Objektorientierte Programmierung mit C++
WS 2012/2013

Andreas F. Borchert

Universität Ulm

4. Februar 2013

Syllabus

Inhalte:

- Einführung in OO-Design, UML und »Design by Contract«.
- Einführung in C++
- Polymorphismus in C++
- Templates
- Dynamische Datenstrukturen mit *smart pointers*
- Statischer vs. dynamischer Polymorphismus
- STL-Bibliothek
- iostream-Bibliothek
- Ausnahmenbehandlungen
- Fortgeschrittene Template-Techniken
- Potentiale und Auswirkungen optimierender Übersetzer bei C++
- Funktionsobjekte und Lambda-Ausdrücke in C++

C++ ist trotz zahlreicher historischer Relikte (C sei Dank) und seiner hohen Komplexität nach wie vor interessant:

- Analog zu C bietet C++ eine hohe Laufzeit-Effizienz.
- Im Vergleich zu anderen OO-Sprachen kann sehr flexibel bestimmt werden, wieviel statisch und wieviel dynamisch festgelegt wird. Entsprechend muss der Aufwand für OO-Techniken nur dort bezahlt werden, wo er wirklich benötigt wird.
- Zahlreiche vorhandene C-Bibliotheken wie etwa die BLAS-Bibliothek oder die GMP (GNU Multi-Precision-Library) lassen sich als Klassen in C++ verpacken.
- Die STL und darauf aufbauende Klassen-Bibliotheken sind recht attraktiv.

Somit ist C++ insbesondere auch für rechenintensive mathematische Anwendungen recht interessant.

Syllabus: Fokus auf C++11

- Im August 2011 wurde der aktuelle ISO-Standard für C++ veröffentlicht, ISO 14882-2012, der beachtliche Neuerungen einführt im Vergleich zu dem vorherigen Standard von 2003.
- Verschiedene dieser neuen Techniken wurden bereits experimentell durch Teile der Boost-Library eingeführt und sind deswegen schon seit etwas längerer Zeit etabliert wie etwa *smart pointers*.
- Auch unabhängig von der konkreten Sprache C++ sind diese interessant, weil sie u.a. zeigen, wie
  - Elemente funktions-orientierter Programmiersprachen in klassischen objekt-orientierten Sprachen effizient eingebettet werden können (Lambda-Ausdrücke) und wie
  - komplexe dynamische Datenstrukturen auch ohne *garbage collection* elegant verwaltet werden können (*smart pointers*).
- Ein wesentliches Ziel der Vorlesung ist es zu zeigen, wie moderne Sprachtechniken effizient oder sogar effizienzsteigernd eingesetzt werden können.

- Im Open-Source-Bereich gibt es zwei Übersetzer, die hier in Frage kommen: GCC 4.7.x und Clang.
- Im Rahmen der Vorlesung werden wir primär mit GCC 4.7.x arbeiten. Auf der Thales steht 4.7.1 zur Verfügung.
- Um diese nutzen zu können, sollten Sie bei uns gcc47 in der Datei *~/.options* aufnehmen.
- Auf den anderen Rechnern bei uns sind nur ältere gcc-Versionen installiert.
- Unter Debian erhalten Sie die aktuelle Fassung im *testing*-Zweig.

Syllabus: Voraussetzungen

- Kenntnisse in
  - einer Programmiersprache (egal welche) und in
  - Algorithmen und Datenstrukturen (Bäume, Hash-Verfahren, Rekursion).
- Freude am Entwickeln von Software und der Arbeit im Team
- Fähigkeit zur selbständigen Arbeitsweise einschließlich dem Lesen von Manualseiten und der eigenständigen Fehlersuche (es gibt keine Tutoren!)

Syllabus: Ziele

- Erwerb von praktischer Erfahrung und soliden Kenntnissen im Umgang mit C++
- Erlernen der Herangehensweise, wie ausgehend von den Anforderungen und dem Entwurf die geeigneten programmiersprachlichen Techniken ausgewählt werden
- Erlernen fortgeschrittener Techniken und der Erwerb der Fähigkeit, diese sinnvoll einzusetzen. Dazu gehören insbesondere Techniken, die von Java nicht unterstützt werden wie etwa statischer Polymorphismus und Lambda-Ausdrücke.
- Erwerb von Grundkenntnissen über die Implementierungen verschiedener Techniken, so dass das zu erwartende Laufzeitverhalten eingeschätzt werden kann

Syllabus: Struktur

- Jede Woche gibt es zwei Vorlesungsstunden an jedem Montag von 16-18 Uhr im Raum E20 in der Helmholtzstraße 18.
- Die Übungen finden an jedem Freitag von 14-16 Uhr ebenfalls im Raum E20 statt.
- Webseite: http://www.mathematik.uni-ulm.de/sai/ws12/cpp/
- Alle Vorlesungsteilnehmer mögen sich bitte bei SLC für die Vorlesung registrieren.

- Die Übungen werden von mir betreut.
- Die praktische Abwicklung der Übungen wird in den ersten Übungen am 19. Oktober vorgestellt.
- Typischerweise läuft das so ab, dass kurz eine Lösung zu der alten Übungsaufgabe vorgestellt wird, Erläuterungen und Hinweise zum neuen Übungsblatt gegeben werden und dann in einigen Fällen die Gelegenheit besteht, im für um diese Zeit reservierten Pool-Raum E44 mit ersten Arbeiten zu beginnen, wobei ich betreuenderweise zur Seite stehe.
- Es gibt keine formale Vorleistung für die Teilnahme an der schriftlichen Prüfung. Dennoch wird die intensive Teilnahme an den Übungen dringend empfohlen, weil nur dann eine erfolgreiche Teilnahme an der schriftlichen Prüfung denkbar ist.

- Für die Studenten, die entsprechend den neueren Prüfungsordnungen studieren (Bachelor, Master mit POs nach 2005), gibt es am Ende des Semesters eine schriftliche Prüfung und kurz vor Beginn des folgenden Sommersemesters eine zweite Prüfung.
- Die Termine können wir gemeinsam festlegen. Damit wir das nächste Woche tun können, sollten Sie alle bitte ihre anderen Termine sammeln, damit Konflikte vermieden werden können.
- Für Studenten in Diplomstudiengängen oder alte Bachelor/Master-Studiengänge werden mündliche Prüfungen angeboten. Hier sind Termine bei mir persönlich zu vereinbaren.
- Wer noch einen Übungsschein benötigt, möge das bitte frühzeitig (also noch im Oktober) mir mitteilen.

- Die Vorlesungsfolien und einige zusätzliche Materialien werden auf der Webseite der Vorlesung zur Verfügung gestellt werden.
- Dort finden sich auch Verweise auf zwei Arbeitsfassungen des C++-Standards (ISO/IEC 14882):
  - November 2006 (näher am früheren Standard von 2003)
  - August 2010 (näher am aktuellen Standard, einiges davon ist bereits bei GCC 4.7 verfügbar, siehe *http://gcc.gnu.org/gcc-4.7/cxx0x_status.html*).
- Der aktuelle Standard kann im Original als PDF auch von ISO oder ANSI bezogen werden (bei ANSI momentan für 30 US$). Das Dokument hat einen Umfang von 1.356 Seiten. (Im Normalfall sollte aber der Draft vom August 2010 genügen. Eine leichte Lektüre ist der Standard nicht.)

- Bjarne Stroustrup, *The C++ Programming Language*, ISBN 0-201-88954-4
- Bjarne Stroustrup, *The Design and Evolution of C++*, ISBN 0-201-54330-3
- David Vandevoorde und Nicolai M. Josuttis, *C++ Templates: The Complete Guide*, ISBN 0-201-73484-2
- David Abrahams und Aleksey Gurtovoy, *C++ Template Metaprogramming*, ISBN 0-321-22725-5
- David R. Musser und Atul Saini, *STL Tutorial and Reference Guide*, ISBN 0-201-63398-1
- Steve Teale, *C++ IOStreams Handbook*, ISBN 0-201-59641-5
- Scott Meyers, *Effective C++*, ISBN 0-201-92488-9
- Scott Meyers, *More Effective C++*, ISBN 0-201-63371-X
- Scott Meyers, *Effective STL*, ISBN 0-201-74962-9

Folgende Literatur ist etwas älter, war jedoch wegweisend:

- Bertrand Meyer, *Object-Oriented Software Construction*, Second Edition, 1997
- Grady Booch, *Object-Oriented Analysis and Design with Applications*, Second Edition, 1994, ISBN 0-8053-5340-2
- Erich Gamma et al, *Design Patterns*, ISBN 0-201-63361-2
- Booch, Jacobson, and Rumbaugh, *The Unified Modeling Language User Guide*, Addison Wesley, 1999, ISBN 0-201-57168-4 (Referenz zu UML, jedoch nicht sehr gelungen)

Syllabus: Sprechstunden

- Sie sind eingeladen, mich jederzeit per E-Mail zu kontaktieren:
  E-Mail: `andreas.borchert@uni-ulm.de`
- Meine reguläre Sprechzeit ist am Mittwoch 10-12 Uhr. Zu finden bin ich in der Helmholtzstraße 18, Zimmer E02.
- Zu anderen Zeiten können Sie auch gerne vorbeischauen, aber es ist dann nicht immer garantiert, daß ich Zeit habe. Gegebenenfalls lohnt sich vorher ein Telefonanruf: 23572.

Syllabus: Nicht verzweifeln!

- Ich helfe auch gerne bei Problemen bei der Lösung von Übungsaufgaben. Bevor Sie völlig verzweifeln, sollten Sie mir Ihren aktuellen Stand per E-Mail zukommen lassen. Dann werde ich versuchen, Ihnen zu helfen.
- Das kann auch am Wochenende funktionieren.

Objekt-orientierte Techniken sind auf dem Wege neuer Programmiersprachen eingeführt worden, um Probleme mit traditionellen Programmiersprachen zu lösen:

- Simula (1973) von Nygaard und Dahl:
  - Erste OO-Programmiersprache.
  - Die Technik wurde eingeführt, um die Modellierung von Simulationen zu erleichtern.

- Smalltalk wurde in den späten 70er-Jahren bei Xeroc PARC entwickelt und 1983 von Adele Goldberg publiziert:
  - Erste radikale OO-Programmiersprache: Alles sind Objekte einschließlich der Klassen.
  - Die Sprache wurde entwickelt, um die Modellierung und Implementierung der ersten graphischen Benutzeroberfläche zu unterstützen.

- C++, das sich zu Beginn noch *C with Classes* nannte, begann seine Entwicklung 1979 und gehört damit zu den frühesten OO-Programmiersprachen.
- Bjarne Stroustrup scheiterte in seinem Bemühen, Simulationen mit den zur Verfügung stehenden Lösungen umzusetzen:
  - Simula: (schöne Programmiersprache; unzumutbare Performance)
  - BCPL: (unzumutbare Programmiersprache; hervorragende Performance)
- Entsprechend war das Ziel von Stroustrup, die Effizienz von C mit der Eleganz von Simula zu kombinieren.

Assembler und viele traditionelle Programmiersprachen (wie etwa Fortran, PL/1 und C) bieten folgende Struktur:

- Eine beliebige Zahl von Übersetzungseinheiten, die unabhängig voneinander zu sogenannten Objekten übersetzt werden können, lassen sich durch den Binder zu einem ausführbaren Programm zusammenbauen.
- Jede Übersetzungseinheit besteht aus global benutzbaren Funktionen und Variablen.
- Parameter und globale Variablen (einschließlich den dynamisch belegten Speicherflächen) werden für eine mehr oder weniger unbeschränkte Kommunikation zwischen den Übersetzungseinheiten verwendet.

- Anwendungen in traditionellen Programmiersprachen tendieren dazu, sich rund um eine Kollektion globaler Variablen zu entwickeln, die von jeder Übersetzungseinheit benutzt und modifiziert werden.
- Dies erschwert das Nachvollziehen von Problemen (wer hat den Inhalt dieser Variable verändert?) und Änderungen der globalen Datenstrukturen sind nicht praktikabel.

Nachfolger der traditionellen Programmiersprachen (wie etwa Modula-2 und Ada) führten Module ein:

- Module schränken den Zugriff ein, d.h. es sind nicht mehr alle Variablen und Prozeduren global zugänglich.
- Stattdessen wird eine Schnittstelle spezifiziert, die alle öffentlich nutzbaren Prozeduren und Variablen aufzählt.
- Abstrakte Datentypen erlauben den Umgang mit Objekten, deren Innenleben verborgen bleibt.
- Dies erlaubt das Verbergen der Datenstrukturen hinter Zugriffsprozeduren.

- Die abstrakten Schnittstellen sind nicht wirklich getrennt von den zugehörigen Implementierungen, d.h. zwischen beiden liegt eine 1:1-Beziehung vor (zumindest aus der Sicht eines zusammengebauten Programms).
- Entsprechend können nicht mehrere Implementierungen eine Schnittstelle gemeinsam verwenden.

- Alle Daten werden in Form von Objekten organisiert (mit Ausnahme einiger elementarer Typen wie etwa dem für ganze Zahlen).
- Auf Objekte wird (explizit oder implizit) über Zeiger zugegriffen.
- Objekte bestehen aus einer Sammlung von Feldern, die entweder einen elementaren Typ haben oder eine Referenz zu einem anderen Objekt sind.
- Objekte sind verpackt: Ein externer Zugriff ist nur über Zugriffsprozeduren möglich (oder explizit öffentliche Felder).

