbeitet analog der impliziten Typkonvertierung; d.h. gibt es eine implizite Typkonvertierung von Typ von `arg` nach `T`, dann ist `static_cast` korrekt, ansonsten ein Fehler
  - die Typumwandlung findet ebenfalls beim Compilieren statt

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

- `dynamic_cast<T>(arg)`
  - dient zur Konvertierung eines Zeigers oder einer Referenz auf ein Objekt einer Basisklasse in einen Zeiger bzw. Referenz auf ein Objekt einer abgeleiteten Klasse, wenn das referenzierte Objekt tatsächlich vom abgeleiteten Typ ist
  - Konvertierungen in der Ableitungshierarchie nach oben (statische) werden bereits beim Compilieren eingesetzt und funktionieren analog `static_cast`
  - Konvertierungen in der Ableitungshierarchie nach unten (dynamische) werden erst zur Laufzeit berechnet
  - `arg` und `T` müssen entweder beide einen Pointer oder beide einen Referenz-Typen haben
  - wenn `arg` und `T` Pointer sind, und die Konvertierung kann nicht durchgeführt werden, wird ein NULL-Zeiger zurückgeliefert
  - wenn `arg` und `T` Referenzen sind, und die Konvertierung kann nicht durchgeführt werden, wird in der Regel eine Exception (`bad_cast`) ausgelöst
  - Sonderfall: wenn `T` ein `void*` ist, muß `arg` ebenfalls ein Pointer sein; in dem Fall wird ein Pointer geliefert, der auf das komplette Objekt zeigt, also das Objekt der Klasse, die sich in der Hierarchie am weitesten unten befindet

6.7 Beispiel `mainRTTI_1`

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



`pC` zeigt auf ein Objekt der Klasse `C`. Die Klasse `C` hat zwei direkte Basisklassen, `A` und `B` und eine indirekte Basisklasse, `S` (virtuelle Ableitung). Somit können zwischen den Pointern `pA`, `pB`, `pC` und `pS` z.B. folgende korrekte Typkonvertierungen formuliert werden:

`pC` → `pA`	implizit	statisch	`static_cast<A*>(pC)`
`pC` → `pB`	implizit	statisch	`static_cast<B*>(pC)`
`pC` → `pS`	implizit	statisch	`static_cast<S*>(pC)`
`pS` → `pC`	explizit	dynamisch	`dynamic_cast<C*>(pS)`
`pA` → `pC`	explizit	dynamisch	`dynamic_cast<C*>(pA)`
`pB` → `pC`	explizit	dynamisch	`dynamic_cast<C*>(pB)`
`pA` → `pB`	explizit	dynamisch	`dynamic_cast<B*>(pA)`
`pB` → `pA`	explizit	dynamisch	`dynamic_cast<A*>(pB)`

Der Quelltext des Beipielprogrammes:

```
// File: DCastExp.cc

#include <typeinfo>
#include <PROTECT/SiOpCall.h>

int main()
{
  C c(1, 'A', 3.3);
  cout << c << endl;

  A* pA = static_cast<A*>(&c);            // Upcast
  cout << (*pA) << endl;

  B* pB = static_cast<B*>(&c);            // Upcast
  cout << (*pB) << endl;
```

```

  S* pS = static_cast<S*>(&c);            // Upcast
  cout << (*pS) << endl;

  C* pC  = dynamic_cast<C*>(pS);          // Downcast
  C* pCA = dynamic_cast<C*>(pA);          // Downcast
  C* pCB = dynamic_cast<C*>(pB);          // Downcast

  if((pC == pCA) && (pC == pCB) && (pC == &c))
    cout << (*pC) << endl;
  else
    cout << "Downcast-Error: Not O.K." << endl;

  B* pQ = dynamic_cast<B*>(pA);           // 'Quer'cast
  if(pQ == pB)
    cout << (*pQ) << endl;
  else
    cout << "Cast-Error: Not O.K." << endl;

  unsigned diff = unsigned(pQ) - unsigned(pA);

  cout << hex << "&c  = " << (&c) << endl
              << "pA  = " << (pA) << endl
              << "pB  = " << (pB) << endl
              << "pS  = " << (pS) << endl
              << "pC  = " << (pC) << endl
              << "pQ  = " << (pQ) << endl
       << dec << "Differenz (dec) = " << diff << endl
              << "sizeof(c)       = " << sizeof(c) << endl;

  A a(2);
  pA = &a;
  A& ra = a;

  pC = dynamic_cast<C*>(pA);              // Error: Casting down to C
  if(!pC)
  {
    cout << "Downcast-Error: But O.K." << endl;
    cout << "The real type of (*pA) is " << typeid(*pA).name() << endl;
  }
  else
    cout << (*pC) << endl;                // ???

  try
  {
    C& rc = dynamic_cast<C&>(ra);         // Error: Casting down to C
  }
```

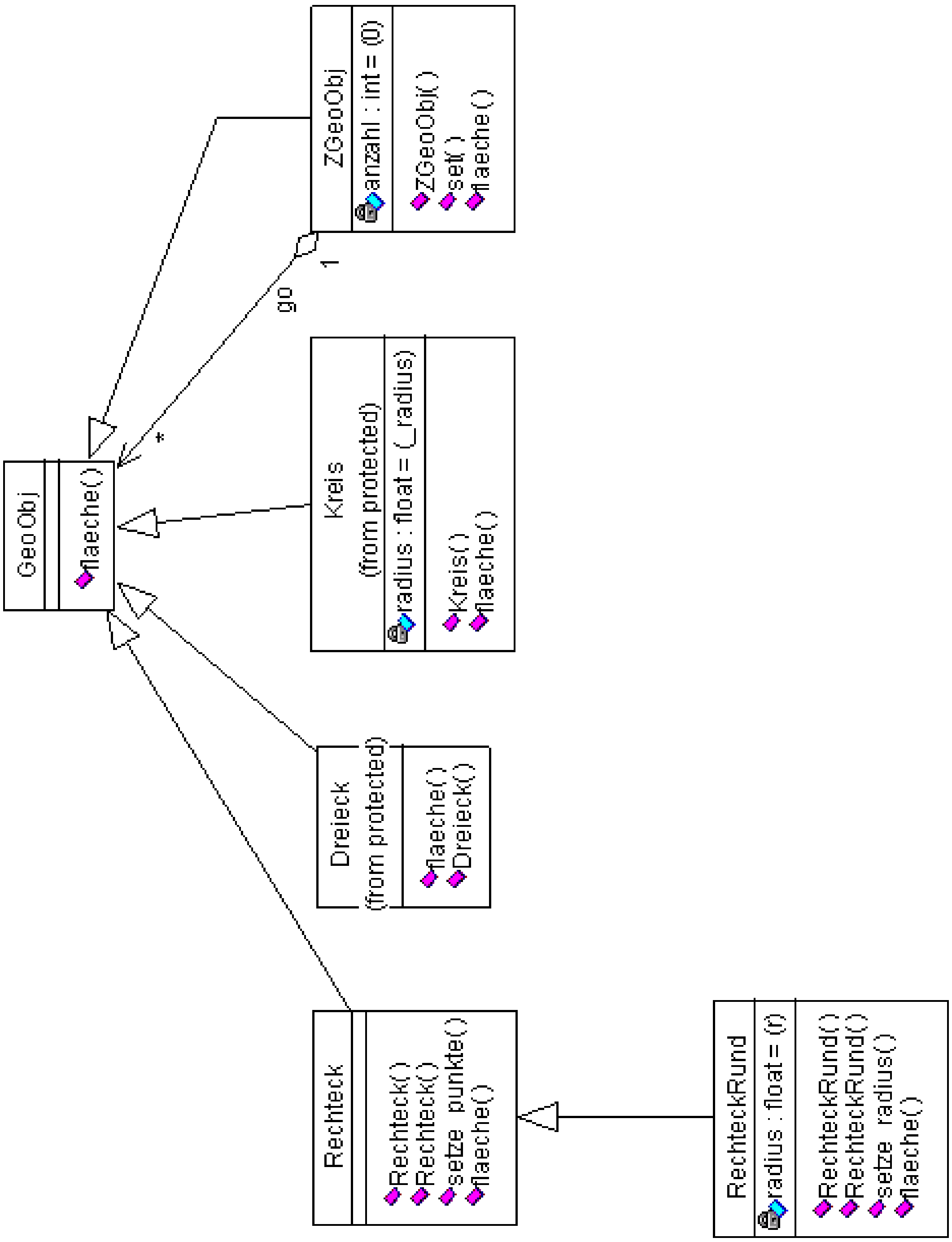
```
    catch(bad_cast& rbc)
    {
      cout << "Exception handled: " << rbc.what() << endl;
    }

    return 0;
  }

```

Folgende Ausgabe kann mit dem Beispielprogramm erzeugt werden:

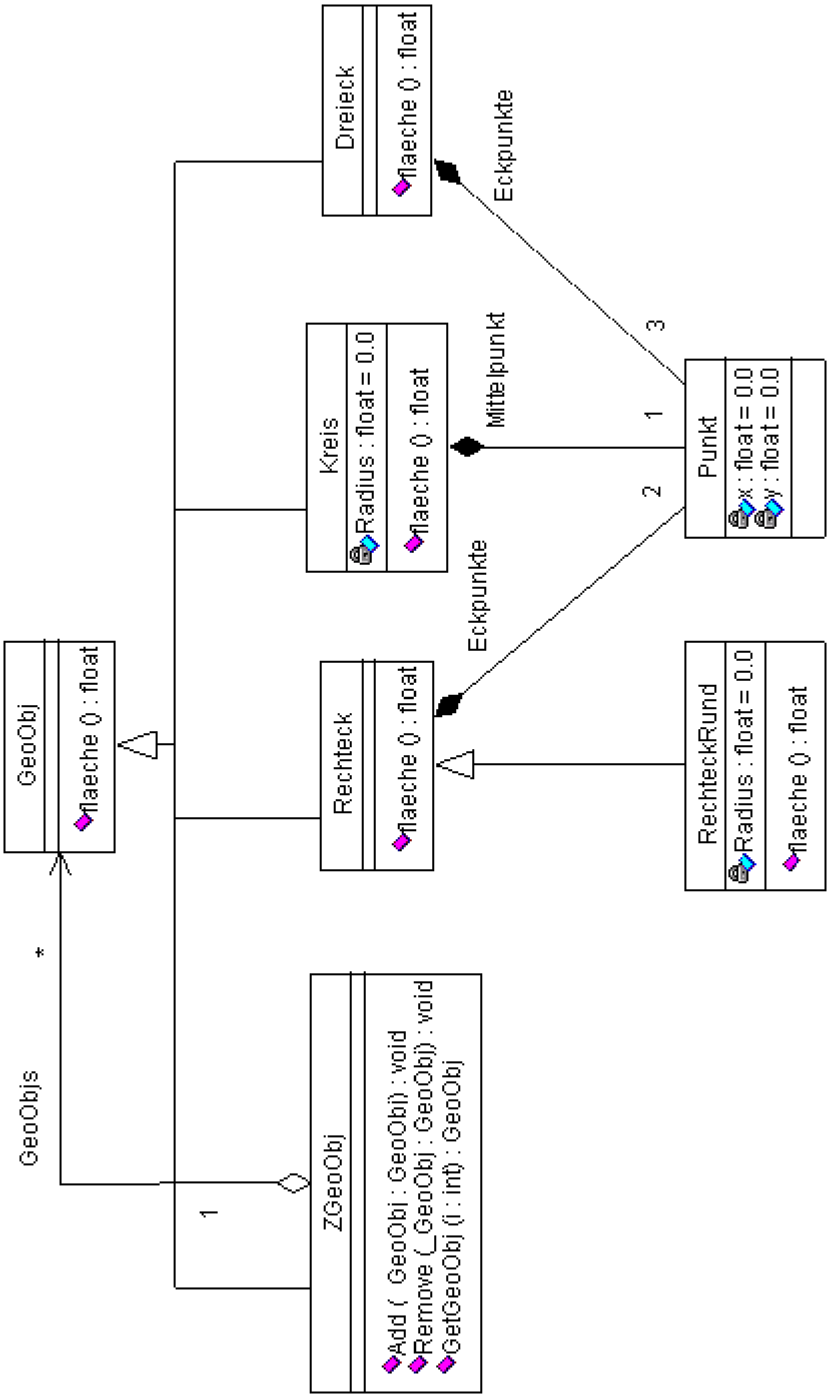
```
 333 1 A 3.3
 333 1 A 3.3
 333 1 A 3.3
 333 1 A 3.3
 333 1 A 3.3
 333 1 A 3.3
&c  = 0xbffffa9c
pA  = 0xbffffa9c
pB  = 0xbffffaa8
pS  = 0xbffffab8
pC  = 0xbffffa9c
pQ  = 0xbffffaa8
Differenz (dec) = 12
sizeof(c)       = 36
Downcast-Error: But O.K.
The real type of (*pA) is TD.1A
Exception handled: bad_cast
```