- Eine Klasse assoziiert Prozeduren (Methoden genannt) mit einem Objekt-Typ. Im Falle abstrakter Klassen können die Implementierungen der Prozeduren auch weggelassen werden, so dass nur die Schnittstelle verbleibt.
- Der Typ eines Objekts (der weitgehend in den OO-Sprachen durch eine Klasse repräsentiert wird) spezifiziert die externe Schnittstelle.
- Objekt-Typen können erweitert werden, ohne die Kompatibilität zu ihren Basistypen zu verlieren. (In Verbindung mit Klassen wird hier gelegentlich von Vererbung gesprochen.)
- Objekte werden von einer Klasse mit Hilfe von Konstrukturen erzeugt (instantiiert).

Es gibt eine Vielzahl von OO-Sprachen, die mit sehr unterschiedlichen Ansätzen in folgenden Bereichen arbeiten:

- Die Verpackung (d.h. die Eingrenzung der Sichtbarkeit) kann über Module, Klassen, Objekten oder über spezielle Deklarationen wie etwa den *friends* in C++ erfolgen.
- Die Beziehungen zwischen Modulen, Klassen und Typen werden unterschiedlich definiert.
- Die Art der Vererbung bzw. Erweiterung: einfache vs. mehrfache Vererbung bzw. Erweiterung von Typen vs. Erweiterung von Klassen.
- Wie wird im Falle eines Methoden-Aufrufs bei einem Objekt der zugehörige Programmtext lokalisiert? Das ist nicht trivial im Falle mehrfacher Vererbung oder gar Multimethoden.

- Aufrufketten durch Erweiterungshierarchien (von der abgeleiteten Klasse hin zur Basisklasse oder umgekehrt).
- Statische vs. dynamische Typen.
- Automatische Speicherbereinigung (*garbage collection*) vs. explizite manuelle Speicherverwaltung.
- Organisation der Namensräume.
- Unterstützung für Ausnahmenbehandlungen und generische Programmierung.
- Zusätzliche Unterstützung für aktive Objekte, Aufruf von Objekten über das Netzwerk und Persistenz.
- Unterstützung traditioneller Programmiertechniken.

- Generische Module sind eine Erweiterung des Modulkonzepts, bei der Module mit Typen parametrisiert werden können.
- Ziel der generischen Programmierung ist die Erleichterung der Wiederverwendung von Programmtext, was insbesondere bei einer 1:1-Kopplung von Schnittstellen und Implementierungen ein Problem darstellt.
- Generische Module wurden zunächst bei CLU eingeführt (Ende der 70er Jahre am MIT) und wurden dann insbesondere bekannt durch Ada, das sich hier weitgehend an CLU orientierte.

- Traditionelle OO-Techniken und generische Module sind parallel entwickelte Techniken zur Lösung der Beschränkungen des einfachen Modulkonzepts.
- Beides sind völlig orthogonale Ansätze, d.h. sie können beide gleichzeitig in eine Programmiersprache integriert werden.
- Dies geschah zunächst für Eiffel (Mitte der 80er Jahre) und wurde später bei Modula-3 und C++ eingeführt.
- OO-Techniken können prinzipiell generische Module ersetzen, umgekehrt ist das jedoch schwieriger.
- Beide Techniken haben ihre Stärken und Schwächen:
  - OO-Techniken: Erhöhter Aufwand zur Lokalisierung des Programmtexts und mehr Typüberprüfungen zur Laufzeit; flexibler in Bezug auf dynamisch nachgeladenen Modulen
  - Generische Module: Höhere Laufzeiteffizienz, jedoch inflexibel gegenüber dynamisch nachgeladenen Modulen

- Die Metaprogrammierung erlaubt es, den Übersetzer selbst zu programmieren.
- Obwohl eine Metaprogrammierung für C++ ursprünglich nicht vorgesehen worden ist, wurde sie durch spezielle Template-Techniken möglich. Aus heutiger Sicht gehört die Metaprogrammierung zu den wesentlichen Elementen von C++ – auch wenn dies häufig wenig sichtbar in den Bibliotheken verborgen ist.
- Durch spezielle Bibliotheken für die Metaprogrammierung wird die Anwendung inzwischen vereinfacht.
- Der Vorteil der Metaprogrammierung liegt darin, mehr Dinge bereits zur Übersetzzeit zu bestimmen, so dass dies nicht mehr zur Laufzeit geschehen muss mit dem wesentlichen Punkt, dass alles zur Übersetzzeit überprüft werden kann.

- Mit der Einführung verschiedener objekt-orientierter Programmiersprachen entstanden auch mehr oder weniger formale graphische Sprachen für OO-Designs.
- Popularität genossen unter anderem die graphische Notation von Grady Booch aus dem Buch "Object-Oriented Analysis and Design", OMT von James Rumbaugh (Object Modeling Technique), die Diagramme von Bertrand Meyer in seinen Büchern und die Notation von Wirfs-Brock et al in "Designing Object-Oriented Software".
- Später vereinigten sich Grady Booch, James Rumbaugh und Ivar Jacobson in Ihren Bemühungen, eine einheitliche Notation zu entwerfen. Damit begann die Entwicklung von UML Mitte der 90er Jahre.

- UML wird als Standard von der Object Management Group (OMG) verwaltet.
- Die Version 2.4.1 ist die aktuelle Fassung vom 5. August 2011.
- Zu dem Standard gehören mehrere Dokumente, wovon für uns insbesondere die *OMG UML Superstructure* interessant ist: Im Abschnitt 7 werden Klassendiagramme beschrieben, im Abschnitt 14 u.a. Sequenzdiagramme und im Abschnitt 16 Use Cases.
- Bei den Abschnitten werden jeweils im Unterabschnitt 3 die einzelnen Elemente eines Diagramms beschrieben und im Unterabschnitt 4 die einzelnen graphischen Elemente einer Diagrammart tabellarisch zusammengefasst.
- Die einzelnen Dokumente des Standards lassen sich von *http://www.omg.org/* herunterladen.

- Anders als die einfacheren Vorgänger vereinigt UML eine Vielzahl einzelner Notationen für verschiedene Aspekte aus dem Bereich des OO-Designs und es können deutlich mehr Details zum Ausdruck gebracht werden.
- Somit ist es üblich, sich auf eine Teilmenge von UML zu beschränken, die für das aktuelle Projekt ausreichend ist.
- Wir beschränken uns im Rahmen der Vorlesung auf nur drei Diagrammarten, die eine größere Verbreitung erfahren haben und dort jeweils nur auf eine kleine Teilmenge der Ausdrucksmöglichkeiten.

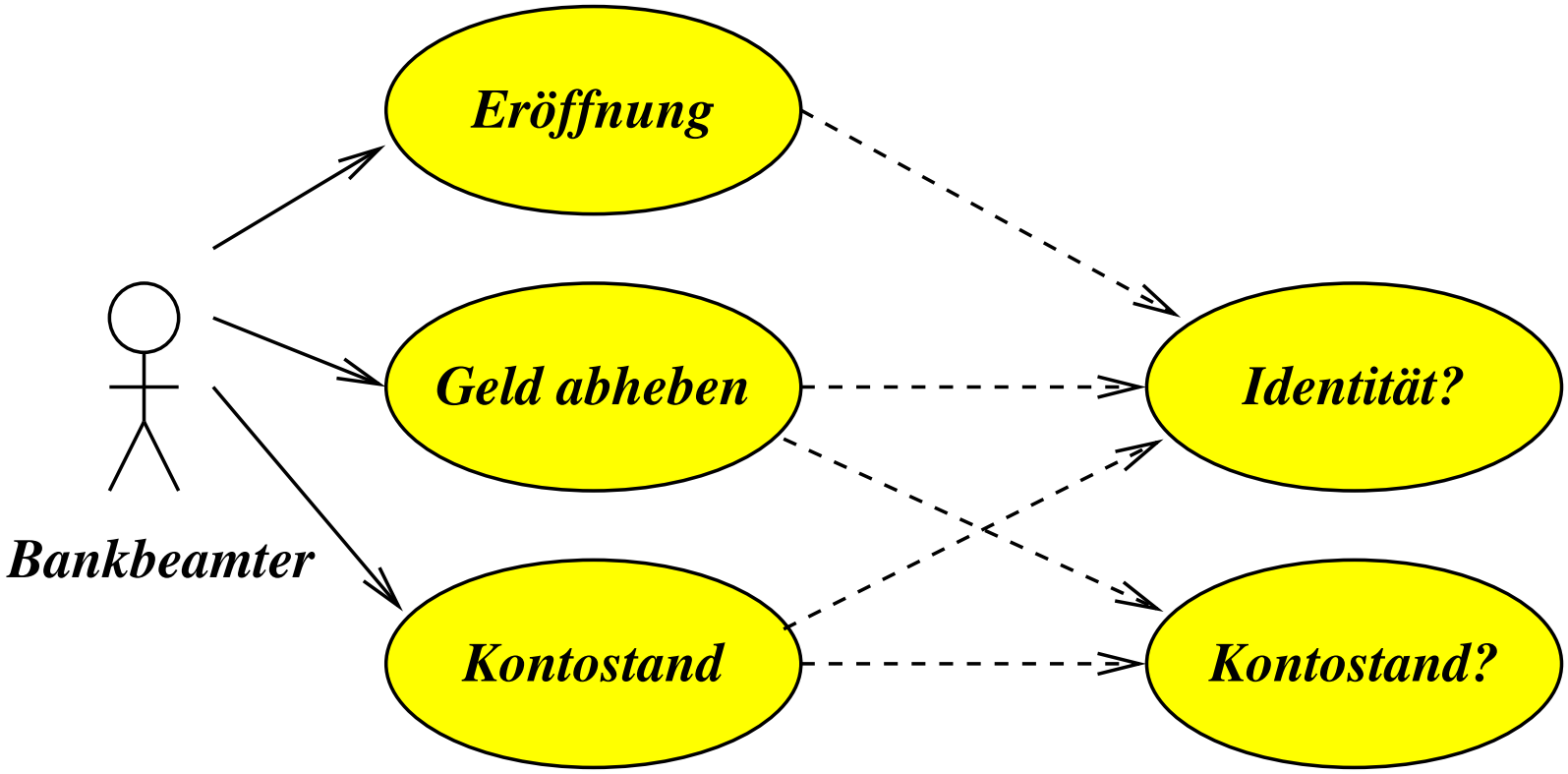
- "Use Cases" dokumentieren während der Analyse die typischen Prozeduren aus der Sicht der aktiven Teilnehmer (Akteure) für ausgewählte Fälle.
- Akteure sind aktive Teilnehmer, die Prozesse in Gang setzen oder Prozesse am Laufen halten.

- Akteure können
  - von Menschen übernehmbare Rollen, die direkt interaktiv mit dem System arbeiten,
  - andere Systeme, die über Netzwerkverbindungen kommunizieren oder
  - interne Komponenten sein, die kontinuierlich laufen (wie beispielsweise die Uhr).
- "Use Cases" werden informell dokumentiert durch die Aufzählung einzelner Schritte, die zu einem Vorgang gehören und können in graphischer Form zusammengefaßt werden, wo nur noch die Akteure, die zusammengefaßten Prozeduren und Beziehungen zu sehen sind.

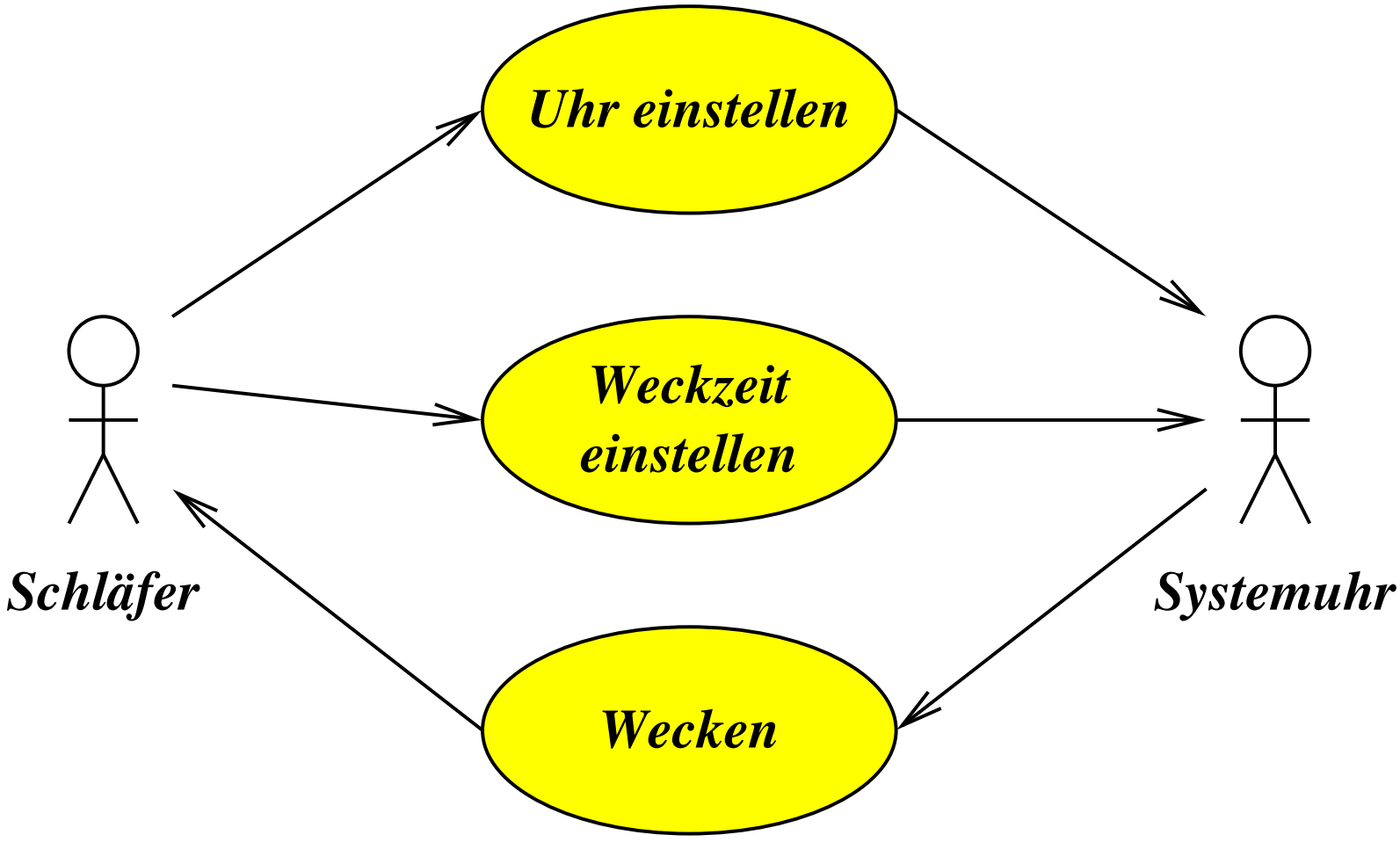
Aus welchen für die Nutzer sichtbaren Schritten bestehen einzelne typische Abläufe bei dem Umgang mit Bankkunden?