6.8 Praktikumsbeispiel: Verwendung von RTTI

Praktikumsaufgabe Nr. 4:
Wir betrachten das `ZGeoObj`*-Beispiel aus dem Teil „A“:*



Soll nun das Composite-*Pattern (2.1, Seite 27) implementiert werden, so muß es eine Funktionalität zum Hinzufügen und auch zum Entfernen von* `Components` *(*`GeoObjs`*) aus der* `Composition` *(*`ZGeoObjs`*) geben:*



D.h. folgende Schnittstelle muß diesbezüglich in `ZGeoObj` *implementiert werden:*

```
ZGeoObj();
ZGeoObj(const ZGeoObj &right);

const ZGeoObj& operator=(const ZGeoObj &right);

void Add(GeoObj& _GeoObj);
GeoObj* Remove(GeoObj& _GeoObj);

GeoObj* GetGeoObj(int i);
```

Um beliebige `GeoObjs` *aus einem* `ZGeoObj` *hinzufügen und auch wieder entfernen zu können, muß es einen Vergleich zwischen* `GeoObjs` *geben. Die Betonung liegt auf dem Vergleich von* `GeoObjs`*, da der Compiler nicht wissen kann, an welcher Stelle im* `ZGeoObj` *welcher genaue Typ eines* `GeoObjs` *gespeichert ist. Es müssen also* `Rechtecke` *mit* `Dreiecken` *etc. verglichen werden können. Diesbezüglich muß es also folgende Operatoren geben:*

```
int operator==(const GeoObj& left, const GeoObj& right);
int operator!=(const GeoObj& left, const GeoObj& right);
```

Um diesen Verleich zu implementieren, können z.B. Typabfragen bzw. dynamische Downcasts verwendet werden.

- *Implementieren Sie daher bitte das* Composite-*Pattern auf der Basis der im Teil „A" entstandenen* `GeoObj`*-Hierarchie und verwenden Sie dazu den RTTI-Mechanismus von C++!*
- *Weisen Sie bitte nach, daß im neuen Beispiel die Methode* `float flaeche()` *z.B. der* `ZGeoObj`*-Objekte wie im Teil „A" korrekt funktioniert.*
- *Weisen Sie bitte nach, daß auch das selektive Entfernen von Objekten aus* `ZGeoObjs` *funktioniert!*
- *Starten Sie bitte dynamische Typabfragen auf in* `ZGeoObjs` *enthaltene* `GeoObjs`*!*
- *Starten Sie bitte Initialisierungen durch und Zuweisungen an* `ZGeoObjs`*!*
- *Weisen Sie nach, daß dynamische Downcasts wirklich abgelehnt werden, wenn der Zieltyp nicht dem konkret referenzierten Objekt entspricht!*
- *Weisen Sie die Gefahr unendlicher Rekursionen im Zusammenhang mit der* `float flaeche()`*-Methode nach!*
- *Welche Möglichkeiten gibt es zum Ersatz der Nutzung des RTTI-Mechanismus?*

Hinweis:
Nehmen Sie als Vorlage die Files:
`DreieckV.h KreisV.h RchteckV.h ZGeoObjV.h GeoObjV.h RchtckRV.h PunktV.h`
und
`DreieckV.cc KreisV.cc RchteckV.cc ZGeoObjV.cc GeoObjV.cc RchtckRV.cc`

`PunktV.cc`

Speichern Sie Ihre eigenen Files bitte unter `PROTECT` *ab! Verwenden Sie für dieses Beispiel anstelle des Suffixes* `V` *in den Filenamen bitte ein* `R` *(also* `RchteckR.h` *...)! Ich schlage folgende* `main()`*-Funktion vor:*

```
// File: mGeoR.cc

#include <typeinfo>
#include <iostream.h>
#include <PunktR.h>
#include <PROTECT/RchteckR.h>
#include <PROTECT/RchtckRR.h>
#include <PROTECT/DreieckR.h>
#include <PROTECT/KreisR.h>
#include <PROTECT/ZGeoObjR.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;

  {  // Dieser Block erzeugt die Ausgabe wie im Teil 'A':
     cout << "Flaeche r1: " << r1.flaeche() << endl;
     cout << "Flaeche r2: " << r2.flaeche() << endl;
     cout << "Flaeche  d: " << d.flaeche() << endl;
     z1.Add(r1);
     z1.Add(d);
     cout << endl;
     cout << "Flaeche z1: " << z1.flaeche() << endl;
     z2.Add(z1);
     z2.Add(r2);
     cout << "Flaeche z2: " << z2.flaeche() << endl;

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

Flaeche z1: 42
Flaeche z2: 87
*/
  }
  cout << "-------------" << endl;
  {  // Testen der Entfernen-Operation:
     GeoObj* pr1 = z1.GetGeoObj(0);
```

```
      if(*pr1 == r1)
        cout << "z1[0] ist Rechteck r1 mit Flaeche "
             << pr1->flaeche() << endl;

      z2.Add(*(z1.Remove(*pr1)));
      cout << "r1 aus z1 entfernt. Flaeche z1 jetzt: "
           << z1.flaeche() << endl;
      cout << "r1 zu z2 addiert. Flaeche z2 noch immer: "
           << z2.flaeche() << endl;
   }
   {  // Testen der Typabfragen:
      GeoObj* pz1 = z2.GetGeoObj(0);
      if(*pz1 == z1)
        cout << "z2[0] ist ZGeoObj z1 mit Flaeche "
             << pz1->flaeche() << endl;

      if(typeid(*(z1.GetGeoObj(1))) == typeid(Dreieck))
        cout << "z1[1] ist ein Dreieck" << endl;
   }
   ZGeoObj nz2(z2);
   {  // Testen der Initialisierung/Zuweisung:
      cout << "neues ZGeoObj nz2 mit Flaeche "
           << nz2.flaeche() << endl;
      nz2 = z1;
      cout << "neues ZGeoObj nz2 jetzt mit Flaeche "
           << nz2.flaeche() << endl;

      nz2.Add(*(new Kreis(Punkt(3.0, 4.0), 10.0)));
      Kreis *pk = dynamic_cast<Kreis*>(nz2.GetGeoObj(2));

      if(pk)
        cout << "nz2[2] ist dynamischer Kreis mit Flaeche: "
             << pk->flaeche() << endl;
      cout << "Flaeche nz2 daher jetzt: " << nz2.flaeche() << endl;
      delete nz2.Remove(*pk);
      cout << "Kreis aus nz2 entfernt und geloescht. Flaeche wieder: "
           << nz2.flaeche() << endl;
   }
   {  // Testen der Downcasts:
      Dreieck *pd = dynamic_cast<Dreieck*>(nz2.GetGeoObj(1));
      if(pd)
        cout << "Downcast zu Dreieck erfolgreich." << endl;
      else
        cout << "Downcast zu Dreieck abgelehnt." << endl;

      Kreis *pk = dynamic_cast<Kreis*>(nz2.GetGeoObj(1));
      if(pk)
```

```
          cout << "Downcast zu Kreis erfolgreich." << endl;
        else
          cout << "Downcast zu Kreis abgelehnt." << endl;
    }
    {  // unendliche Rekursion:
        z1.Add(z2);
        cout << "Flaeche von z1: " << flush;
        cout << z1.flaeche() << endl;
    }
  }
```

Das Programm sollte folgenden Output bringen:

```
Flaeche r1: 30
Flaeche r2: 45
Flaeche  d: 12

Flaeche z1: 42
Flaeche z2: 87
--------------
z1[0] ist Rechteck r1 mit Flaeche 30
r1 aus z1 entfernt. Flaeche z1 jetzt: 12
r1 zu z2 addiert. Flaeche z2 noch immer: 87
z2[0] ist ZGeoObj z1 mit Flaeche 12
z1[1] ist ein Dreieck
neues ZGeoObj nz2 mit Flaeche 87
neues ZGeoObj nz2 jetzt mit Flaeche 12
nz2[2] ist dynamischer Kreis mit Flaeche: 314.159
Flaeche nz2 daher jetzt: 326.159
Kreis aus nz2 entfernt und geloescht. Flaeche wieder: 12
Downcast zu Dreieck erfolgreich.
Downcast zu Kreis abgelehnt.
Flaeche von z1: Segmentation fault (core dumped)
```

6.9 Praktikumsbeispiel: Ersatz für RTTI

Praktikumsaufgabe Nr. 5:
Die Verwendung der RTTI-Funktionalität ist oft unerwünscht. RTTI wird oft als nicht portabel genug oder unsicher angesehen. Tatsächlich kann die Verwendung der RTTI durch einen guten Entwurf vollständig vermieden oder durch virtuelle Funktionen und abstrakte Klassen ersetzt werden. Eine solche Ersetzung wird sich dann in der Regel besser in das Gesamtkonzept einfügen als die Verwendung der RTTI. So auch in unserem Beispiel:
Bitte tauschen Sie die Verwendung der RTTI-Funktionalität im Praktikumsbeispiel 4 auf Seite 73 vollständig gegen objektorientierte Prinzipien (virtuelle Funktionen) aus! Entfernen Sie das Includieren von `typeinfo` *und die Übersetzung mit Einfügen der RTTI!*