Konto-Eröffnung	Feststellung der Identität Persönliche Angaben erfassen Kreditwürdigkeit überprüfen
Geld abheben	Feststellung der Identität Überprüfung des Kontostandes Abbuchung des abgehobenen Betrages
Auskunft über den Kontostand	Feststellung der Identität Überprüfung des Kontostandes

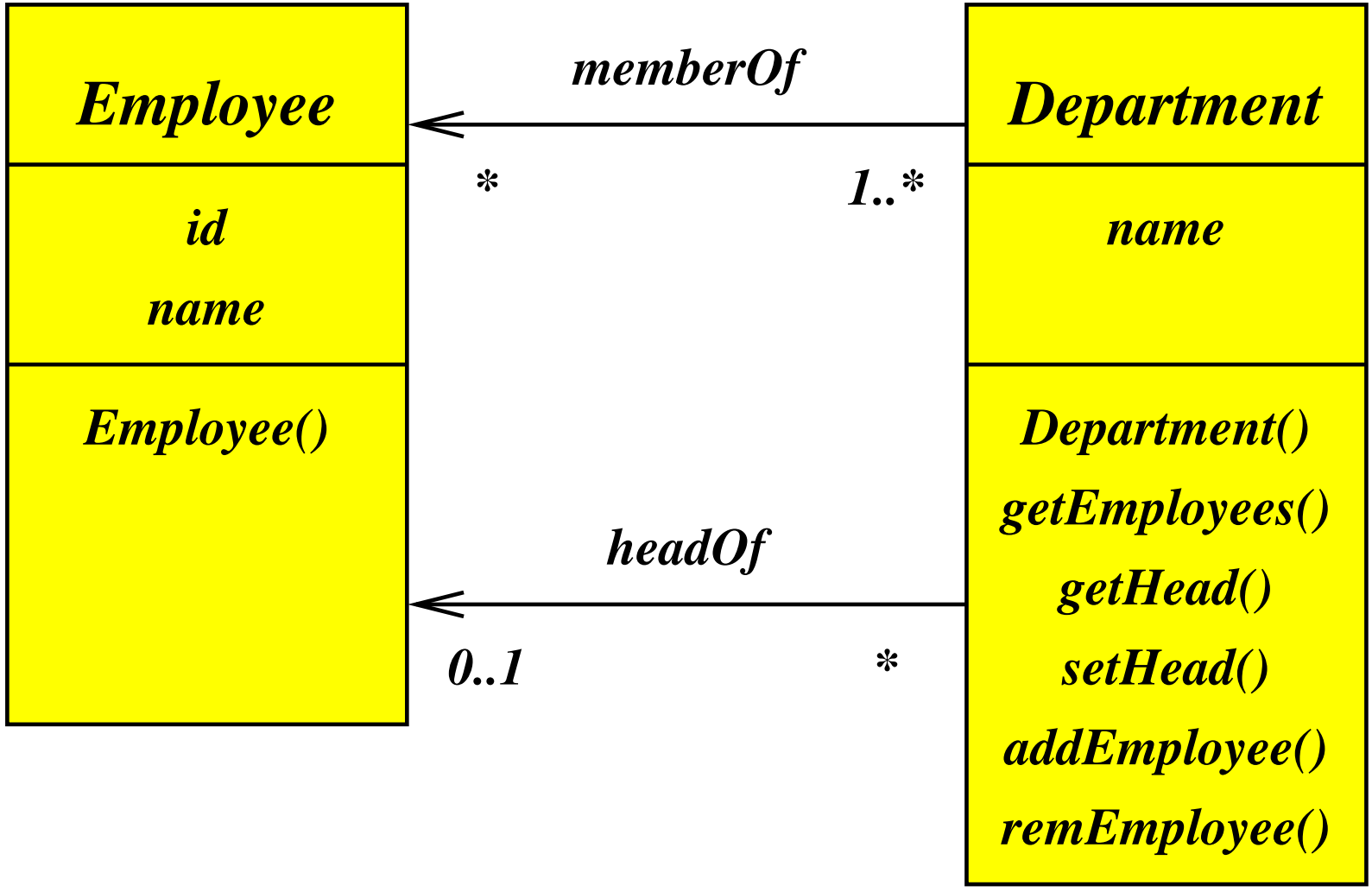
- Hier wurden nur die Aktivitäten aufgeführt, die der Schalterbeamte im Umgang mit dem System ausübt.
- Der Akteur ist hier der Schalterbeamte, weil er in diesen Fällen mit dem System arbeitet. Der Kunde wird nur dann zum Akteur, wenn er beispielsweise am Bankautomaten steht oder über das Internet auf sein Bankkonto zugreift.
- Interessant sind hier die Gemeinsamkeiten einiger Abläufe. So wird beispielsweise der Kontostand bei zwei Prozeduren überprüft.


Eröffnung
Geld abheben
Kontostand
Identität?
Kontostand?
Bankbeamter

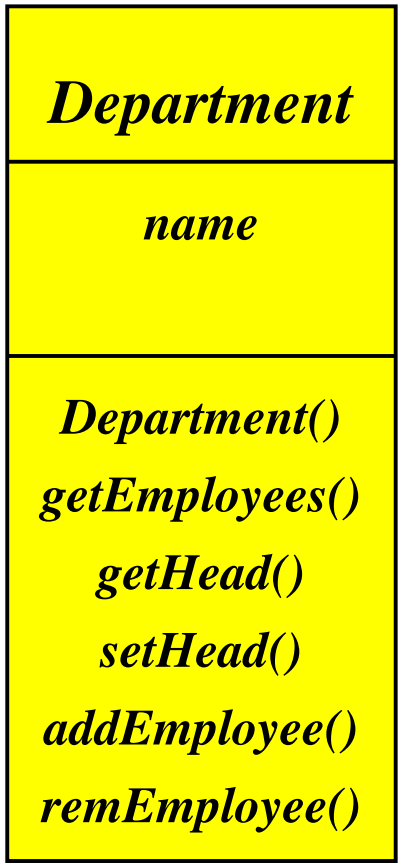
- Eine glatte Linie mit einem Pfeil verbindet einen Akteur mit einem Use-Case. Das bedeutet, daß die mit dem Use-Case verbundene Prozedur von diesem Akteur angestoßen bzw. durchgeführt wird.
- Gestrichelte Linien repräsentieren Beziehungen zwischen mehreren Prozeduren. Damit können Gemeinsamkeiten hervorgehoben werden.
- Wichtig: Pfeile repräsentieren **keine** Flußrichtungen von Daten. Es führt hier insbesondere kein Pfeil zu dem Bankbeamten zurück.
- Bei neueren UML-Versionen fallen die Pfeile weg, weil sie letztlich redundant sind.


Uhr einstellen
Weckzeit einstellen
Wecken
Schläfer
Systemuhr

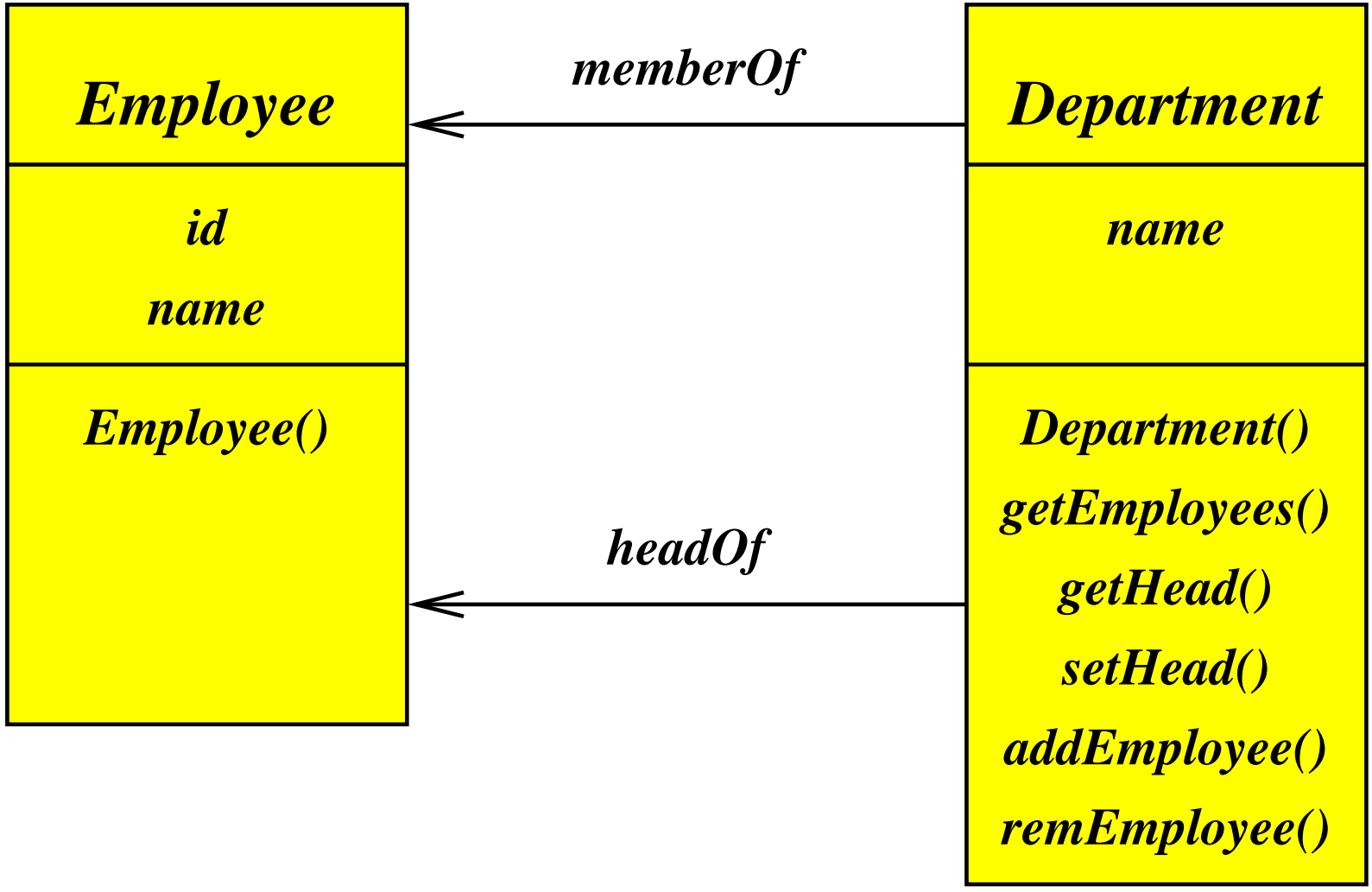
- Es können auch Pfeile von Prozeduren zu Akteuren gehen, wenn sie eine Benachrichtigung repräsentieren, die sofort wahrgenommen wird.
- Ein Wecker hat intern einen Akteur — die Systemuhr. Sie aktualisiert laufend die Zeit und muß natürlich eine Neu-Einstellung der Zeit sofort erfahren.
- Das Auslösen des Wecksignals wird von der Systemuhr als Akteur vorgenommen. Diese Prozedur führt (hoffentlich) dazu, daß der Schläfer geweckt wird. In diesem Falle ist es berechtigt, auch einen Pfeil von einer Prozedur zu einem menschlichen Akteur zu ziehen.


Employee
id
name
Employee()
memberOf
*
1..*
Department
name
Department()
getEmployees()
getHead()
setHead()
addEmployee()
remEmployee()
headOf
0..1
*

- Klassen-Diagramme bestehen aus Klassen (dargestellt als Rechtecke) und deren Beziehungen (Linien und Pfeile) untereinander.
- Bei größeren Projekten sollte nicht der Versuch unternommen werden, alle Details in ein großes Diagramm zu integrieren. Stattdessen ist es sinnvoller, zwei oder mehr Ebenen von Klassen-Diagrammen zu haben, die sich entweder auf die Übersicht oder die Details in einem eingeschränkten Bereich konzentrieren.