Frage:
Wie kann man eine Typinformation auch von einer Klasse abfragen?
Hinweis:
Speichern Sie Ihre eigenen Files bitte unter `PROTECT` *ab! Verwenden Sie jetzt anstelle des Suffixes* `R` *in den Filenamen bitte ein* `N` *(also* `RchteckN.h` *...)! Ich schlage folgende* `main()`*-Funktion vor:*

```
// File: mGeoN.cc

#include <iostream.h>
#include <PunktN.h>
#include <PROTECT/RchteckN.h>
#include <PROTECT/RchtckRN.h>
#include <PROTECT/DreieckN.h>
#include <PROTECT/KreisN.h>
#include <PROTECT/ZGeoObjN.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;

  {  // Dieser Block erzeugt die Ausgabe wie im Teil 'A':
     cout << "Flaeche r1: " << r1.flaeche() << endl;
     cout << "Flaeche r2: " << r2.flaeche() << endl;
     cout << "Flaeche  d: " << d.flaeche() << endl;
     z1.Add(r1);
     z1.Add(d);
     cout << endl;
     cout << "Flaeche z1: " << z1.flaeche() << endl;
     z2.Add(z1);
     z2.Add(r2);
     cout << "Flaeche z2: " << z2.flaeche() << endl;

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

Flaeche z1: 42
Flaeche z2: 87
*/
  }
  cout << "-------------" << endl;
  {  // Testen der Entfernen-Operation:
```

```
        GeoObj* pr1 = z1.GetGeoObj(0);
        if(*pr1 == r1)
          cout << "z1[0] ist Rechteck r1 mit Flaeche "
               << pr1->flaeche() << endl;

        z2.Add(*(z1.Remove(*pr1)));
        cout << "r1 aus z1 entfernt. Flaeche z1 jetzt: "
             << z1.flaeche() << endl;
        cout << "r1 zu z2 addiert. Flaeche z2 noch immer: "
             << z2.flaeche() << endl;
     }
     {  // Testen der Typabfragen:
        GeoObj* pz1 = z2.GetGeoObj(0);
        if(*pz1 == z1)
          cout << "z2[0] ist ZGeoObj z1 mit Flaeche "
               << pz1->flaeche() << endl;

        if((z1.GetGeoObj(1))
            && (z1.GetGeoObj(1)->GetTypeString() == "[Dreieck]"))
          cout << "z1[1] ist ein Dreieck" << endl;
     }
     ZGeoObj nz2(z2);
     {  // Testen der Initialisierung/Zuweisung:
        cout << "neues ZGeoObj nz2 mit Flaeche "
             << nz2.flaeche() << endl;
        nz2 = z1;
        cout << "neues ZGeoObj nz2 jetzt mit Flaeche "
             << nz2.flaeche() << endl;

        nz2.Add(*(new Kreis(Punkt(3.0, 4.0), 10.0)));
        GeoObj* pk = nz2.GetGeoObj(2);
        if((pk) && (nz2.GetGeoObj(2)->GetTypeString() == "[Kreis]"))
          cout << "nz2[2] ist dynamischer Kreis mit Flaeche: "
               << pk->flaeche() << endl;
        cout << "Flaeche nz2 daher jetzt: " << nz2.flaeche() << endl;
        delete nz2.Remove(*pk);
        cout << "Kreis aus nz2 entfernt und geloescht. Flaeche wieder: "
             << nz2.flaeche() << endl;
     }
     {  // unendliche Rekursion:
        z1.Add(z2);
        cout << "Flaeche von z1: " << flush;
        cout << z1.flaeche() << endl;
     }
  }
```

Das Programm sollte folgenden Output bringen:

```
Flaeche r1: 30
Flaeche r2: 45
Flaeche  d: 12

Flaeche z1: 42
Flaeche z2: 87
--------------
z1[0] ist Rechteck r1 mit Flaeche 30
r1 aus z1 entfernt. Flaeche z1 jetzt: 12
r1 zu z2 addiert. Flaeche z2 noch immer: 87
z2[0] ist ZGeoObj z1 mit Flaeche 12
z1[1] ist ein Dreieck
neues ZGeoObj nz2 mit Flaeche 87
neues ZGeoObj nz2 jetzt mit Flaeche 12
nz2[2] ist dynamischer Kreis mit Flaeche: 314.159
Flaeche nz2 daher jetzt: 326.159
Kreis aus nz2 entfernt und geloescht. Flaeche wieder: 12
Flaeche von z1: Segmentation fault (core dumped)
```

7 Streams in C++: erweiterte Kapitel

7.1 Manipulatoren

- Einstellung des Formatstatus über die `setf()`-Aufrufe ist unbequem; man wünscht sich eine streamorientiertere Notation, z.B.
  `cout << i << oct << i << hex << i << endl;`

- `oct`, `hex` und `endl` sind hierbei Manipulatoren für den Stream `cout`

Manipulatoren ohne Argumente

- einfache Manipulatoren sind Zeiger auf Funktionen mit folgenden Typdefinitionen:

```
typedef ios& (*__manip)(ios&);
typedef istream& (*__imanip)(istream&);
typedef ostream& (*__omanip)(ostream&);
```

- für diese Manipulator-Typen sind in den einzelnen Stream-Klassen folgende `operator>>`-Operatorfunktionen definiert:

```
ostream& ostream::operator<<(__manip func)
  {(*func)(*this); return *this;}
ostream& ostream::operator<<(__omanip func)
  { return (*func)(*this);}

istream& istream::operator>>(__manip func)
  {(*func)(*this); return *this;}
istream& istream::operator>>(__imanip func)
  { return (*func)(*this); }
```

- folgende Manipulatoren sind auf diese Art und Weise realisiert (iostream.h einbinden!):

```
extern istream& ws(istream& ins);     // whitesp. aus istream entf.
extern ostream& flush(ostream& outs); // Puffer eines ostream leeren
extern ostream& endl(ostream& outs);  // '\n' an ostream anhaengen
extern ostream& ends(ostream& outs);  // '\0' an ostream anhaengen

inline ios& dec(ios& i)
{ i.setf(ios::dec, ios::dec|ios::hex|ios::oct); return i; }
inline ios& hex(ios& i)
{ i.setf(ios::hex, ios::dec|ios::hex|ios::oct); return i; }
inline ios& oct(ios& i)
{ i.setf(ios::oct, ios::dec|ios::hex|ios::oct); return i; }
```

Beispiel:

Programmausgabe:

```
Zwei Zahlen, getrennt durch Whitespaces, eingeben:
|16|32|
|20|40|
|10|20|
```

Manipulatoren mit Argumenten

- Manipulationen mit Argumenten werden analog denen ohne Argumente indirekt über einen Funktionsaufruf realisiert. Die dazu gerufene Funktion nimmt jetzt neben der Referenz auf einen Stream zusätzlich noch das Manipulator-Argument, Beispiel:

  ```
  ios& width(ios& ros, int n) {
    ros.width(n);
    return ros;
  }
  ```

- da ein `operator<<` neben dem Stream nur noch ein Argument haben kann, muß die Manipulation mit einem Argument abgebildet werden auf die Streamausgabe eines temporären Objektes, welches oben beschriebene Funktion und das Manipulator-Argument speichert

- bei einer Manipulation wie z.B.
  `cout << setw(10);`
  könnte auf folgende Weise ein temporäres Objekt geschaffen werden:

  ```
  ios& width(ios& ros, int n) {
    ros.width(n);
    return ros;
  }
  smanip<int> setw(int n) {
    return smanip<int>(&width, n);
  }
  ```

  in diesem Beispiel ist das temporäre Objekt vom Typ `smanip<int>`, ein instanziiertes Template, wobei der Typparameter durch den Typ des Manipulatorargumentes ersetzt wurde

- das `smanip`-Template könnte folgendermassen definiert werden:

  ```
  template <class TP> class smanip {
    ios& (*_f)(ios&, TP);
    TP _a;
  ```

```
public:
  smanip(ios& (*f)(ios&, TP), TP a) : _f(f), _a(a) {}

friend istream& operator>>(istream& i, const smanip<TP>& m);
friend ostream& operator<<(ostream& o, const smanip<TP>& m);
};

template<class TP>
inline istream& operator>>(istream& i, const smanip<TP>& m)
{ (*m._f)(i, m._a); return i; }

template<class TP>
inline ostream& operator<<(ostream& o, const smanip<TP>& m)
{ (*m._f)(o, m._a); return o;}
```

Objekte vom Typ `smanip`<T> speichern somit einen Zeiger auf eine Funktion, die die eigentliche Manipulation ausführt sowie das zugehörige Manipulationsargument vom Typ `T`. Die für den jeweiligen instanziierten `smanip`-Typ überladene `operator<<`-Funktion bringt dann die im temporären `smanip`-Objekt gespeicherte Funktion mit dem Manipulatorargument zur Ausführung.

- im Header <iomanip.h> werden solche Templates für die Manipulation von `istream`'s (`imanip`), von `ostream`'s (`omanip`) und allgemein für Objekte der Klasse `ios` (`smanip`) definiert
- folgende Manipulatoren mit einem Argument werden bereits in <iomanip.h> erzeugt:

```
smanip<int> setbase(int);
smanip<int> setfill(int);
smanip<int> setprecision(int);
smanip<int> setw(int);

smanip<ios::fmtflags> resetiosflags(ios::fmtflags);
smanip<ios::fmtflags> setiosflags(ios::fmtflags);
```

Praktikumsaufgabe Nr. 6:
Aus dem Kapitel 1.2.1 auf Seite 13 liegen Definitionen für die Klassen `Name` *und* `NamemitTitel` *vor. In diesen Klassen ist eine virtuelle Methode* `printOn` *definiert.*
Definieren Sie nun bitte auf der Basis dieser Methode eine Manipulatorfunktion `PrintName` *zur Ausgabe von* `Name`*-Objekte auf Manipulatorart!*
Überlegung:
Muß die Technik des Manipulators ohne Argument oder die des Manipulators mit Argument angewendet werden?
Hinweis:
Zur konkreten Ausgabe eines `Name`*-Objektes soll durch die Manipulatorsyntax der indirekte Aufruf der printOn-Methode realisiert werden.*

Folgende Verwendung:

```
// File: mManName.cc

#include <NamTitel.h>
#include <PROTECT/ManName.h>

int main()
{
  Name *pDerStinomann = new Name("Otto", "Normal");
  Name *pDerStinoHerr = new NamemitTitel("Herr", *pDerStinomann);
  NamemitTitel *pDieStinoFrau
                = new NamemitTitel("Frau", "Lotte", "Normal");

  cout << PrintName(*pDerStinomann) << endl
       << PrintName(*pDerStinoHerr) << endl
       << PrintName(*pDieStinoFrau) << endl;

  Name *pAnOtherStino = new NamemitTitel(*pDieStinoFrau);

  delete pDerStinoHerr;

  cout << PrintName(*pAnOtherStino) << endl;
  return 0;
}
```

Weisen Sie auch das polymorphe Verhalten des Manipulators (bzw. der printOn-Methode) in Bezug auf Objekte verschiedener `Name`*-Klassen (z.B.* `NamemitTitel`*) nach!*
Hinweis:
Im Prinzip brauchen Sie nur einen weiteren Header-File `PROTECT/ManName.h` *zu schreiben; alle anderen Files können Sie übernehmen, compilieren und linken.*
Output des Programmes:

```
Otto Normal
Herr Otto Normal
Frau Lotte Normal
Frau Lotte Normal
```

7.2 Persistenz von Klassen- bzw. Objekthierarchien

Es ist oft eine Grundnotwendigkeit, aus einem laufenden Programm heraus die existierenden Objektstrukturen teilweise oder vollständig zu sichern. Als persistente Objekte bezeichnet man daher solche Objekte, die in einer gewissen Form auf einem persistenten Medium (z.B. auf einem Massenspeicher) existieren und damit den Programmlauf überleben können. Als Kriterium für die Form gilt, daß sie das Reaktivieren eines gespeicherten Objektes in einen aktuellen Programmlauf hinein möglichst typsicher gestatten muß. Als *Objektpersistenz* wird

oft das Konzept und die Implementation der Fähigkeit persistenter Objekte in einem System bezeichnet.
Verschiedene Grundkonzepte sind dazu denkbar:

1. **Nutzung des stringorientierten Streaming:**
   Klassen definieren Streaming-Operatoren (`<<` und `>>`) und sorgen selbst für die Aus-/Eingabe der Typ- und Versionsinformationen, eigener Datenelemente, Daten aggregierter Objekte und Daten der eigenen Subobjekte (Basisklassen!).
   Vorteile:
   - einmal definierte Streaming-Operatoren arbeiten i.d.R. für alle gängigen Streamobjekte, d.h. Objekte können auch z.B. in andere Speicherbereiche ausgelagert oder auf Netzverbindungen gesendet werden
   - die Stringdarstellung ist i.d.R. portabel

   Nachteile:
   - Speicherplatz für nicht binären Darstellung
   - Genauigkeit von Gleitkommazahlen
   - Objekte wissen i.d.R. „nicht viel“ (Mindesttyp) über die Typinformation der indirekt aggregierten Objekte (Container!)
     $\Rightarrow$ Problem der Rekonstruktion
2. ***Hardcopy***
   Übergabe der Adresse und Größe eines Objektes an ein „I/O-Objekt“, welches dann eine Anzahl Bytes (Größe des Objektes) von der gegebenen Adresse liest und auf das verbundene Medium schreibt.
   Vorteile:
   - schnelle und rel. sichere Methode
   - geringer Speicherplatz

   Nachteile:
   - nicht portabel (Zahlendarstellung, Alignments)
   - Typinformation muß i.d.R. zusätzlich gespeichert werden
   - enthaltene Pointer sind nach dem Einlesen ungültig, also auch die Pointer auf aggregierte Objekte
     $\Rightarrow$ Problem der Rekonstruktion
3. **Abbildung der Objekte auf Datenbank**
   sehr viele und verschiedenartige Konzepte, Grundunterscheidung:
   - relationales Datenbankmodell
     Nachteile:
     - zwei verschiedene Modelle für Programm und Datenbank
     - evtl. aufwendige Abbildung der Objekte zu Tabellen und Datensätzen

- objektorientiertes Datenbankmodell (z.B. *Poet*)
  - 1:1 - Abbildung des C++-Modells in Datenbank
  - Datenbankoperationen auf Objektlevel
  - ist im Prinzip *Objektpersistenz*

Jedes Konzept aus dieser Aufzählung hat eigentlich folgendes Problem zu lösen: Wie kann die Objekthierarchie, also die zu einem Objekt direkt oder indirekt aggregierten weiteren Objekte rekonstruiert werden?
Wir wollen dazu vorerst eine *naive* prototypische Lösung für unser `GeoObj`-Beispiel entwickeln und setzen der Einfachheit und Transparenz wegen auf das Grundkonzept 1 auf.

- wir erweitern die abstrakte `GeoObj`-Klasse um die Schnittstelle für das Streaming:

```
// File: NaivGeoO.h

#ifndef GeoObj_h
#define GeoObj_h 1

#include <iostream.h>
#include <String.h>

class GeoObj
{
// ...
public:
// ...
  virtual void WriteOn(ostream&) const = 0;
  virtual void ReadFrom(istream&) = 0;
// ...
};

#endif
```

- die Streaming-Operatoren nutzen die jeweils in den konkreten Klassen spezialisiert definierten (in `GeoObj` allerdings abstrakten) virtuellen Methoden `ReadFrom` und `WriteOn`; vor dem Schreiben des eigentlichen Objektes wird mittels der virtuellen Methode `GetTypeString` eine Typbeschreibung von diesem abgefragt und auf den Stream geschrieben:

```
// File: NaivGeoO.cc

#include <NaivGeoO.h>
// ...
ostream& operator<<(ostream& ros, const GeoObj& gos)
{
```

```
  ros << gos.GetTypeString();
  gos.WriteOn(ros);
  return ros;
}
// ...
```

- umgekehrt muß vor dem Einlesen des Objektes vom Stream dessen Typbeschreibung gelesen werden; hier kann schon ein Vergleich mit dem Typ des aktuellen Zielobjektes (das Objekt, in welches eingelesen werden soll) Typunstimmigkeiten verhindern helfen; im Falle der Nichtübereinstimmung wird im `istream`-Objekt ein `ios::failbit`-Status gesetzt, der jede weitere Leseoperation von diesem verhindert:

```
// ...
istream& operator>>(istream& ris, GeoObj& gis)
{
  if(ris)
  {
    String TypeString;
    ris >> TypeString;

    if(TypeString == gis.GetTypeString())
      gis.ReadFrom(ris);
    else
    {
      cerr << "Error in GeoObj operator>>:"<< endl
           << "  read a " << TypeString << endl
           << "  but need a " << gis.GetTypeString() << endl;
      ris.setstate(ios::failbit);
    }
  }
  return ris;
}
```

- wir versuchen jetzt, die Methoden `ReadFrom` und `WriteOn` für die verschiedenen Klassen zu implementieren:

- Klasse `Rechteck`:

```
// File: NaivRctk.cc

#include <math.h>
#include <NaivRctk.h>
// ...
void Rechteck::WriteOn(ostream& ros) const
{
  ros << " " << EckPunkt[0]
```

```
      << " " << EckPunkt[1];
}

void Rechteck::ReadFrom(istream& ris)
{
  ris >> EckPunkt[0]
      >> EckPunkt[1];
}

String Rechteck::GetTypeString() const
{
  return "[Rechteck]";
}
```

- Klasse `RechteckRund`:

```
// File: NaivRtRn.cc

#include <NaivRtRn.h>

const float RechteckRund::PI = 3.14159265389;
// ...
void RechteckRund::WriteOn(ostream& ros) const
{
  Rechteck::WriteOn(ros);
  ros << " " << Radius;
}

void RechteckRund::ReadFrom(istream& ris)
{
  Rechteck::ReadFrom(ris);
  ris >> Radius;
}

String RechteckRund::GetTypeString() const
{
  return "[RechteckRund]";
}
```

- zwei kleine Programme sollen zeigen, daß das Streaming bis hierher funktioniert:

```
// File: oNaiv.cc

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

```
#include <NaivRtRn.h>

int main()
{
  Rechteck r1(Punkt(0., 0.), Punkt(6., 5.));
  RechteckRund r2(Punkt(6., 0.), Punkt(15., 5.));

  cout << r1 << endl;
  cout << r2 << endl;
}
```

```
// File: iNaiv.cc

#include <iostream.h>
#include <PunktV.h>
#include <NaivRctk.h>
#include <NaivRtRn.h>

int main()
{
  Rechteck r1;
  RechteckRund r2;

  cin >> r1;
  cin >> r2;

  cerr << "Flaeche r1: " << r1.flaeche() << endl;
  cerr << "Flaeche r2: " << r2.flaeche() << endl;
}
```

- `oNaiv` gibt die beiden Objekte in Ihrer textuellen Beschreibung auf den Standardausgabe-Stream; `iNaiv` liest diese vom Standardeingabe-Stream und berechnet die Flächen mittels der `flaeche`-Methode

- wenn Ihnen Pipes zur Verfügung stehen, dann können Sie die beiden Programme verknüpft ausführen:

```
% ./oNaiv | ./iNaiv
Flaeche r1: 30
Flaeche r2: 45
```

- wir implementieren jetzt die Methoden `ReadFrom` und `WriteOn` von `ZGeoObj`; die `WriteOn`-Methode ist noch recht einfach; bitte beachten Sie, daß Nullzeiger im Array ausgelassen und die korrekte Zahl gültiger aggregierter Objekte `nCount` auf den Stream geschrieben wird:

```
// File: NaivZGeo.cc
```

```
#include <NaivZGeo.h>
#include <NaivRctk.h>
#include <NaivRtRn.h>
// ...
void ZGeoObj::WriteOn(ostream& ros) const
{
// Compacting...
  int i, nCount = 0;
  for(i = 0; i < Count; i++)
    if(GeoObjs[i])
      nCount++;

  ros << " " << nCount << endl;
  for(i = 0; i < Count; i++)
    if(GeoObjs[i])
    {
      ros << GeoObjs[i]->GetTypeString();
      GeoObjs[i]->WriteOn(ros);
      ros << endl;
    }
}
```

- die `ReadFrom`-Methode bringt einige Probleme:
  hier müssen anhand der gelesenen Typbeschreibungen entsprechende Objekte zum Aggregieren erzeugt werden; eine `if-else`-Kette und Benennung konkreter Klassen ist unumgänglich:

```
// File: NaivZGeo.cc
// ...
void ZGeoObj::ReadFrom(istream& ris) {
  String TypeString;
  GeoObj *NewGeoObj;
  int nCount = 0;

  ris >> nCount;
  for(int i = 0; (i < nCount) && (ris); i++) {
    ris >> TypeString;
    if(TypeString == "[Rechteck]") {
      NewGeoObj = new Rechteck;
      NewGeoObj->ReadFrom(ris);
      Add(*NewGeoObj);
    } else
    if(TypeString == "[RechteckRund]") {
      // ... siehe ZGeoObj
    } else
    if(TypeString == "[ZGeoObj]") {
      NewGeoObj = new ZGeoObj;
```

```
        NewGeoObj->ReadFrom(ris);
        Add(*NewGeoObj);
      } else
      {
        cerr << "Error in GeoObj factory:"<< endl
             << "  read a " << TypeString << endl
             << "  but don't know how to make the object" << endl;
        ris.setstate(ios::failbit);
      }
    }
  }
```

Das ist ein schwerwiegender Design-Fehler! Warum?

1. die `ZGeoObj`-Klasse war so konzipiert, nur die `GeoObj`-Schnittstelle seiner aggregierten Objekte zu kennen; nun müssen alle Klassen der zu verwendenden `GeoObjs` den Methoden der Klasse `ZGeoObj` bekannt gemacht werden, d.h. es müssen in `NaivZGeo.cc` die Headerfiles mit den Definitionen aller zu verwendenden Klassen includiert werden

2. beispielsweise kennt ein `ZGeoObj` im Augenblick noch nicht die Klasse `Dreieck`; es wird also das Einlesen und Konstruieren eines `Dreiecks` ablehnen (Beispiele `oNaivZGe/iNaivZGe`: oNaivZGe erzeugt ein Dreieck, welches von iNaivZGe nicht verarbeitet werden kann; `iNaivZGe` setzt den Fehler-Status im `cin` und gibt die Fläche der bis dahin gelesenen Elemente aus):

   ```
   ./oNaivZGe  | ./iNaivZGe
   Error in GeoObj factory:
     read a [Dreieck]
     but don't know how to make the object
   Flaeche z1: 120
   ```

3. der Entwurf ist dahingehend unflexibel geworden, daß neue Klassen nicht nur definiert, sondern auch in `ZGeoObj` erfaßt werden müssen; das kann Recompilieren einer ganzen Klassenhierarchie bedeuten!

4. was passiert mit den dynamisch konstruierten Objekten? das `ZGeoObj` war bezüglich der Verantwortlichkeit so konzipiert, die enthaltenen Pointer nicht automatisch (z.B. zum Ende der Lebensdauer) zu löschen (um z.B. auch Adressen automatischer Variablen speichern zu können; außerdem der Einfachheit wegen)

- zur Lösung des Problems werden wir die Erzeuger-Patterns genauer studieren müssen; siehe folgendes Kapitel

8 Objektpersistenz und Erzeuger-Pattern

- Erzeuger-Pattern dienen meistens dazu, die Erzeugung von Objekten dynamischer und flexibler zu gestalten; einige dienen der Einschränkung vorhandener Möglichkeiten der freien Erzeugung (z.B. Singleton)

- die wesentlichen Problempunkte unseres Entwurfes waren:

  1. Benennung konkreter Klassen in der Implementation der Klasse `ZGeoObj`, die ihrerseits nur sich selbst und die Schnittstelle ihrer abstrakten Basisklasse `GeoObj` kennen sollte

  2. `if-else`-Kette, die für jede neue Klasse erweitert werden muß

- betrachten wir zur Lösung unseres Problems zunächst einmal die Kurzfassung der Beschreibung zum abstract factory-Pattern:

Name des Pattern: Abstract Factory (Quelle: Gamma ([11])

Zweck
Bietet eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander abhängiger Objekte, ohne ihre konkreten Klassen zu benennen.

Anwendbarkeit
Verwenden Sie das Abstrakte-Fabrik-Muster, wenn

1. ein System unabhängig davon sein soll, wie seine Produkte erzeugt, zusammengesetzt und repräsentiert werden.

2. ein System mit einer von mehreren Produktfamilien konfiguriert werden soll.

3. eine Familie von verwandten Produktobjekten entworfen wurde, zusammen verwendet zu werden, und Sie diese Konsistenzbedingung sicherstellen müssen.

4. Sie eine Klassenbibliothek von Produkten anbieten möchten, von denen Sie nur die Schnittstellen, nicht aber ihre Implementierungen offenlegen möchten.

Struktur und Teilnehmer

- für das `ZGeoObj`-Beispiel treffen vor allem die Punkte 1 und 4 zu; die anderen wären aber für `GeoObjs` mit systemabhängigem Outfit auch denkbar
- wir definieren uns eine Klasse `AbstractFactory` in Anlehnung an das gleichnamige Pattern, welche eine Methode zum Liefern neuer Objekte vom Typ `GeoObj` hat
- da unsere Fabrik gleichzeitig die Verantwortung für die Pointer, die sie erzeugt, behalten soll, verfügt sie über eine einfache Funktionalität zum Management dieser (Array von Pointern: `GeoObj* GeoObjs[MAX]`, Methode zum Sichern eines Pointers: `void _SaveGeoObj(GeoObj*)`)
- wir definieren wir eine weitere Methode `CollectGeoObj()` zum Zurückgeben eines Pointers an die Fabrik
- zum Ende der Lebenszeit eines Objektes einer von `AbstractFactory` abgeleiteten Klasse sollen alle über Pointer referenzierten `GeoObj`-Objekte gelöscht werden; dazu muß erstens der Destruktor von `AbstractFactory` und zweitens der von `GeoObj` als `virtual` deklariert sein:

```
// File: AbstFcry.h

#ifndef AbstractFactory_h
#define AbstractFactory_h 1

class GeoObj;

class AbstractFactory
{
private:
  enum{MAX = 1000};
  int Count;
  GeoObj* GeoObjs[MAX];
  AbstractFactory(const AbstractFactory&); // not allowed
  AbstractFactory& operator=(const AbstractFactory&);

protected:
  void _SaveGeoObj(GeoObj*);

public:
  AbstractFactory();
  virtual ~AbstractFactory();
  virtual GeoObj* CreateGeoObj() = 0;
  virtual operator void*() const = 0;      // can create more objects?
  virtual void CollectGeoObj(GeoObj*&);
};

#endif
```

- die einzigen *pure virtual functions* sind `CreateGeoObj()` (Festlegung, wie `GeoObjs` erzeugt werden sollen, erst in konkreter Fabrik) und der Konvertierungsoperator zum Erzeugen eines integralen Types aus einer Fabrik (zum Einsatz in logischen Operatoren zur Feststellung, ob eine Fabrik aktuell fähig ist, `GeoObjs` zu „produzieren“)

- alle weiteren Methoden werden zu diesem Zeitpunkt implementiert:

```
// File: AbstFcry.cc

#include <AbstFcry.h>
#include <GeoObj.h>

AbstractFactory::AbstractFactory()
: Count(0)
{
  for(int i = 0; i < MAX; i++)
    GeoObjs[i] = NULL;
}

AbstractFactory::~AbstractFactory()
{
  for(int i = 0; i < Count; i++)
  {
    if(GeoObjs[i])
    {
      cerr << "Deleting a "
           << GeoObjs[i]->GetTypeString() << " object." << endl;
      delete GeoObjs[i];
    }
  }
}

void AbstractFactory::_SaveGeoObj(GeoObj* pg)
{
  if(Count < MAX)
    GeoObjs[Count++] = pg;
}

void AbstractFactory::CollectGeoObj(GeoObj*& pg)
{
  for(int i = 0; i < Count; i++)
    if((pg) && (GeoObjs[i] == pg))
    {
      delete GeoObjs[i];
      GeoObjs[i] = NULL;
      pg = NULL;
```

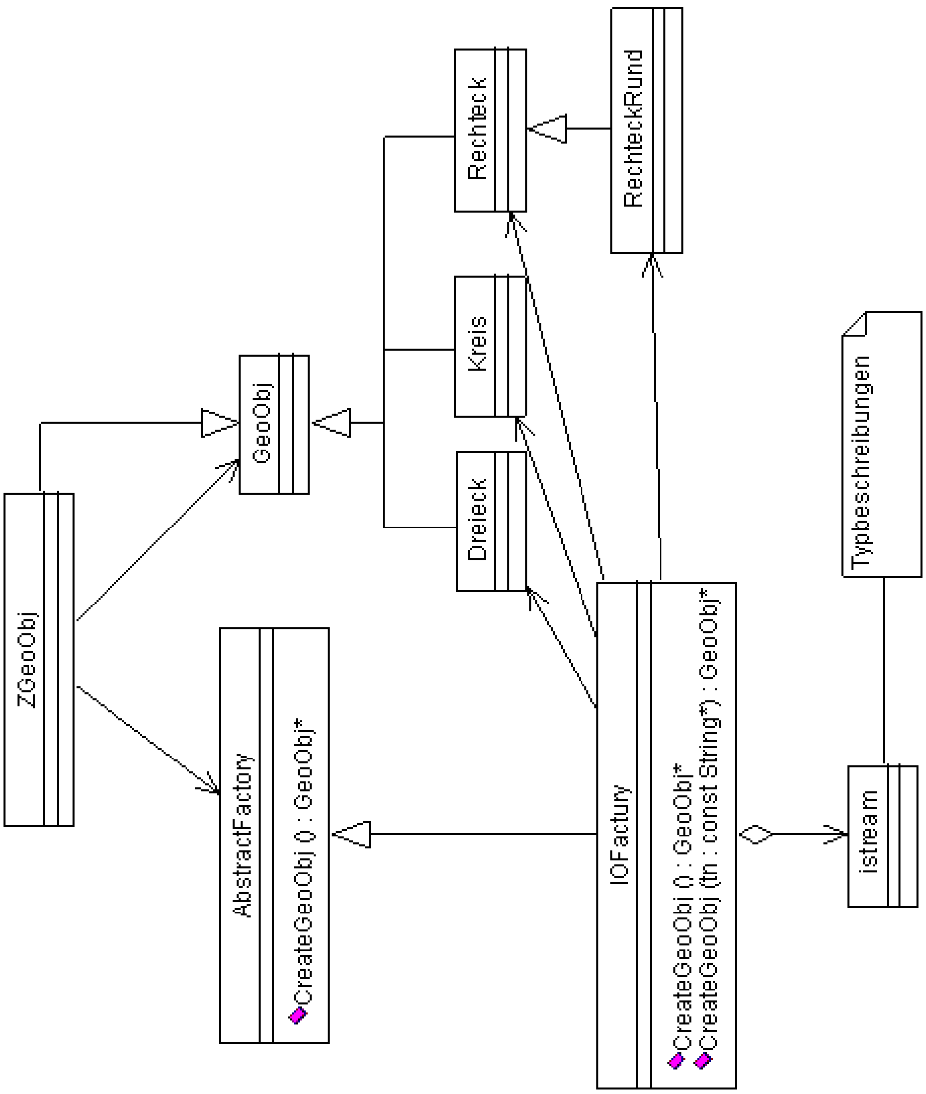
```
        }
}
```

- beachten Sie die syntaktische Raffinesse des Argumentes an die Methode `void AbstractFactory::CollectGeoObj(GeoObj*& pg)` das Argument ist vom Typ her eine Referenz auf einen Pointer; nach dem Löschen des verbundenen Objektes wird an dieses Argument eine `NULL` zugewiesen, um dem aufrufenden Kontext zu verdeutlichen, daß die Fabrik die vollständige Kontrolle über den Pointer übernommen hat und das assoziierte Objekt gelöscht oder der Speicherbereich zum Löschen freigegeben wurde

- Bemerkungen zum Speichermanagement:
  - die Fabrik muß den Speicher des assoziierten Objektes nicht sofort freigeben; i.d.R. reicht auch der explizite Aufruf des Destruktors und Einordnung des Speicherblockes in einen geeigneten Container, der z.B. einen schnellen Zugriff darauf anhand der assoziizerten Größe des Speicherblockes gestattet
  - da eine konkrete Fabrik für das Erzeugen einer ganz bestimmten Produktpalette (`GeoObjs`) zuständig ist, wird die Anzahl der verschiedenen Größen der von dieser dynamisch anzufordernden Speicherblöcke der Anzahl der konkreten Produkttypen entsprechen; also stark begrenzt sein
  - es kann also u.U. wesentlich effizienter für eine konkrete Fabrik sein, beim Erzeugen eines neuen Objektes für die Allokation dessen Speicherbereiches anhand dessen Größe eine Wiederbenutzung eines zum Löschen freigegeben, aber aktuell noch nicht gelöschten Speicherblockes zu versuchen
  - hierzu sollte nicht die globalen Operatoren `new` und `delete` überschrieben, sondern klassenlokale Operatoren definiert werden; auch ist die Anwendung der *placement new*-Syntax (`new`-Operator mit einem Argument) denkbar
  - wenn das Programm in eine *Low-Memory*-Situation eintritt, sollte die Fabrik angewiesen werden, alle überflüssigen Speicherblöcke zu löschen

- wir wollen vor der weiteren Entwicklung Überlegungen zur Zuordnung der Elemente unseres bisherigen Entwurfes zu den Elementen der Patternstruktur anstellen:
  - Abstract Factory ist die Klasse `AbstractFactory`
  - Abstract Product ist die Klasse `GeoObj`
  - Concrete Factory ist die Fabrik, die uns neue Objekte anhand der Daten in einem Stream liefern, oder Objekte beim Zurückgeben auf einen Stream speichern soll; wir benennen die Klasse entsprechend mit `IOFactory`

– Concrete Product ist ein Objekt einer von `GeoObj` abgeleiteten, konkreten Klasse
– Client der Abstract Factory ist jede Instanz, die die Methoden `CreateGeoObj()` oder `CollectGeoObj()` aufruft; das kann z.B. die `main()`-Funktion oder aber auch die `ReadFrom()`-Methode der Klasse `ZGeoObj` sein



- es wird also mindestens die `ZGeoObj`-Klasse sein, die wir explizit mit einem Zeiger auf ein Objekt der Klasse `AbstractFactory` werden ausstatten müssen, da wir uns aber vorerst auf dem Niveau der abstrakten Klassen bewegen wollen, werden wir den Pointer als Datenelement der abstrakten Klasse `GeoObj` zuordnen:

```
// File: GeoObj.h

#ifndef GeoObj_h
#define GeoObj_h 1

#include <iostream.h>
#include <String.h>

class AbstractFactory;

class GeoObj
{
private:
  AbstractFactory* pFactory;
// ...