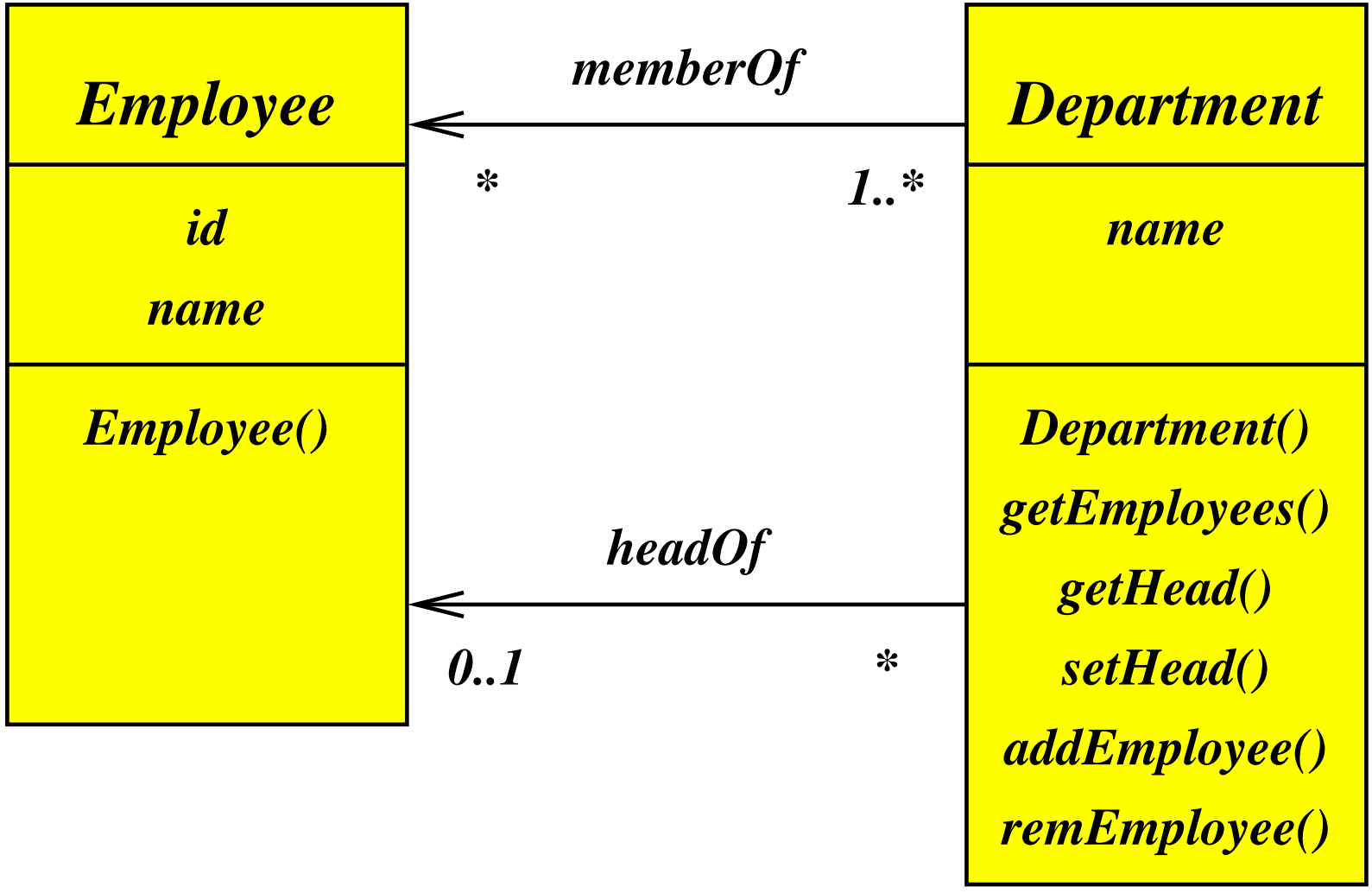


- Die Rechtecke für eine Klasse spezifizieren den Namen der Klasse und die öffentlichen Felder und Methoden. Die erste Methode sollte (sofern vorhanden) der Konstruktor sein.
- Diese Sektionen werden durch horizontale Striche getrennt.
- Bei einem Übersichtsdiagramm ist es auch üblich, nur den Klassennamen anzugeben.
- Private Felder und private Methoden werden normalerweise weggelassen. Eine Ausnahme ist nur angemessen, wenn eine Dokumentation für das Innenleben einer Klasse angefertigt wird, wobei dann auch nur das Innenleben einer einzigen Klasse gezeigt werden sollte.


Employee
id
name
Employee()
memberOf
headOf
Department
name
Department()
getEmployees()
getHead()
setHead()
addEmployee()
remEmployee()

- Primär werden bei den dargestellten Beziehungen Referenzen in der Datenstruktur berücksichtigt.
- Referenzen werden mit durchgezogenen Linien dargestellt, wobei ein oder zwei Pfeile die Verweisrichtung angeben.
- In diesem Beispiel kann ein Objekt der Klasse *Department* eine Liste von zugehörigen Angestellten liefern.
- Zusätzlich ist es mit gestrichelten Linien möglich, die Benutzung einer anderen Klasse zum Ausdruck zu bringen. Ein typisches Beispiel ist die Verwendung einer fremden Klasse als Typ in einer Signatur.
- Beziehungen reflektieren Verantwortlichkeiten. Im Beispiel ist die Klasse *Department* für die Beziehungen *memberOf* und *headOf* zuständig, weil die Pfeile von ihr ausgehen.
- Eine Beziehung kann beidseitig mit Pfeilen versehen sein, dann sind beide Klassen dafür verantwortlich.

- Durch die Beziehungen wird typischerweise offenbar, wie eine Navigation durch eine Datenstruktur möglich ist, d.h. ist beginnend von einem Objekt einer Klasse eine Traverse über weitere Objekte möglich auf Basis der vorhandenen Beziehungen.
- Pfeilrichtungen werden in diesem Sinne gerne als potentielle Navigationsrichtungen interpretiert, d.h. die Klasse, von der aus ein Pfeil zu einer anderen Klasse ausgeht, sollte auch Methoden anbieten, mit der eine entsprechende Abfrage oder Traverse möglich ist.
- Eine Beziehung ohne Pfeile sagt nichts zu Verantwortlichkeiten oder Navigierbarkeit aus.
- Wenn Pfeile verwendet werden, sollten diese vollständig sein. Es erscheint wenig sinnvoll, nur die eine Richtung anzugeben, wenn sich eine Beziehung auch in der anderen Richtung navigieren lässt. Der UML-Standard lässt dies zu und verwendet stattdessen ein „x“ als Markierung für Nicht-Navigierbarkeit. Wir verzichten darauf.


Employee
id
name
Employee()
memberOf
*
1..*
Department
name
Department()
getEmployees()
getHead()
setHead()
addEmployee()
remEmployee()
headOf
0..1
*

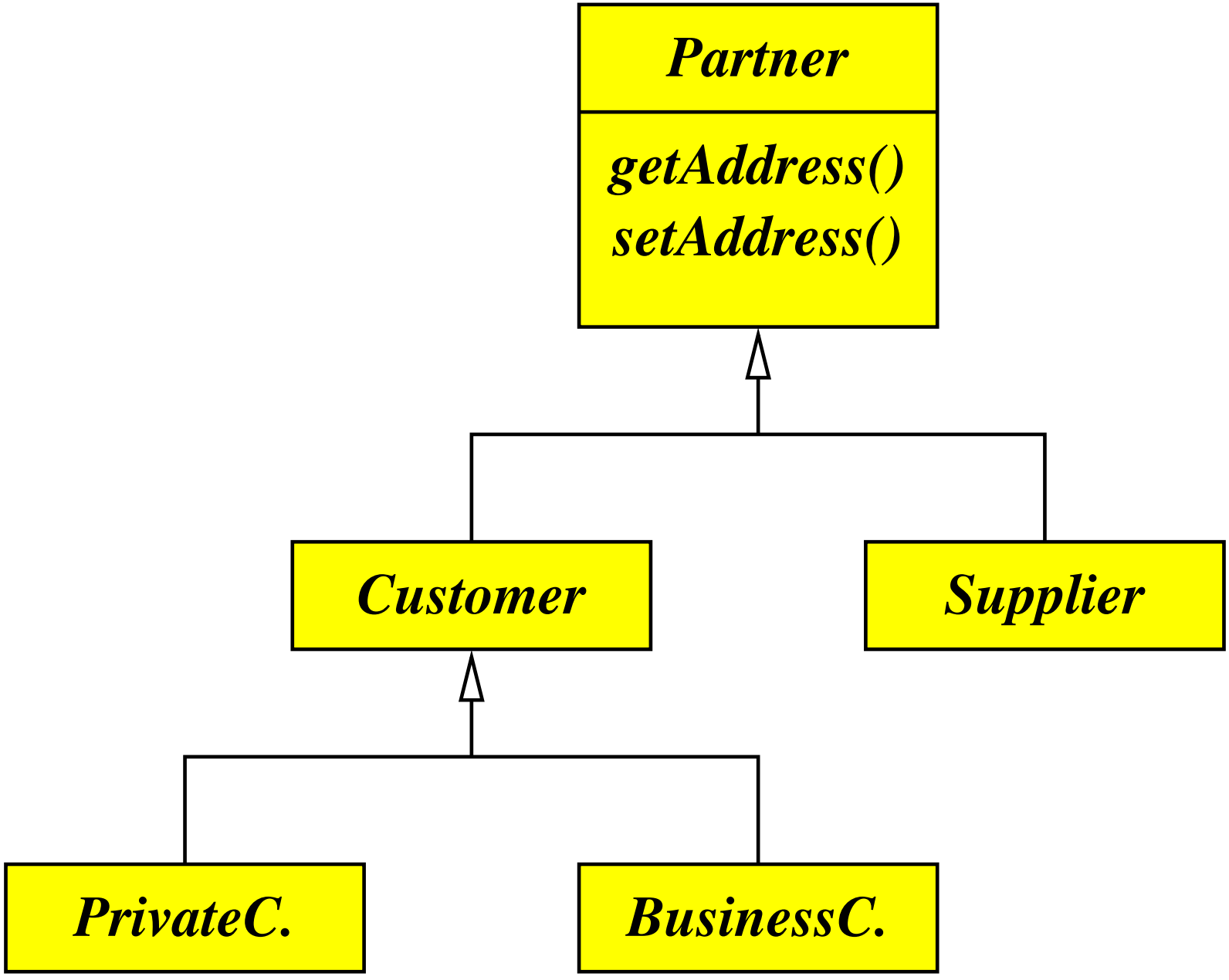
Komplexitätsgrade

- Komplexitätsgrade spezifizieren jeweils aus der Sicht eines *einzelnen* Objekts, wieviele konkrete Beziehungen zu Objekten der anderen Klasse existieren können.
- Ein Komplexitätsgrad wird in Form eines Intervalls angegeben (z.B. "0..1"), in Form einer einzelnen Zahl oder mit "*" als Kurzform für 0 bis unendlich.
- Für jede Beziehung werden zwei Komplexitätsgrade angegeben, jeweils aus Sicht eines Objekts der beiden beteiligten Klassen.
- In diesem Beispiel hat eine Abteilung gar keinen oder einen Leiter, aber ein Angestellter kann für beliebig viele Abteilungen die Rolle des Leiters übernehmen.

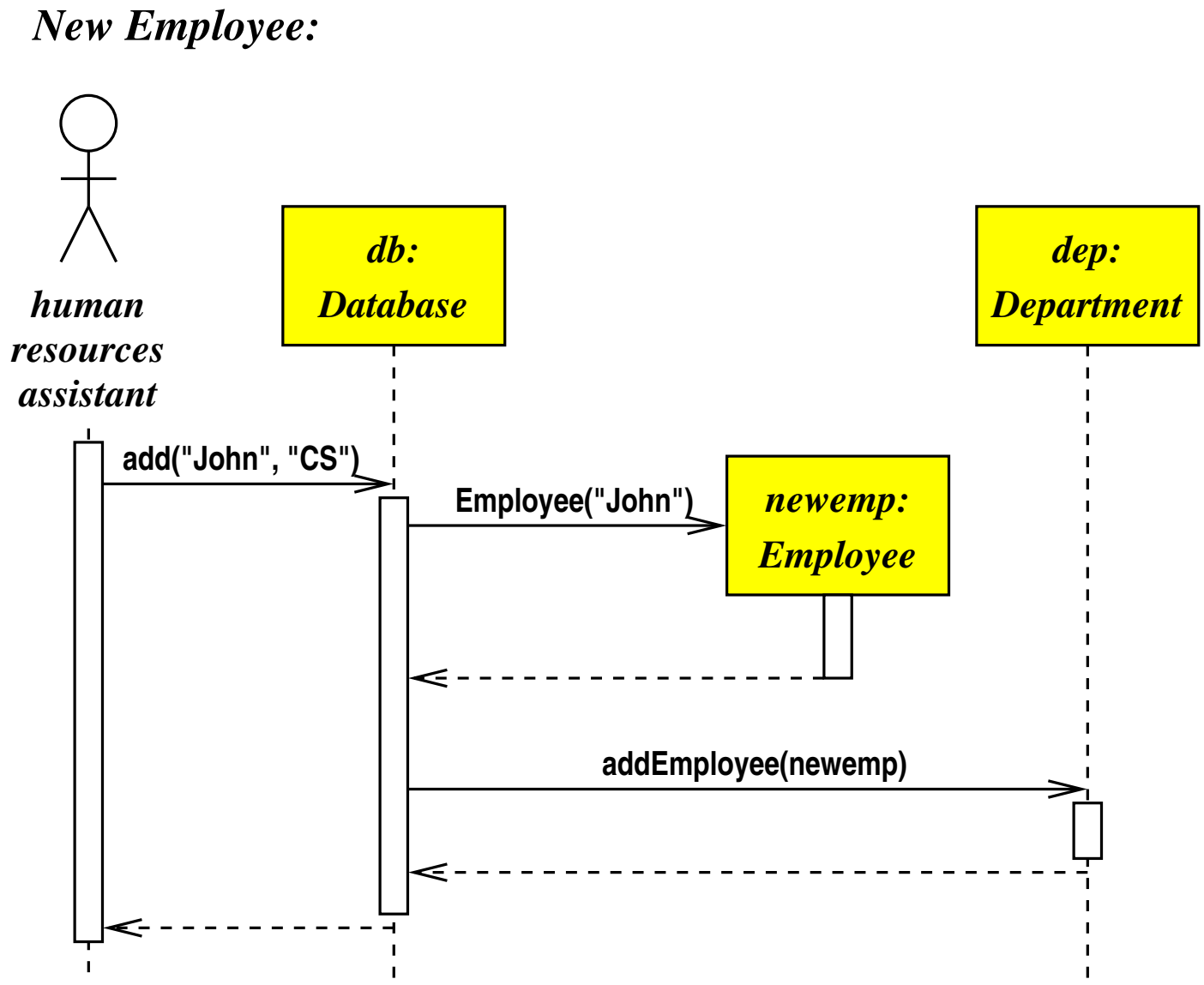
Bei der Implementierung ist der Komplexitätsgrad am Pfeilende relevant:

- Ein Komplexitätsgrad von 1 wird typischerweise durch eine private Referenz, die auf ein Objekt der anderen Klasse zeigt, repräsentiert. Dieser Zeiger muß dann immer wohldefiniert sein und auf ein Objekt zeigen.
- Bei einem Grad von 0 oder 1 darf der Zeiger auch *NULL* sein.
- Bei "*" werden Listen oder andere geeignete Datenstrukturen benötigt, um alle Verweise zu verwalten. Solange für die Listen vorhandene Sprachmittel oder Standard-Bibliotheken für Container verwendet werden, werden sie selbst nicht in das Klassendiagramm aufgenommen.
- Im Beispiel hat die Klasse *Department* einen privaten Zeiger *head*, der entweder *NULL* ist oder auf einen *Employee* zeigt.
- Für die Beziehung *memberOf* wird hingegen bei der Klasse *Department* eine Liste benötigt.