public:
  GeoObj(AbstractFactory* _pFactory = NULL)
  : pFactory(_pFactory){}
  virtual ~GeoObj(){};
// ...
  AbstractFactory* SetFactory(AbstractFactory*);
  AbstractFactory* GetFactory();
};

inline AbstractFactory* GeoObj::SetFactory(AbstractFactory* _pFactory)
{
  AbstractFactory* pOld = pFactory;
  pFactory = _pFactory;
  return pOld;
}

inline AbstractFactory* GeoObj::GetFactory()
{
  return pFactory;
}

#endif
```

- beachten Sie bitte, daß die Klasse `GeoObj` jetzt ein Datenelement besitzt, welches beim Kopieren und Zuweisen von Objekten davon abgeleiteter Klassen unbedingt mit Beachtung finden muß:

```
ZGeoObj::ZGeoObj(const ZGeoObj &right)
: GeoObj(right)
{
// ...
```

}

- beachten Sie bitte weiterhin, daß wir das Handling der TypeStrings in Zukunft der Fabrik überlassen wollen; die Implementation der Streaming-Operatoren von `GeoObj` vereinfachen sich daher wesentlich:

```
// File: GeoObj.cc

#include <GeoObj.h>
#include <AbstFcry.h>
// ...
ostream& operator<<(ostream& ros, const GeoObj& gos)
{
  gos.WriteOn(ros);
  return ros;
}

istream& operator>>(istream& ris, GeoObj& gis)
{
  if(ris)
    gis.ReadFrom(ris);
  return ris;
}
```

- wir kommen nun zur Definition der CONCRETE FACTORY, der `IOFactory`-Klasse:
  - eine `IOFactory` implementiert die virtuellen Methoden, die durch die Schnittstelle der `AbstractFactory` vorgegeben wird
  - eine `IOFactory` hat einen `istream` zum kontinuierlichen Lesen der Typinformationen der zu erzeugenden Objekte und einen `ostream` zum finalen Ausgaben des kompletten Inhaltes des `GeoObjs` bei der `CollectGeoObj`-Operation; die Verantwortung zum Löschen der assoziierten Streaming-Objekte zum Ende der Lebenszeit der `IOFactory` soll in der IOFactory durch eine interne Variable
    `bool deleteStreams;`
    angezeigt werden
  - die `IOFactory` bekommt eine weitere `CreateGeoObj()`-Methode, die eine Typbeschreibung als Argument nimmt und das passende Produkt erzeugen soll:

```
// File: IOFactry.h

#ifndef IOFactory_h
#define IOFactory_h 1
```

```
#include <stl.h>
#include <String.h>
#include <bool.h>
#include <fstream.h>
#include <AbstFcry.h>

class IOFactory : public AbstractFactory
{
private:
// ...
  bool deleteStreams;
  istream* pIStream;
  ostream* pOStream;
  IOFactory(const IOFactory&); // not allowed!
  IOFactory& operator=(const IOFactory&);

public:
// ...
  virtual ~IOFactory();
  virtual GeoObj* CreateGeoObj();
  virtual GeoObj* CreateGeoObj(const String&);
  virtual operator void*() const;
  virtual void CollectGeoObj(GeoObj*&);
};

#endif
```

- zur Vermeidung der Benennung konkreter Klassen in der Implementation der `IOFactory` führen wir ein weiteres Pattern ein: Prototype:

Name des Pattern: Prototype (Quelle: Gamma ([11])

Zweck
Bestimme die Arten zu erzeugender Objekte durch die Verwendung eines prototypischen Exemplars und erzeuge neue Objekte durch Kopieren dieses Prototypen.

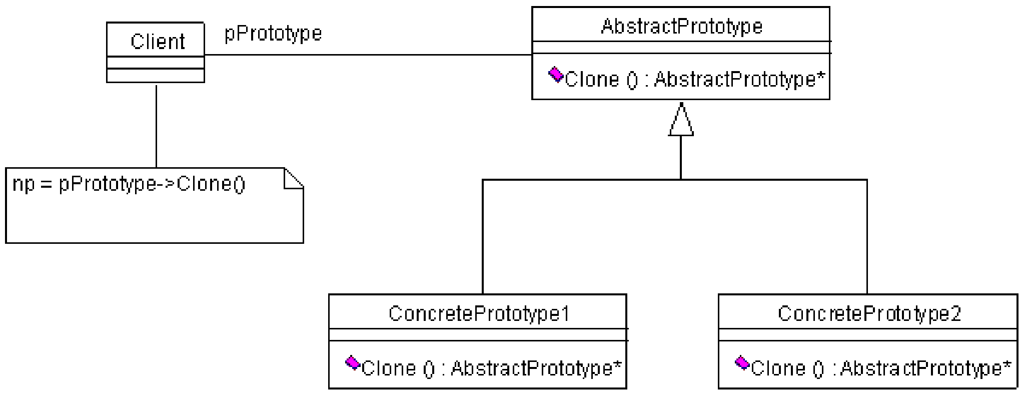
Anwendbarkeit
Verwenden Sie das Prototypmuster, wenn ein System unabhängig davon sein soll, wie seine Produkte erzeugt, zusammengesetzt und repräsentiert werden, und

- wenn die Klassen zu erzeugender Objekte erst zur Laufzeit spezifiziert werden, beispielsweise durch dynamisches Laden, oder

- um zu vermeiden, eine Klassenhierarchie von Fabriken zu erstellen, die parallel zur Klassenhierarchie der Produkte verläuft, oder

- wenn Exemplare einer Klasse nur wenige unterschiedliche Zustandskombinationen haben können. Es ist möglicherweise bequemer, eine entsprechende Anzahl von Prototypen einzurichten und sie zu klonen statt die Objekte einer Klasse jedesmal von Hand mit dem richtigen Zustand zu erzeugen.

Struktur und Teilnehmer



- das ist interessant! wir werden für alle konkreten `GeoObjs` Prototypen bereitstellen, die zum Ereugen neuer Objekte einfach zu klonen sind
- dabei gibt es noch einen weiteren positiven Effekt:
  wir initialisieren den Prototypen für Objekte vom Typ `ZGeoObj` mit der Adresse der zu verwendenden Fabrik; demzufolge erhalten alle neuen `ZGeoObj`-Objekte bereits automatisch den korrekten Zeiger auf diese gespeichert!
- nun aber vorerst zur Verwaltung der Prototypen:
  - die Prototypen müssen mit den textuellen Typbeschreibungen assoziiert werden; d.h. wir müssen eine Abbildung der Menge von Typbeschreibungen auf die Menge der Prototypen schaffen
  - für diese Abbildung ist der `map`-Typ aus der Standard-Template-Library geeignet
  - wir instanziieren eine Klasse des Templates wie folgt:

```
typedef less<String> StringCompare;
typedef map<String, GeoObj*, StringCompare> MapOfPrototypes;
```

  - wir assoziieren die `IOFactory` mit einem solchen Container
  - die Adresse eines Objektes einer auf diese Art instanziierten Containerklasse wird zur Initialisierung des `IOFactory`-Objektes im Konstruktor übergeben:

```
// File: IOFactry.h

#ifndef IOFactory_h
#define IOFactory_h 1

#include <stl.h>
#include <String.h>
#include <bool.h>
#include <fstream.h>
#include <AbstFcry.h>

typedef less<String> StringCompare;
typedef map<String, GeoObj*, StringCompare> MapOfPrototypes;

class IOFactory : public AbstractFactory
{
private:
  MapOfPrototypes* pPrototypes;
// ...
public:
  IOFactory(MapOfPrototypes*);
  IOFactory(MapOfPrototypes*, istream*, ostream* = NULL);
  IOFactory(MapOfPrototypes*, const String&,
                              const String& = String(""));
// ...
};

#endif
```

- es wurden verschiedene Konstruktoren vorgesehen; entsprechend der verschiedenen Möglichkeiten, eine `IOFactory` mit fertig konstruierten Streams zu verbinden oder Namen von Files zu übergeben, zu denen sich das `IOFactory`-Objekt selbst die Verbindungen (die Stream-Objekte) erzeugen soll (entsprechend `deleteStreams` setzen!)
- bezüglich der Implementation der Methoden sind die beiden `CreateGeoObj()`-Methoden besonders interessant:

```
// File: IOFactry.cc

#include <IOFactry.h>
#include <GeoObj.h>

// ...
GeoObj* IOFactory::CreateGeoObj()
{
  if(pIStream && (*pIStream))
  {
```

```
    String TypeString;
    (*pIStream) >> TypeString;

    if(TypeString != "")
      return CreateGeoObj(TypeString);
    else
      return NULL;
  }
  else
  {
    cerr << "Have no istream; returning NULL" << endl;
  }

  return NULL;
}

GeoObj* IOFactory::CreateGeoObj(const String& TypeString)
{
  GeoObj* pProto = (*pPrototypes)[TypeString];
  GeoObj* pClone = NULL;

  if(pProto)
    pClone = pProto->Clone();
  else
  {
    cerr << "Error in GeoObj factory:"<< endl
         << "  could not find prototype for a "
         << TypeString << endl;
    return NULL;
  }

  if(pClone)
    _SaveGeoObj(pClone);
  else
  {
    cerr << "Error in GeoObj factory:"<< endl
         << "  could not clone prototype for a "
         << TypeString << endl;
    return NULL;
  }

  return pClone;
}
// ...
```

- Funktion:

- die erste `CreateGeoObj()`-Methode (ohne Argument) muß sich die Typbeschreibung des zu erzeugenden Objektes besorgen
- sie nutzt den assoziierten `istream`, um Typinformationen zu lesen
- nach dem Lesen der Typbeschreibung übergibt sie diese an die zweite Methode als Argument
- die zweite Methode wird im durch `pPrototypes` referenzierten `map`-Objekt nach dem assoziierten Prototypen suchen
- wird ein solcher Prototyp gefunden, dann wird er geklont
- die Adresse des neuen Objektes wird im eigenen Array gesichert und als Returnwert der ersten Methode übergeben

- welche Veränderungen ergeben sich durch das neue Design an unseren `GeoObj`-Klassen?
  - nicht viele
  - die Schnittstelle der abstrakte Klasse muß um die `Clone()`-Methode erweitert werden:

```
// File: GeoObj.h
// ...
class AbstractFactory;

class GeoObj
{
public:
  GeoObj(AbstractFactory* _pFactory = NULL)
  : pFactory(_pFactory){}
// ...
  virtual GeoObj* Clone() const = 0;
// ...
};

#endif
```

  - diese Methode muß für alle konkreten `GeoObj`-Klassen implementiert werden, hier ein Beispiel:

```
// File: ZGeoObj.cc

#include <ZGeoObj.h>

ZGeoObj::ZGeoObj(AbstractFactory* _pFactory)
: GeoObj(_pFactory), Count(0)
{
  for(int i = 0; i < MAX; i++)
    GeoObjs[i] = NULL;
```

```
}
// ...
GeoObj* ZGeoObj::Clone() const
{
  return new ZGeoObj(*this);
}
// ...
```

  – eine wesentliche Vereinfachung erfährt die Methode `ReadFrom` der `ZGeoObj`-Klasse; die `WriteOn` Methode bleibt bestehen

Praktikumsaufgabe Nr. 7:
Implementieren Sie bitte basierend auf dem neuen Design die Methode `void ZGeoObj::ReadFrom(istream& ris)`*! Lassen Sie sich die einem* `ZGeoObj` *assoziierten* `GeoObjs` *durch die Fabrik* `GetFactory()` *erzeugen! Testen Sie bitte den Zeiger auf die* `AbstractFactory` *auf Gültigkeit! Rufen Sie zur Initialisierung der neuen Objekte wiederum deren* `ReadFrom()`*-Methode auf!*
Hinweis:
Verwenden Sie bitte als Vorlage das im Public-Bereich befindliche File `ZGeoObj.cc`.

- welche Flexibilität uns das neue Konzept bietet, das soll nun anhand der nachfolgenden kleinen Beispiele gezeigt werden:

  1. Test der `IOFactory`:
     - Verbinden des `IOFactory`-Objektes mit `cin`
     - Erzeugen von Prototypen;
       beachten Sie:
       der Prototyp für `ZGeoObj` wird mit der Adresse des `IOFactory`-Objektes initialisiert
     - dynamisches Erzeugen neuer Objekte anhand der von der Standardeingabe gelesenen Typbeschreibungen, (Ende: Ctrl-D bzw. Ctrl-Z)
       Muster für Typbeschreibung: `[Dreieck]`
     - Testen Sie bitte auch unbekannte Typen, um die Reaktion des Systems zu erfahren!