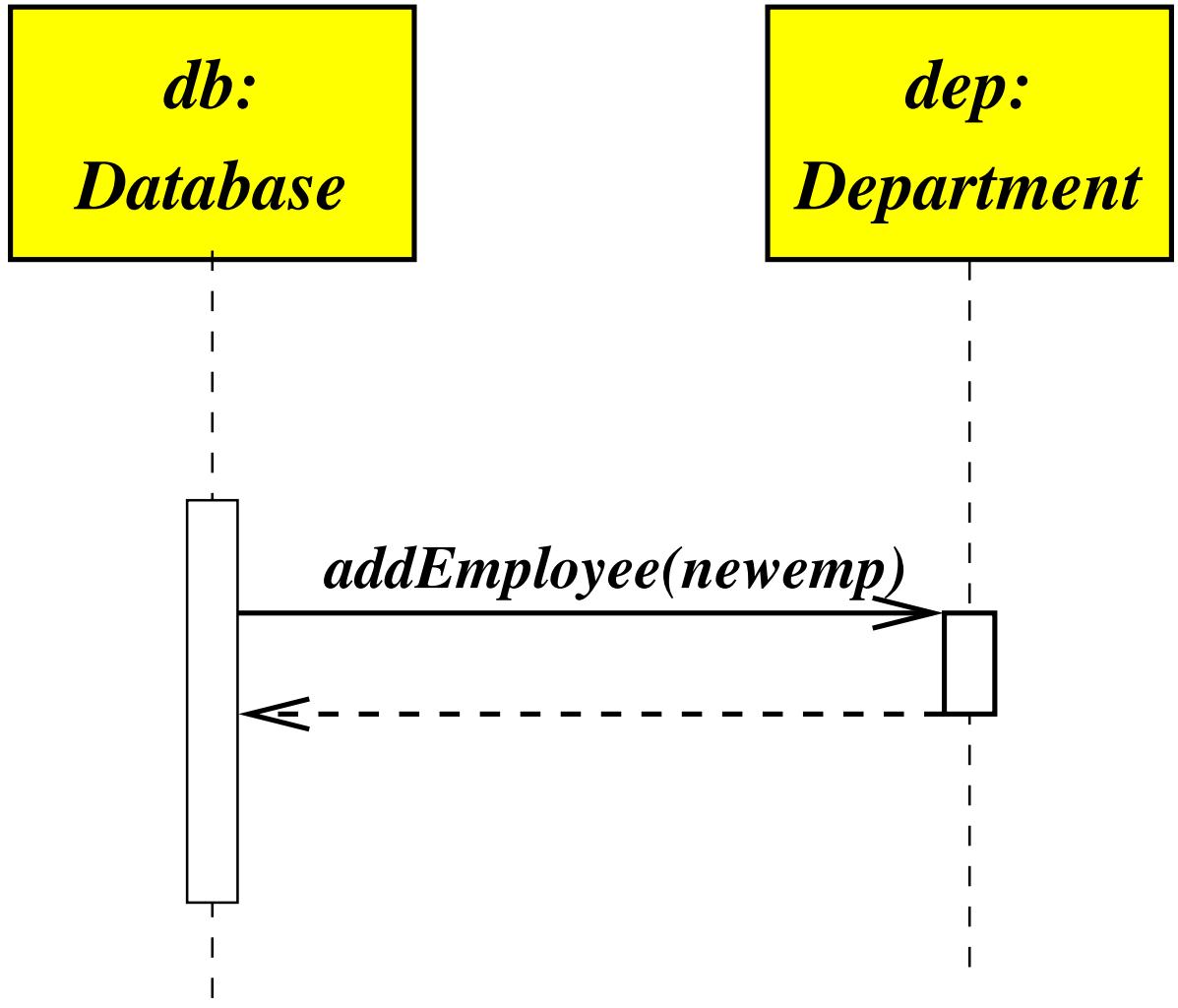
- Auch der Komplexitätsgrad am Anfang des Pfeiles ist relevant, da er angibt, wieviel Verweise insgesamt von Objekten der einen Klasse auf ein einzelnes Objekt der anderen Klasse auftreten können.
- Im Beispiel muß jeder Angestellte in mindestens einer Abteilung aufgeführt sein. Er darf aber auch in mehreren Abteilungen beheimatet sein.
- Um die Konsistenz zu bewahren, darf der letzte Verweis einer Abteilung zu einem Angestellten nicht ohne weiteres gelöscht werden. Dies ist nur zulässig, wenn auch gleichzeitig der Angestellte gelöscht wird oder in eine andere Abteilung aufgenommen wird.
- Die Klasse, von der ein Pfeil ausgeht, ist üblicherweise für die Einhaltung der zugehörigen Komplexitätsgrade verantwortlich.


Partner
getAddress()
setAddress()
Customer
Supplier
PrivateC.
BusinessC.

- Dieses Beispiel zeigt eine einfache Klassen-Hierarchie, bei der *Customer* und *Supplier* Erweiterungen von *Partner* sind. *Customer* ist wiederum eine Verallgemeinerung von *PrivateCustomer* und *BusinessCustomer*.
- Alle Erweiterungen erben die Methoden *getAddress()* und *setAddress()* von der Basis-Klasse.
- Dieser Entwurf erlaubt es, Kontakt-Adressen verschiedener Sorten von Partnern in einer Liste zu verwalten. Damit bleibt z.B. der Ausdruck von Adressen unabhängig von den vorhandenen Ausprägungen.

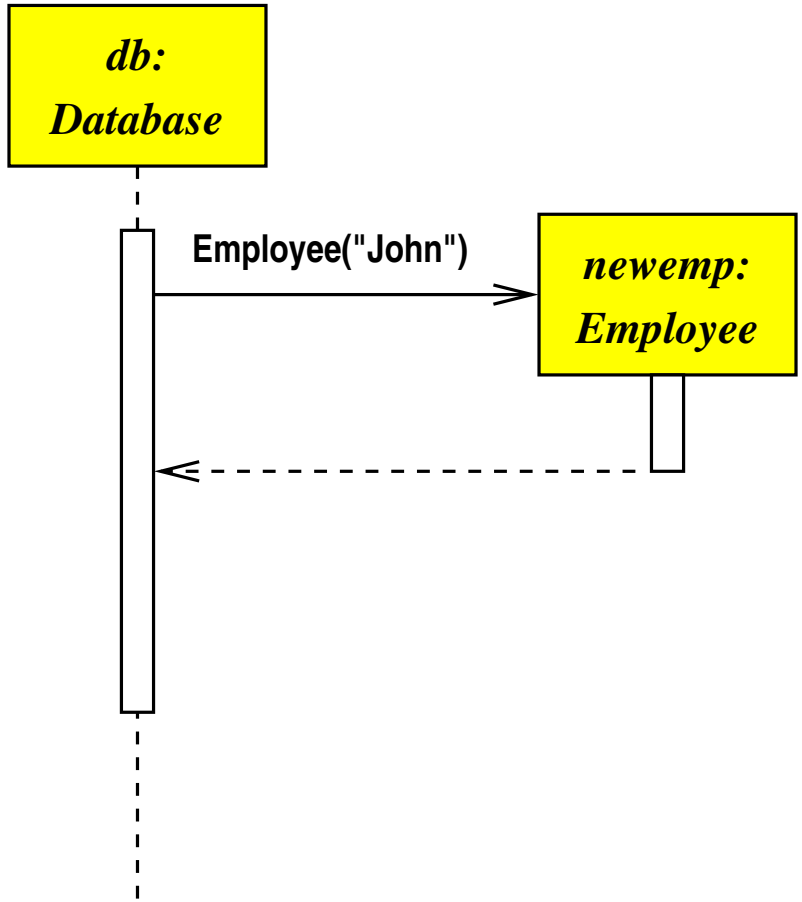

New Employee:
human resources assistant
db: Database
dep: Department
add("John", "CS")
Employee("John")
newemp: Employee
addEmployee(newemp)

- Sequenz-Diagramme zeigen den Kontrolfluß für ausgewählte Szenarien.
- Die Szenarien können unter anderem von den Use-Cases abgeleitet werden.
- Sie demonstrieren, wie Akteure und Klassen miteinander in einer sequentiellen Form operieren.
- Insbesondere wird die zeitliche Abfolge von Methodenaufrufen für einen konkreten Fall dokumentiert.
- Sequenz-Diagramme helfen dabei zu ermitteln, welche Methoden bei den einzelnen Klassen benötigt werden, um eine Funktionalität umzusetzen.


db:
Database
dep:
Department
addEmployee(newemp)

- Die Zeitachse verläuft von oben nach unten.
- Jedes an einem Szenario beteiligte Objekt wird durch ein Rechteck dargestellt, das die Klassenbezeichnung und optional einen Variablennamen enthält.
- Die Zeiträume, zu denen ein Objekt nicht aktiv ist, werden mit einer gestrichelten Linie dargestellt.
- Ein Objekt wird dann durch einen Methodenaufruf aktiv. Der Zeitraum, zu dem sich eine Methode auf dem Stack befindet, wird durch langgezogenes Rechteck dargestellt.
- Der Methodenaufruf selbst wird durch einen Pfeil dargestellt, der mit dem Aufruf selbst beschriftet wird.
- Die Rückkehr kann entweder weggelassen werden oder sollte durch eine gestrichelte Linie markiert werden.

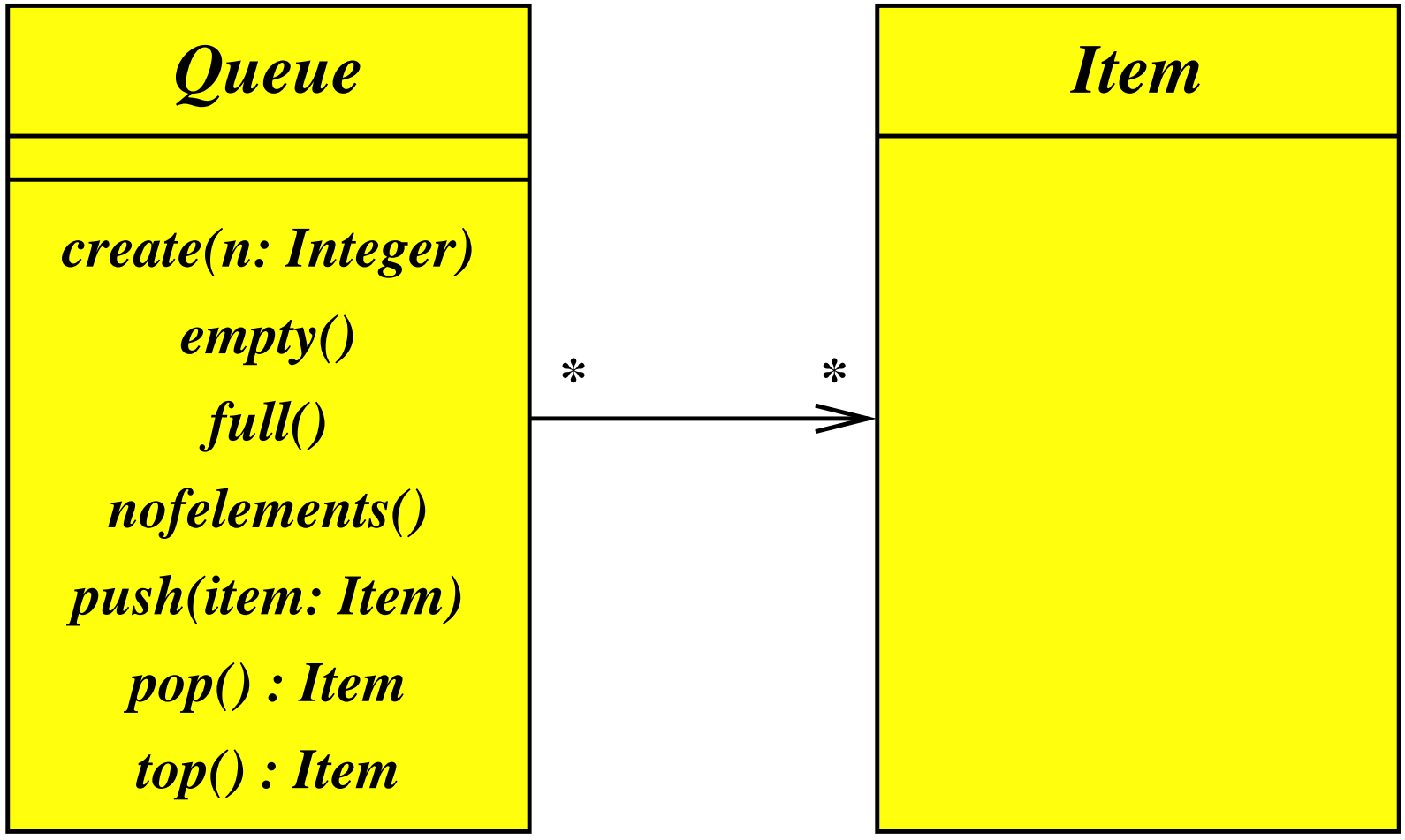
Konstruktoren in einem Sequenz-Diagramm



- Objekte, die erst im Laufe des Szenarios durch einen Konstruktor erzeugt werden, werden rechts neben dem Pfeil plaziert.
- Ganz oben stehen nur die Objekte, die zu Beginn des Szenarios bereits existieren.
- Da ein neu erzeugtes Objekt sofort aktiv ist, gibt es keine gestrichelte Linie zwischen dem Objekt und der ersten durch ein weißes Rechteck dargestellten Aktivitätsphase.

- Der Begriff des Vertrags (*contract*) in Verbindung von Klassen wurde von Bertrand Meyer in seinem Buch »Object-oriented Software Construction« und in seinen vorangegangenen Artikeln geprägt.
- Die Idee selbst basiert auf frühere Arbeiten über die Korrektheit von Programmen von Floyd, Hoare und Dijkstra.
- Wenn wir die Schnittstelle einer Klasse betrachten, haben wir zwei Parteien, die einen Vertrag miteinander abschließen:
  - Die Klienten, die die Schnittstelle nutzen, und
  - die Implementierung selbst mitsamt all den Implementierungen der davon abgeleiteten Klassen.

- Dieser Vertrag sollte explizit in formaler Weise im Rahmen des Designs einer Klasse spezifiziert werden. Er besteht aus:
  - Vorbedingungen (*preconditions*), die spezifizieren, welche Vorraussetzungen zu erfüllen sind, bevor eine Methode aufgerufen werden darf.
  - Nachbedingungen (*postconditions*), die spezifizieren, welche Bedingungen nach dem Aufruf der Methode erfüllt sein müssen.
  - Klasseninvarianten, die Bedingungen spezifizieren, die von allen Methoden jederzeit aufrecht zu halten sind.


Queue
create(n: Integer)
empty()
full()
nofelements()
push(item: Item)
pop() : Item
top() : Item
*
*
Item

Vertrag einer Warteschlange (*Queue*)

Methode	Vorbedingung	Nachbedingung
create()	$n > 0$	*empty*()&& *nofelements*()== 0
push()	!*full*()	!*empty*()&& *nofelements*() erhöht sich um 1
pop()	!*empty*()	*nofelements*() verringert sich um 1; beim i-ten Aufruf ist das i-te Objekt, das *push*() übergeben worden ist, zurückzuliefern.
top()	!*empty*()	*nofelements*() bleibt unverändert; liefert das Objekt, das auch bei einem nachfolgenden Aufruf von *pop*() geliefert werden würde

Klassen-Invarianten:

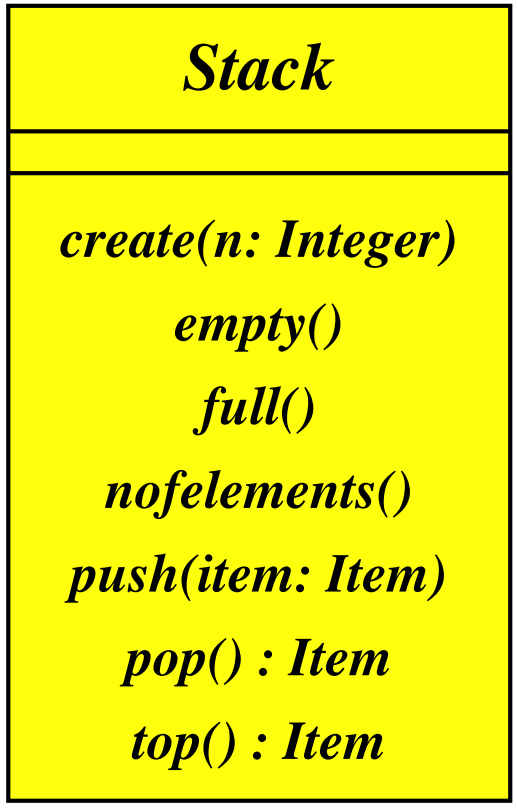
- *nofelements*()== 0 && *empty*()|| *nofelements*()> 0 && !*empty*()
- *empty*()&& !*full*()|| *full*()&& !*empty*()|| !*full*()&& !*empty*()
- *nofelements*()>= *n* || !*full*()

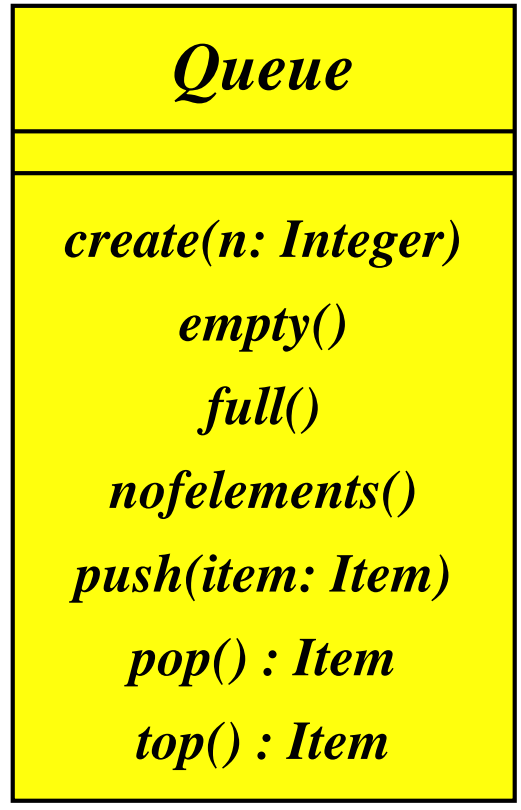
```
void Queue::push(const Item& item) {
   // precondition
   assert(!full());
   // prepare to check postcondition
   int before = nofelements();

   // ... adding item to the queue ...

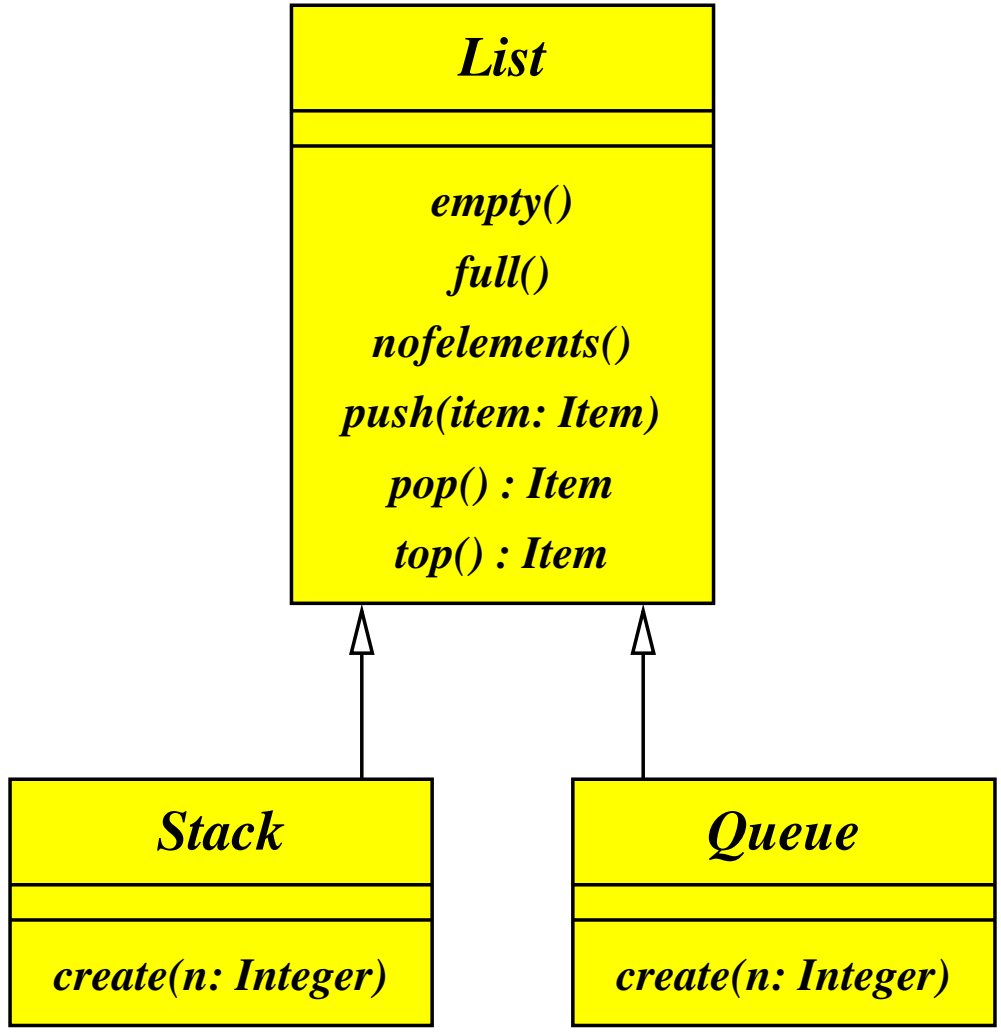
   // checking postcondition
   int after = nofelements();
   assert(!empty() && after == before + 1);
}
```

- Teile des Vertrags können in Assertions verwandelt werden, die in die Implementierung einer Klasse aufzunehmen sind.
- Dies erleichtert das Finden von Fehlern, die aufgrund von Vertragsverletzungen entstehen.
- Der Verlust an Laufzeiteffizienz durch Assertions ist vernachlässigbar, solange diese nicht im übertriebenen Maße eingesetzt werden. (Das liegt u.a. auch daran, dass die Überprüfungen häufig parallelisiert ausgeführt werden können dank der Pipelining-Architektur moderner Prozessoren.)





- Signaturen alleine spezifizieren noch keine Klasse.
- Die gleiche Signatur kann mit verschiedenen Semantiken und entsprechend unterschiedlichen Verträgen assoziiert werden.

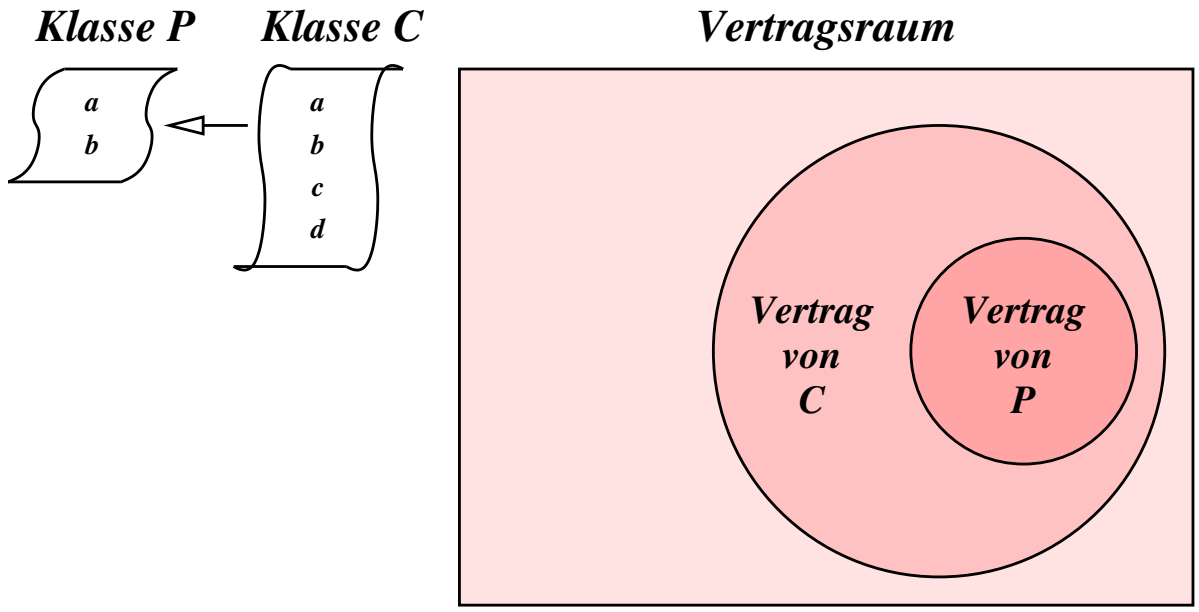

List
empty()
full()
nofelements()
push(item: Item)
pop() : Item
top() : Item
Stack
create(n: Integer)
Queue
create(n: Integer)