```
// File: mFctTstC.cc

#include <iostream.h>
#include <IOFactry.h>
#include <PunktV.h>
#include <Rechteck.h>
#include <RchtckRn.h>
#include <Dreieck.h>
#include <Kreis.h>
#include <ZGeoObj.h>

```

```
int main()
{
  MapOfPrototypes Prototypes;
  IOFactory Factory(&Prototypes, &cin);
  Prototypes["[Rechteck]"] = new Rechteck;
  Prototypes["[RechteckRund]"] = new RechteckRund;
  Prototypes["[Dreieck]"] = new Dreieck;
  Prototypes["[Kreis]"] = new Kreis;
  Prototypes["[ZGeoObj]"] = new ZGeoObj(&Factory);

  while(Factory)
  {
    GeoObj *pNewGeoObj = Factory.CreateGeoObj();
    if(pNewGeoObj)
      cout << pNewGeoObj->GetTypeString()
           << " created." << endl;
  }
}
```

2. Test der `IOFactory`:
   - siehe oben; Unterschied: statt eines Streams wird der Filename übergeben

```
// File: mFctTstF.cc

#include <iostream.h>
#include <IOFactry.h>
#include <PunktV.h>
#include <Rechteck.h>
#include <RchtckRn.h>
#include <Dreieck.h>
#include <Kreis.h>
#include <ZGeoObj.h>

int main()
{
  MapOfPrototypes Prototypes;
  IOFactory Factory(&Prototypes, "mFctTstF.in");
  Prototypes["[Rechteck]"] = new Rechteck;
  Prototypes["[RechteckRund]"] = new RechteckRund;
  Prototypes["[Dreieck]"] = new Dreieck;
  Prototypes["[Kreis]"] = new Kreis;
  Prototypes["[ZGeoObj]"] = new ZGeoObj(&Factory);

  while(Factory)
  {
    GeoObj *pNewGeoObj = Factory.CreateGeoObj();
```

```
      if(pNewGeoObj)
        cout << pNewGeoObj->GetTypeString()
             << " created." << endl;
    }
  }
```

3. bis hierhin haben wir nur die dynamische Erzeugung von Objekten mit Defaultkonstruktion vorgenommen; num sollen auch die Inhalte der Objekte eingelesen werden -
   Einlesen der Daten der dynamischen Objekte von einem dedizierten Stream:
   – Erzeugen zweier `ifstreams`, Verbinden dieser mit Files: einem Steuerfile, welcher die Typbeschreibungen enthält und einem Datenfile mit dem Inhalt der zu erzeugenden Objekte
   – Start mit dem Einlesen eines Objektes; dieses kann weitere Objekte enthalten (... ein `ZGeoObj` sein)
   – die Objekthierarchie baut sich dynamisch auf
   – nach Ende wird die Fläche das gesamten Objektes (die Summe) ausgegeben und das Objekt an die Fabrik zurückgegeben

```
// File: mGeoFacD.cc

#include <fstream.h>
#include <IOFactry.h>
#include <PunktV.h>
#include <Rechteck.h>
#include <RchtckRn.h>
#include <Dreieck.h>
#include <Kreis.h>
#include <ZGeoObj.h>

int main()
{
  MapOfPrototypes Prototypes;
  ifstream ITypes("mGeoFacD.in");
  ifstream IData("mGeoFacD.dat");
// Collection gives data on cout:
  IOFactory Factory(&Prototypes, &ITypes, &cout);

  Prototypes["[Rechteck]"] = new Rechteck;
  Prototypes["[RechteckRund]"] = new RechteckRund;
  Prototypes["[Dreieck]"] = new Dreieck;
  Prototypes["[Kreis]"] = new Kreis;
  Prototypes["[ZGeoObj]"] = new ZGeoObj(&Factory);

  GeoObj *pz2 = Factory.CreateGeoObj();
  if(pz2)
```

```
    {
      cout << pz2->GetTypeString() << " created." << endl;
      cout << "Now reading data ... ";
      IData >> (*pz2);
      cout << "finished." << endl;
      cout << "Flaeche: " << pz2->flaeche() << endl;
      Factory.CollectGeoObj(pz2);
      cout << "Pointer after collection is: "
           << hex << pz2 << endl;
    }
  }
```

4. Einlesen der Typbeschreibungen und der Daten der dynamischen Objekte von einem gemeinsamen Stream:
   – Erzeugen eines `ifstreams` zum Einlesen der Typbeschreibungen und Daten
   – Erzeugen eines `ofstreams`, auf den die Fabrik während der `CollectGeoObj(pz2)`-Operation die gesamten Daten wieder ausgibt

```
  // File: mGeoFacG.cc

  #include <fstream.h>
  #include <IOFactry.h>
  #include <PunktV.h>
  #include <Rechteck.h>
  #include <RchtckRn.h>
  #include <Dreieck.h>
  #include <Kreis.h>
  #include <ZGeoObj.h>

  int main()
  {
    MapOfPrototypes Prototypes;
    ifstream ITypesAndDataIn("practdat.in");
    ofstream ITypesAndDataOut("practdat.out");
    IOFactory Factory(&Prototypes,
                      &ITypesAndDataIn,
                      &ITypesAndDataOut);

    Prototypes["[Rechteck]"] = new Rechteck;
    Prototypes["[RechteckRund]"] = new RechteckRund;
    Prototypes["[Dreieck]"] = new Dreieck;
    Prototypes["[Kreis]"] = new Kreis;
    Prototypes["[ZGeoObj]"] = new ZGeoObj(&Factory);

    GeoObj *pz2 = Factory.CreateGeoObj();
```

```
    if(pz2)
    {
      cout << pz2->GetTypeString() << " created." << endl;
      cout << "Now reading data ... ";
      ITypesAndDataIn >> (*pz2);
      cout << "finished." << endl;
      cout << "Flaeche: " << pz2->flaeche() << endl;
      Factory.CollectGeoObj(pz2);
      cout << "Pointer after collection is: "
           << hex << pz2 << endl;
    }
  }
```

Praktikumsaufgabe Nr. 8:
Das Beispiel `mGeoFacG.cc` *ist mit einem Input-File* `practdat.in` *verknüpft, welcher ein relativ komplexes persistentes* `ZGeoObj` *enthält. Nach dem richtigen Implementieren der* `ZGeoObj::ReadFrom()`*-Methode sollten Sie fähig sein, den File korrekt einzulesen und und die Fläche als Summe der Flächen der Teilobjekte durch die* `flaeche()`*-Methode berechnen und ausgeben zu lassen. Bitte versuchen Sie dies.*

9 Fehlerbehandlung und Robustheit

9.1 Beispiel: Invariantentest

- in der traditionellen C-Programmierung wird in der Entwicklungsphase eines Programmes oft vom `assert`-Mechanismus Gebrauch gemacht
- ein Aufruf von
  `void assert(int expression);` führt im Falle
  `expression == 0` zu einer Ausgabe der Form
  `Assertion failed: expression, file xyz, line nnn`

Begriff der *Invariante*:

- der Wert der Elemente eines Objektes heißt *Zustand* des Objektes
- Initialisierung sollte ein Objekt in einen 'wohldefinierten' Zustand bringen
- 'wohldefinierter' Zustand eines Objektes soll nach (im strengeren Sinne sogar während) der auf dieses Objekt angewandten Operationen aufrechterhalten werden bis zur Löschung des Objektes (Funktionen, die auf Daten eines Objektes arbeiten und aufgerufen werden können, während der Zustand des Objektes nicht 'wohldefiniert' ist, sollten nicht an der öffentlichen Schnittstelle (`public, protected`) erscheinen)
- Regeln (Daten, Bedingungen), die die 'Wohldefiniertheit' eines Objektes festlegen (bzw. die Grenzen dafür) kennzeichnen die *Invariante* eines Objektes

Praxis:

- ... den Zustand wohldefiniert machen, kann nur der Code des Objektes selbst
- die Invariante kann also zumindest als Code realisiert werden, der den Zustand eines Objektes auf 'Wohldefiniertheit' prüft bzw. vor Abweichungen davon schützt
- häufige Realisierung: Pre- und Postconditions; Precondition prüft Zustand des Objektes und Wert der Argumente beim Eintritt in den Code einer Methode; Postcondition prüft den Zustand des Objektes vor Verlassen der Methode
- Implementation von Invariantentests:

```
// File: precond.h

#ifndef __PRECONDITION_H
#define __PRECONDITION_H

#include <stdio.h>
```

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

class precondition : public xmsg
{
private:
  static const String MakeString(const char *type,
                                 const char *txt,
                                 const char *file,
                                 int line);
public:
  precondition(const char *txt,
               const char *file,
               int line)
               : xmsg(MakeString("Precondition",txt,file,line)){}
};

#define PRECONDITION(p) PRECONDITIONX(p,#p)

#if __DEBUG < 1
#define PRECONDITIONX(p,s)   ((void)0)
#else
#define PRECONDITIONX(p,s)   \
    if(!(p)) {throw precondition(s,__FILE__,__LINE__);}
#endif

#endif

```

- die `PRECONDITION(p)`-Anweisung ist als Makro realisiert, welches erst wirklich Code erzeugt, wenn die Präprozessorvariable `__DEBUG` einen Wert ≥ 1 hat

- wird Code erzeugt, dann expandiert das Makro an der originalen Quelltextzeile zu einer Überprüfung der Bedingung (Precondition) und wirft bei Verletzung eine `precondition`-Exception aus

- dem Konstruktor des Exception-Objektes wird das Bedingungsstatement als String und zusätzlich der Name des Quelltextfiles und die Nummer der aktuellen Quelltextzeile übergeben

- Beispiel:

```
#define __DEBUG 2

#include <precond.h>

void ZGeoObj::Add(GeoObj& _GeoObj)
```

```
{
  PRECONDITION(Count < MAX);
  GeoObjs[Count++] = &_GeoObj;
}
```

- hier wurde in die `Add()`-Methode von `ZGeoObj`, anstelle des ohnehin unzureichenden Tests mittels der `if`-Anweisung eine Precondition eingebaut

Praktikumsaufgabe Nr. 9:
Bitte formulieren Sie Preconditions zum Invariantentest der Klasse `RechteckRund`! *Beachten Sie besonders die Methode* `flaeche`. *Schreiben Sie bitte ein Beispielprogramm, in welchem Sie gezielt diese Precondition verletzen, die ausgeworfene Exception handeln und auswerten!*

10 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` $\rightarrow$ `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` $\rightarrow$ `int`$>$ oder $<$`float` $\rightarrow$ `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.