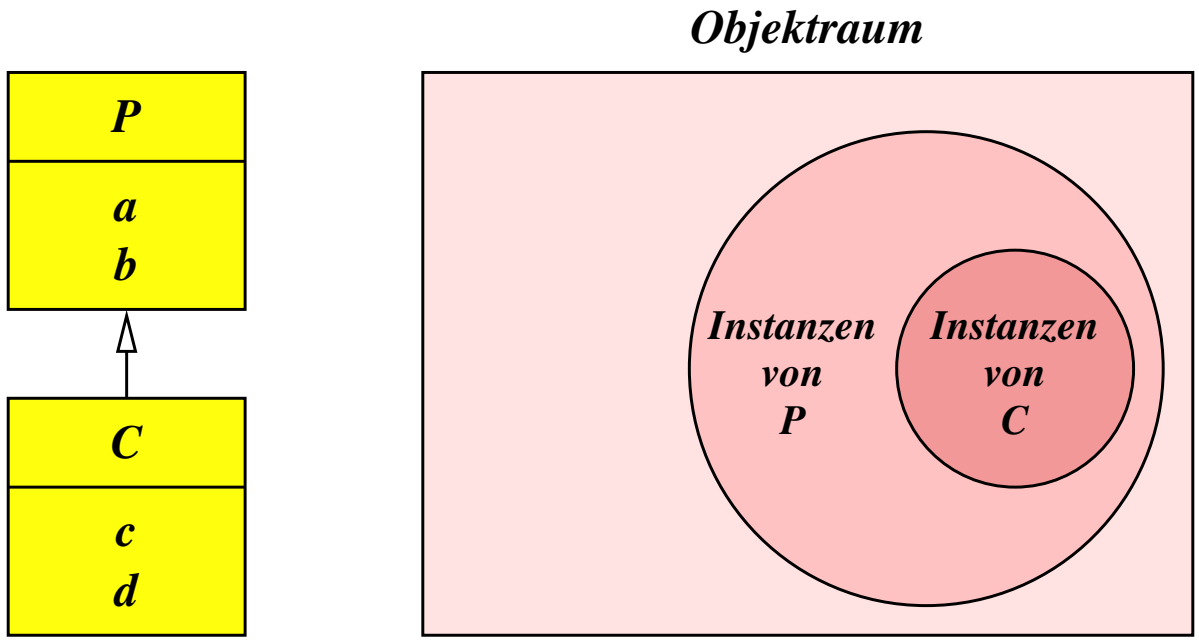
- Einige Klienten benötigen nur allgemeine Listen, die alles akzeptieren, was ihnen gegeben wird. Diese Klienten interessieren sich nicht für die Reihenfolge, in der die Listenelemente später entnommen werden.
- Der Vertrag für *List* spezifiziert, dass *pop()* bei Einhalten der Vorbedingung einer nicht-leeren Liste irgendein zuvor eingefügtes Element zurückliefert, das bislang noch nicht zurückgegeben worden ist. Die Reihenfolge selbst bleibt undefiniert.
- *Queue* erweitert diesen Vortrag dahingehend, dass als Ordnung die ursprüngliche Reihenfolge des Einfügens gilt (FIFO).
- *Stack* hingegen erweitert diesen Vertrag mit der Spezifikation, dass *pop()* das zuletzt eingefügte Element zurückzuliefern ist, das bislang noch nicht zurückgegeben wurde (LIFO).
- Erweiterungen sind jedoch verpflichtet, in jedem Falle den Vertrag der Basisklasse einzuhalten. Entsprechend dürfen Verträge nicht durch Erweiterungen abgeschwächt werden.
- Die Einhaltung dieser Regel stellt sicher, dass ein Objekt des Typs *Stack* überall dort verwendet werden darf, wo ein Objekt des Typs *List* erwartet wird.

- Vererbung ist im Bereich der OO-Techniken eine Beziehung zwischen Klassen, bei denen eine *abgeleitete Klasse* den Vertrag mitsamt allen Signaturen von einer *Basisklasse* übernimmt.
- Da in der Mehrzahl der OO-Sprachen Klassen mit Typen kombiniert sind, hat die Vererbung zwei Auswirkungen:
  - Kompatibilität: Instanzen der abgeleiteten Klasse dürfen überall dort verwendet werden, wo eine Instanz der Basisklasse erwartet wird.
  - Gemeinsamer Programmtext: Die Implementierung der Basisklasse kann teilweise von der abgeleiteten Klasse verwendet werden. Dies wird für jede Methode einzeln entschieden. Einige OO-Sprachen (einschließlich C++) ermöglichen den gemeinsamen Zugriff auf ansonsten private Datenfelder zwischen der Basisklasse und der abgeleiteten Klasse.

- Die Komplexität dieser Beziehung kann 1:* (*einfache Vererbung*) oder *:* (*mehrfache Vererbung*) sein.
- C++ unterstützt mehrfache Vererbungen.
- Java unterstützt nur einfache Vererbungen, bietet aber zusätzlich das typen-orientierte Konzept von Schnittstellen an.
- In C++ kann die Schnittstellen-Technik von Java auf Basis sogenannter abstrakter Klassen erreicht werden. In diesem Falle übernehmen Basisklassen ohne zugehörige Implementierungen die Rolle von Typen.

Vertragsraum vs. Objektraum





Vorteile der Vererbung

- Erweiterbarkeit / Polymorphismus: Neue Funktionalität kann hinzugefügt werden, ohne bestehende Klassen zu verändern, solange die neuen Klassen sich als Erweiterung bestehender Klassen formulieren lassen.
- Wiederverwendbarkeit: Für eine Serie ähnlicher Anwendungen kann ein Kern an Klassen definiert werden (*framework*), die jeweils anwendungsspezifisch erweitert werden.
- Verbergung (*information hiding*): Je allgemeiner eine Klasse ist, umso mehr verbirgt sie vor ihren Klienten. Je mehr an Implementierungsdetails verborgen bleibt, umso seltener sind Klienten von Änderungen betroffen und der Programmtext des Klienten bleibt leichter verständlich, weil die vom Leser zu verinnerlichenden Verträge im Umfang geringer sind.

Wann ist Vererbung einzusetzen? 72

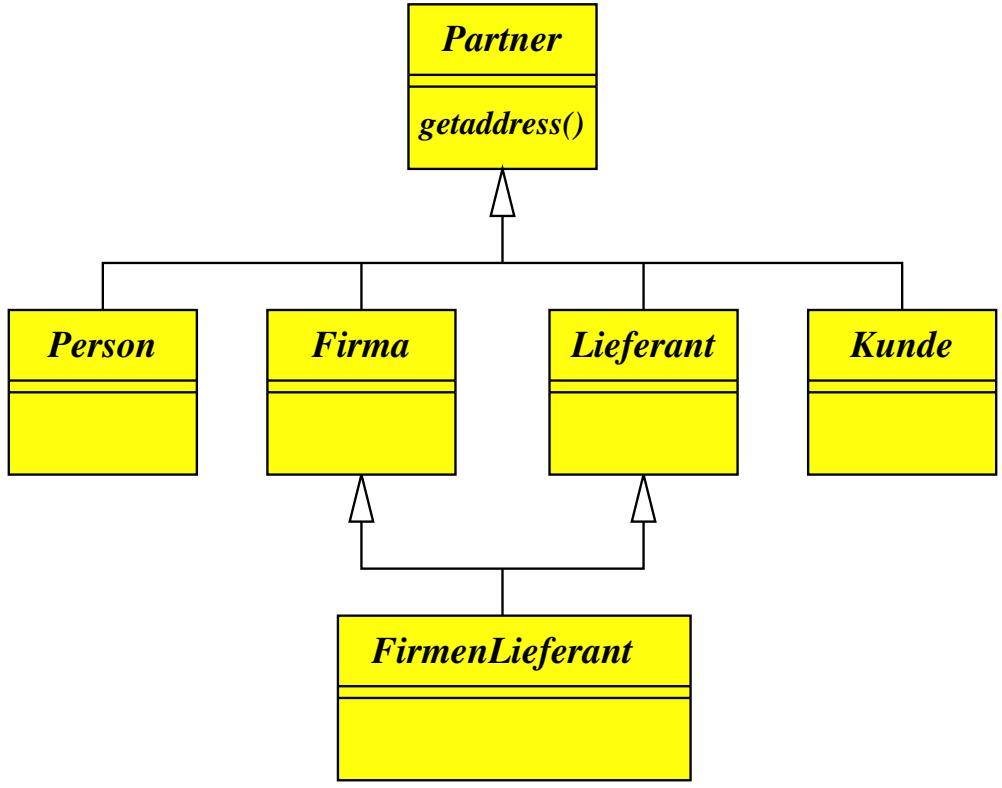
Vererbung sollte genutzt werden, wenn

- mehrere Implementierungen mit einer gemeinsamen Schnittstelle auskommen können,
- Rahmen (*frameworks*) für individuelle Erweiterungen (*plugins*) sinnvoll sind und wenn
- sie zur Schaffung einer sinnvollen Typhierarchie dient, die die statische Typsicherheit erhöht.

Vererbung sollte **nicht** genutzt werden, um

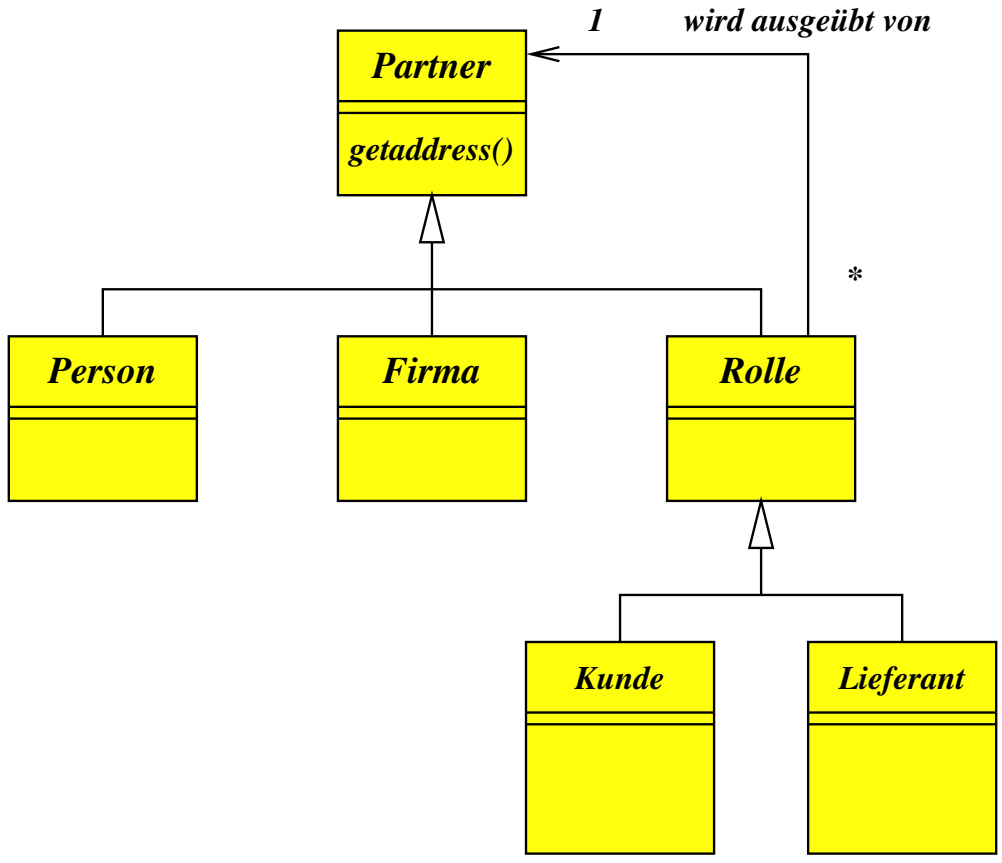
- bereits existierenden Programmtext wiederzuverwenden, wenn es sich dabei nicht um eine strikte *is-a*-Beziehung im Sinne einer sauberen Vertragshierarchie handelt oder um
- Objekte in ein hierarchisches Klassensystem zu zwängen, wenn diese bzw. deren zugehörigen realen Objekte die Einordnung im Laufe ihrer Lebenszeit verändern können.

Beispiel für schlechte Vererbung



- Die Rollenverteilung (z.B. als Lieferant oder Kunde) ist statisch und die Zahl der Kombinationsmöglichkeiten (und der entsprechend zu definierenden Klassen) explodiert.

Objekte mit unterschiedlichen Rollen



- Ein Partner-Objekt kann während seiner Lebenszeit sowohl die Rolle eines Kunden oder auch eines Lieferanten übernehmen.
- Dieses Pattern entspricht dem Decorator-Pattern aus dem Werk von Gamma et al.

- Bjarne Stroustrup startete sein Projekt *C with Classes* im April 1979 bei den Bell Laboratories nach seinen Erfahrungen mit Simula und BCPL.
- Sein Ziel war es, die Klassen von Simula als Erweiterung zur Programmiersprache C einzuführen, ohne Laufzeiteffizienz zu opfern. Der Übersetzer wurde als Präprozessor zu C implementiert, der *C with Classes* in reguläres C übertrug.
- 1982 begann ein Neuentwurf der Sprache, die dann den Namen C++ erhielt. Im Rahmen des Neuentwurfs kamen virtuelle Funktionen (und damit Polymorphismus), die Überladung von Operatoren, Referenzen, Konstanten und verbesserte Typüberprüfungen hinzu.