10.0.1 Nachteile des bisherigen Designs

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

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

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

10.1 Templates: Grundlagen

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

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

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

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

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

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

10.2 Beispiel: Map-Template

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

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

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

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

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

```
#ifndef _MAPT_H_
#define _MAPT_H_

#include <iostream.h>
#include <String.h>
#include <MapAssoT.h>
#include <MapIterT.h>

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

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

public:
  Map(V _InitVal, int = 16);
  ~Map();
```

```
  V& operator[](const K&);

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

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

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

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

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

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

```
        anzahl *= 2;
      }
    }
    frei++;
    pp->Key = k1;
    pp->Value = InitVal;
    return pp->Value;
  }

  #endif
```

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

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

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

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

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

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

10.3 Iteration

- ein Container speichert Objekte, d.h. Daten, mit denen das Programm im weiteren Verlauf arbeitet; denkbar wären Aktionen wie Aus-, Eingabe, Suchen, Verknüpfen, Kopieren, Einfügen, Löschen, Umordnen (Sortieren), ...

- müssen im Zusammenhang mit einer dieser Aktionen (z.B. Suchen) entsprechende Operationen (z.B. Vergleich) auf mehrere Objekte, also auf einen Bereich des Containers (z.B. von Anfang bis Ende) angewendet werden, dann spricht man von einer Operation auf der Basis einer *Iteration* über diesen Bereich

- Iterationen können möglicherweise von verschiedenen Stellen in einem Programmsystem aus gleichzeitig über demselben Containerbereich durchgeführt werden; das Speichern des Zustandes einer Iteration (z.B. aktuelles Element) sollte daher nicht im Container selbst erfolgen
- man definiert Iteratorobjekte, die einerseits den Zustand der Iteration speichern und andererseits Funktionen zur Steuerung der Iteration besitzen;
- für die syntaktische Konstruktion von Iterationen gibt es viele Möglichkeiten, z.B.
  - ```
    Container_Type c;
    Iterator_Type i;
    for(i = c.First(); i; c.next(i)) Use(c(i));
    ```
    * z.B. `c.First()` liefert einen Iterator, der mit dem ersten Element initialisiert ist
    * `c.next(i)` inkrementiert den Iterator
    * `Use(c(i))` ruft eine Funktion für das vom Iterator als aktuelles bestimmte Element des Containers `c` auf
  - ```
    Element_Type e;
    Container_Type c;
    Iterator_Type i(c);
    while(e = i.next()) Use(e);
    ```
    * z.B. `i.next()` gibt in `e` jeweils das aktuelle Element oder NULL (Ende!) zurück und inkrementiert die interne Position der Iteration
  - oft auch zeigerartige Notation:
    ```
    Container_Type c;
    Iterator_Type i;
    for(i = c.First(); i; i++) Use(*i);
    ```
    * `*i` *dereferenziert* den Iterator und liefert eine Referenz auf das aktuelle Element
  - in der STL wird der Iterationsbereich oft durch zwei Iteratoren abgesteckt, die untereinander verglichen werden können:
    ```
    Container_Type c;
    Iterator_Type First = c.begin(), Last = c.end();
    while(First != Last) Use(*first++);
    ```

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

10.4 Generische Algorithmen

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

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

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

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

```
// File: MapFuncT.h

#ifndef _MAPFUNCT_H_
#define _MAPFUNCT_H_

#include <iostream.h>
#include <String.h>
#include <MapAssoT.h>
#include <MapIterT.h>

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

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

```
  bool NextVal = true;
  MapIterator<K,V> Next;

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

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

      Next++;
    }

    ActVal++;
  }
  return ros;
}

#endif
```

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

- eine mögliche Programmausgabe:

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

```
d4 t6

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

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

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

11 Standard Template Library (STL)

In diesem Kapitel kann es nur um eine Einführung in die C++-Standardbibliothek, speziell die Standard Template Library, gehen. Tiefergehende Informationen sind den Programmier- und Referenzhandbüchern der jeweiligen Implementation zu entnehmen.

11.1 Grundlagen

- Standardisierungsgremien (ISO WG21 und ANSI X3J16) definieren auch Standardbibliothek

- C++-Standardbibliothek nach ANSI/ISO-Norm besteht aus den Komponenten:

  - Sammlung von Datenstrukturen und Algorithmen (Standard Template Library)
  - IOStream-Klassen
  - Funktionalität für landesspezifische Zeichensätze
  - String-Template
  - Template für Darstellung komplezer Zahlen
  - Mechanismus zur Beschreibung eines einheitlichen (portablen) Mechanismus zur Beschreibung der Ausführungsumgebung (Limits, eingebaute Typen)
  - Funktionen zur Speicherverwaltung
  - Funktionen zur weiteren Sprachunterstützung
  - Funktionen fürs Exception-Handling

- STL verwendet kaum objektorientierte Designmöglichkeiten, Kapselung als Zusammenführung von Daten und Funktionalität wird vermieden; Datenstrukturen und Algorithmen liegen getrennt vor

- in STL ist kein Konzept zur Fehlerbehandlung enthalten, Gültigkeit von Aufrufen, Iteratoren, Argumenten etc. muß vom Programmierer sichergestellt werden

- STL ist daher eher eine Sammlung von Datenstrukturen und davon getrennter Algorithmen als eine objektorientierte Klassenbibliothek

- Vorteile dieses Ansatzes sind:

  - schlankerer Quelltext
  - Flexibilität der Algorithmen
  - Effizienz

11.1.1 Verwendungsmöglichkeit der STL

- zum Beschreiben eines komplexen Problems sollten eher die objektorientierten Techniken, also vor allem Kapselung und Vererbung verwendet werden
- STL sollte dort zum Einsatz kommen, wo im Kern hocheffiziente Algorithmen benötigt werden
- STL sollte dann mit dem programmeigenen Fehlerbehandlungskonzept verknüpft werden
- in Multithreading-Umgebungen muß die STL mit Vorsicht eingesetzt werden; ein Container kann nicht sicher geschützt werden, wenn er durch mehrere Iteratoren in parallelen Threads bearbeitet wird; in dem Falle sind Rahmenobjekte zur Threadsynchronisation einzusetzen

11.2 Container

- Standardbibliothek stellt nicht weniger als zehn verschiedene Container-Klassen bereit: `vector`, `list`, `deque`, `set`, `multiset`, `map`, `multimap`, `stack`, `queue`, `priority_queue`
- Container dienen zur Verwaltung von Objekten eines Typs (bzw. von diesem abgeleitete Typen)
- verschiedene Containerklassen haben unterschiedliche Techniken zur Verwaltung der Objektmenge
- *sequentielle Container* (`vector`, `list`, `deque`) verwalten Mengen, deren Elemente durch die Reihenfolge des Einfügens geordnet sind:

Frank	Marian	Claus	Martin	Rico	....

- *sortierte Container* (`set`, `multiset`, `map`, `multimap`) verwalten Mengen, deren Elemente nach einem speziellen Sortierkriterium geordnet sind, sortierte Container verwalten also die eigentlichen sortierten Mengen, Beispiel Map:

Claus	Frank	Marian	Martin	Rico	... Keys (Strings, sortiert)
0	3	5	10	13	... Values (int’s, den Strings zugeordnet)

- Überblick:

	Name	Merkmale
sequentielle Container	`vector`	direkter Zugriff auf Elemente, effizientes Einfügen am Ende
	`list`	effizientes Einfügen und Entfernen an beliebiger Position
	`deque`	direkter Zugriff, effizientes Einfügen am Anfang oder Ende
	`stack`	Einfügen und Entfernen nur am Anfang
	`queue`	Einfügen am Ende, Entfernen am Anfang
	`priority_queue`	effizienter Zugriff und Entfernen des größten Wertes
sortierte Container	`set`	Elemente nach ihrem Wert sortiert, effizienter Test auf Inklusion, effizientes Einfügen und Entfernen
	`multiset`	ist `set`, wobei gleiche Elemente mehrfach erlaubt
	`map`	verwaltet Schlüssel/Werte-Paare, Paare geordnet nach Schlüssel, Zugriff auf Werte über Index, effizientes Einfügen und Entfernen
	`multimap`	ist `map`, wobei gleiche Schlüssel mehrfach erlaubt

11.2.1 Wahl des geeigneten Containers

- Wie soll auf die Elemente zugegriffen werden?
  direkter Zugriff: `vector` oder `deque`
- Ist die Reihenfolge, in der die Elemente im Container gespeichert sind, wichtig?
  - wenn über die gesamte Lebensdauer des Containers die Elemente in definierter Reihenfolge gespeichert sein sollen, dann sortierten Container verwenden (`(multi-)set` oder `(multi-)map`)
  - wenn Sortierung nur zu einem bestimmten Zeitpunkt notwendig: sequentiellen Container verwenden, Sortieralgorithmus laufen lassen, wenn nötig
  - wenn Sortierung anhand der Reihenfolge der Einfügung wichtig ist, dann `stack` oder `queue` verwenden
- Ändert sich die Größe der Struktur während der Ausführung stark?
  `list` oder `set` verwenden, denn `vector` oder `deque` kann Größe in der Regel nur erhöhen
- Kann die Größe des Containers abgeschätzt werden?
  `vector` bietet Möglichkeit, Speicherbereich im Vorfeld zu reservieren

- Muß häufig auf Inklusion bestimmter Werte getestet werden?
  `set` oder `map` verwenden: Aufwand logarithmisch zur Größe des Containers (bei sequentiellen Containern im schlechtesten Fall Vergleich mit jedem Element)

- Soll die Menge indiziert werden? Kann die Menge als Folge von Index/Wert-Paaren gedacht werden?
  wenn Index vom Typ Integer, kann `vector` oder `deque` verwendet werden; wenn Index anderen, sortierbaren Typ (z.B. Strings) haben, dann `map` verwenden

- Wie können Werte verglichen werden?
  Für die Speicherung in einem STL-Container muß in der Regel der Gleichheitsoperator für den Typ der Werte sinnvoll definiert sein; für die Speicherung in einer `set` muß zudem noch der Kleiner-als-Operator für die Werte, für die Speicherung in einer `map` muß dieser für den Typ der Schlüssel (Indextyp) definiert werden

- Muß häufig der größte Wert des Containers gefunden oder entfernt werden?
  `priority_queue` verwenden

- an welcher Position sollen die Elemente dem Container hinzugefügt werden?
  beliebige Position: `list` verwenden; Anfügen am Anfang: `deque` verwenden; Einfügen am Ende: `queue` verwenden

- Sollen häufig zwei Container zu einem zusammengeführt werden? `set` oder `list` verwenden, je nachdem, ob Ordnung der Elemente aufrechterhalten werden soll

11.2.2 Nicht enthaltene Container und Möglichkeiten

Container	Möglichkeiten der Definition
`tree`	`set` ist intern durch binären Suchbaum implementiert; also `set` verwenden
mehrdim. Arrays	konstruieren durch `vector`, der andere `vector`-Objekte aufnimmt
`graph`	konstruieren durch `map`, die andere `map`-Objekte aufnimmt
`sparse array`	konstruieren wie bei `graph`
`hash tables`	konstruieren durch `vector` oder `deque`, welche `lists` oder `sets` aufnehmen

11.2.3 Beispiele

Container	Beispiele
`vector`	Sieb des Eratosthenes
`list`	Speichern beliebiger Dinge,
	Prozeßeinträge, Fileknoten, Lagerbestände
`deque`	Sortieren von Beständen
`set`, `multiset`	Wörterbuch
`map`, `multimap`	Datenbanken, Übersetzung, diskrete Funktionen
`stack` (LIFO)	Taschenrechner
`queue` (FIFO)	Warteschlangen: Batchsysteme, Simulationen
`priority_queue`	ereignisgesteuerte Simulation

Beispiel: Sieb des Eratosthenes
File: stl_sieve.cc