- 1985 begann Bell Laboratories mit der Auslieferung von *Cfront*, der C++ in C übersetzte und damit eine Vielzahl von Plattformen unterstützte.
- 1990 wurde für C++ bei ANSI/ISO ein Standardisierungskomitee gegründet.
- Vorschläge für Templates in C++ gab es bereits in den 80er-Jahren und eine erste Implementierung stand 1989 zur Verfügung. Sie wurde 1990 vom Standardisierungskomitee übernommen.
- Analog wurden Ausnahmenbehandlungen 1990 vom Standardisierungskomitee akzeptiert. Erste Implementierungen hierfür gab es ab 1992.
- Namensräume wurden erst 1993 in C++ eingeführt.
- Im September 1998 wurde mit ISO 14882 der erste Standard für C++ veröffentlicht. Die aktuelle Fassung des Standards ist von August 2011 und wird kurz C++11 genannt.

Greeting.hpp

```
#ifndef GREETING_H
#define GREETING_H

class Greeting {
   public:
      void hello();
      void hi();
}; // class Greeting

#endif
```

- Klassendeklarationen (mitsamt allen öffentlichen und auch privaten Datenfeldern und Methoden) sind in Dateien, die mit ».hpp« oder ».h« enden, unterzubringen. Hierbei steht ».h« für Header-Datei bzw. ».hpp« Header-Datei von C++.
- Alle Zeilen, die mit einem # beginnen, enthalten Direktiven für den Makro-Präprozessor. Dieses Relikt aus Assembler- und C-Zeiten ist in C++ erhalten geblieben. Die Konstruktion in diesem Beispiel stellt sicher, dass die Klassendeklaration nicht versehentlich mehrfach in den zu übersetzenden Text eingefügt wird.

Greeting.hpp

```
class Greeting {
   public:
      void hello();
}; // class Greeting
```

- Eine Klassendeklaration besteht aus einem Namen und einem Paar geschweifter Klammern, die eine Sequenz von Deklarationen eingrenzen. Die Klassendeklaration wird (wie sonst alle anderen Deklarationen in C++ auch) mit einem Semikolon abgeschlossen.
- Kommentare starten mit »//« und erstrecken sich bis zum Zeilenende.

Greeting.hpp

```
class Greeting {
   public:
      void hello();
}; // class Greeting
```

- Die Deklarationen der einzelnen Komponenten einer Klasse, in der C++-Terminologie *member* genannt, fallen in verschiedene Kategorien, die die Zugriffsrechte regeln:

private	nur für die Klasse selbst und ihre Freunde zugänglich
protected	offen für alle davon abgeleiteten Klassen
public	uneingeschränkter Zugang

  Wenn keine der drei Kategorien explizit angegeben wird, dann wird automatisch **private** angenommen.

Greeting.hpp

```
class Greeting {
   public:
      void hello();
}; // class Greeting
```

- Alle Funktionen (einschließlich der Methoden einer Klasse) haben einen Typ für ihre Rückgabewerte. Wenn nichts zurückzuliefern ist, dann kann **void** als Typ verwendet werden.
- In Deklarationen folgt jeweils dem Typ eine Liste von durch Kommata getrennten Namen, die mit zusätzlichen Spezifikationen wie etwa () ergänzt werden können.
- Die Angabe () sorgt hier dafür, dass aus *hello* eine Funktion wird, die Werte des Typs **void** zurückliefert, d.h. ohne Rückgabewerte auskommt.

Greeting.cpp

```
#include <iostream>
#include "Greeting.hpp"

void Greeting::hello() {
   std::cout << "Hello, fans of C++!" << std::endl;
} // hello()
```

- C++-Programme bzw. die Implementierung einer öffentlichen Schnittstelle in einer Header-Datei werden üblicherweise in separaten Dateien untergebracht.
- Als Dateiendung sind ».cpp«, ».cc« oder ».C« üblich.
- Letztere Variante ist recht kurz, hat jedoch den Nachteil, dass sie sich auf Dateisystemen ohne Unterscheidung von Klein- und Großbuchstaben nicht von der Endung ».c« unterscheiden lässt, die für C vorgesehen ist.
- Der Übersetzer erhält als Argumente nur diese Dateien, nicht die Header-Dateien.

Greeting.cpp

```
#include <iostream>
#include "Greeting.hpp"

void Greeting::hello() {
   std::cout << "Hello, fans of C++!" << std::endl;
} // hello()
```

- Die Direktive **#include** bittet den Präprozessor um das Einfügen des genannten Textes an diese Stelle in den Eingabetext für den Übersetzer.
- Anzugeben ist ein Dateiname. Wenn dieser in <...> eingeschlossen wird, dann erfolgt die Suche danach nur an Standardplätzen, wozu das aktuelle Verzeichnis normalerweise nicht zählt.
- Wird hingegen der Dateiname in "..." gesetzt, dann beginnt die Suche im aktuellen Verzeichnis, bevor die Standardverzeichnisse hierfür in Betracht gezogen werden.

Greeting.cpp

```
#include <iostream>
#include "Greeting.hpp"

void Greeting::hello() {
   std::cout << "Hello, fans of C++!" << std::endl;
} // hello()
```

- Der eigentliche Übersetzer von C++ liest nicht direkt von der Quelle, sondern den Text, den der Präprozessor zuvor generiert hat.
- Andere Texte, die nicht direkt oder indirekt mit Hilfe des Präprozessors eingebunden werden, stehen dem Übersetzer nicht zur Verfügung.
- Entsprechend ist es strikt notwendig, alle notwendigen Deklarationen externer Klassen in Header-Dateien unterzubringen, die dann sowohl bei den Klienten als auch dem implementierenden Programmtext selbst einzubinden sind.

Greeting.cpp

```
void Greeting::hello() {
   std::cout << "Hello, world!" << std::endl;
} // hello()
```

- Methoden werden üblicherweise außerhalb ihrer Klassendeklaration definiert. Zur Verknüpfung der Methode mit der Klasse wird eine Qualifizierung notwendig, bei der der Klassenname und das Symbol :: dem Methodennamen vorangehen. Dies ist notwendig, da prinzipiell mehrere Klassen in eine Übersetzungseinheit integriert werden können.
- Eine Funktionsdefinition besteht aus der Signatur und einem Block. Ein terminierendes Semikolon wird hier nicht verwendet.
- Blöcke schließen eine Sequenz lokaler Deklarationen, Anweisungen und weiterer verschachtelter Blöcke ein.
- Funktionen dürfen nicht ineinander verschachtelt werden.

Greeting.cpp

```
void Greeting::hello() {
   std::cout << "Hello, world!" << std::endl;
} // hello()
```

- Die Präprozessor-Direktive **#include** <iostream> fügte Deklarationen in den zu übersetzenden Text ein, die u.a. auch *cout* innerhalb des Namensraumes *std* deklariert hat. Die Variable *std*::*cout* repräsentiert die Standardausgabe und steht global zur Verfügung.
- Da C++ das Überladen von Operatoren unterstützt, ist es möglich, Operatoren wie etwa << (binäres Verschieben) für bestimmte Typkombinationen zu definieren. Hier wurde die Variante ausgewählt, die als linken Operator einen *ostream* und als rechten Operator eine Zeichenkette erwartet.
- *endl* repräsentiert den Zeilentrenner.
- *cout* << "Hello,␣world!" gibt die Zeichenkette auf *cout* aus, liefert den Ausgabekanal *cout* wieder zurück, wofür der Operator << erneut aufgerufen wird mit der Zeichenkette, die von *endl* repräsentiert wird, so dass der Zeilentrenner ebenfalls ausgegeben wird.

SayHello.cpp

```
#include "Greeting.hpp"

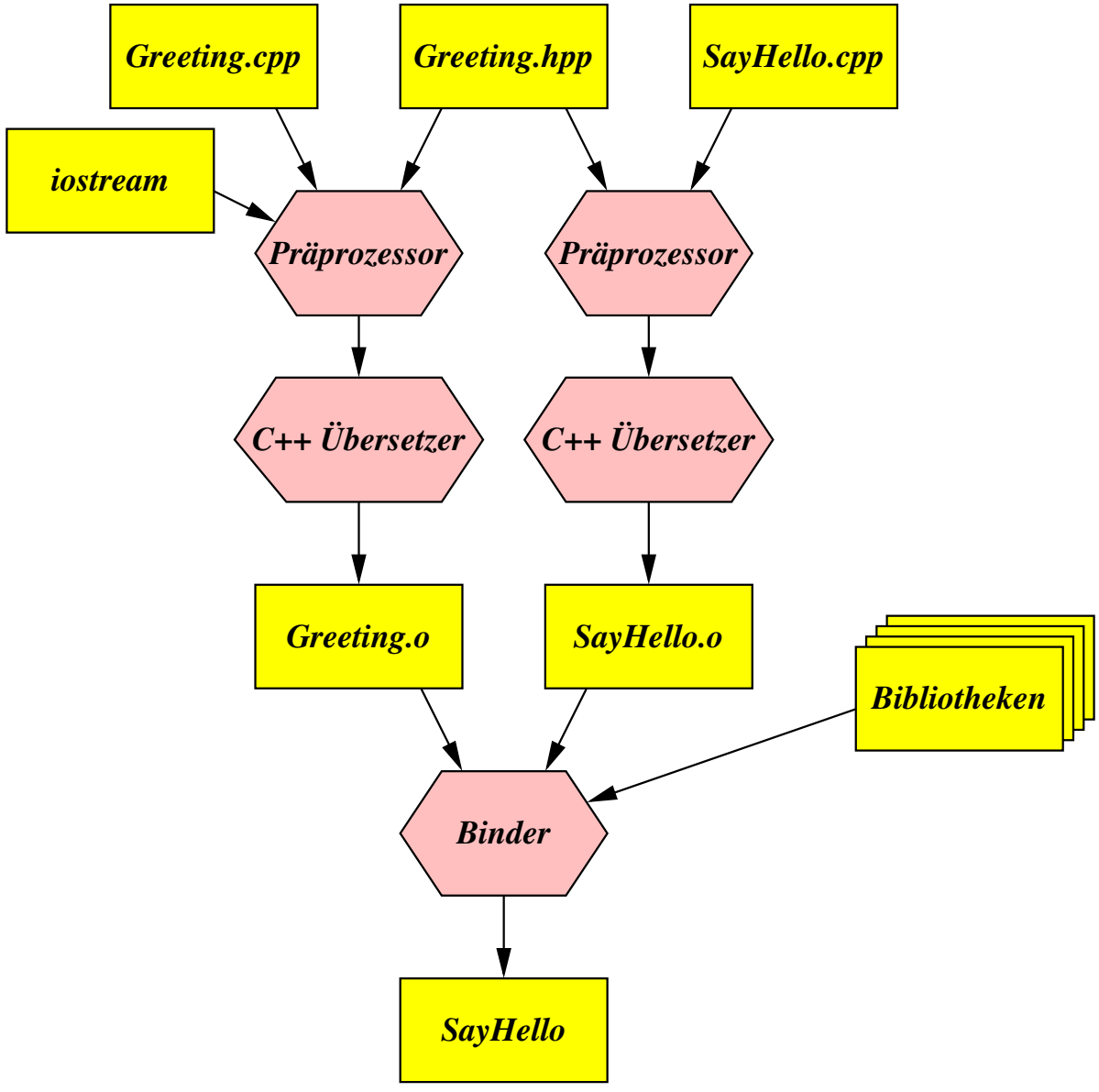
int main() {
   Greeting greeting;
   greeting.hello();
   greeting.hello();
   return 0;
} // main()
```

- Dank dem Erbe von C müssen nicht alle Funktionen einer Klasse zugeordnet werden.
- In der Tat darf die Funktion *main*, bei der die Ausführung (nach der Konstruktion globaler Variablen) startet und die Bestandteil eines jeden Programmes sein muss, nicht innerhalb einer Klasse definiert werden.
- Sobald *main* beendet ist, wird das Ende der gesamten Programmausführung eingeleitet.
- Der ganzzahlige Wert, den *main* zurückgibt, wird der Ausführungsumgebung zurückgegeben. Entsprechend den UNIX-Traditionen steht hier 0 für Erfolg und andere Werte deuten ein Problem an.

SayHello.cpp

```
int main() {
   Greeting greeting;
   greeting.hello();
   return 0;
} // main()
```

- Mit *Greeting greeting* wird eine lokale Variable mit dem Namen *greeting* und dem Datentyp *Greeting* definiert. Das entsprechende Objekt wird hier automatisch instantiiert, sobald *main* startet.
- Durch *greeting.hello*() wird die Methode *hello* für das Objekt *greeting* aufgerufen. Die Klammern sind auch dann notwendig, wenn keine Parameter vorkommen.


Greeting.cpp
Greeting.hpp
SayHello.cpp
iostream
Präprozessor
Präprozessor
C++ Übersetzer
C++ Übersetzer
Greeting.o
SayHello.o
Bibliotheken
Binder
SayHello

- Die gängigen Implementierungen für C++ stellen nur eine schwache Form der Schnittstellensicherheit her.
- Diese wird typischerweise erreicht durch das Generieren von Namen, bei denen teilweise die Typinformation mit integriert ist, so dass Objekte gleichen Namens, jedoch mit unterschiedlichen Typen nicht so ohne weiteres zusammengebaut werden.

```
thales$ ls
Greeting.cpp  Greeting.hpp  SayHello.cpp
thales$ wget --quiet \
> http://www.mathematik.uni-ulm.de/sai/ws12/cpp/cpp/makefile
thales$ sed 's/PleaseRenameMe/SayHello/' <makefile >makefile.tmp &&
> mv makefile.tmp makefile
thales$ make depend
gcc-makedepend  Greeting.cpp SayHello.cpp
thales$ make
g++ -Wall -g -std=gnu++11   -c -o Greeting.o Greeting.cpp
g++ -Wall -g -std=gnu++11   -c -o SayHello.o SayHello.cpp
g++ -o SayHello  Greeting.o SayHello.o
thales$ ./SayHello
Hello, fans of C++!
Hello, fans of C++!
thales$ make realclean
rm -f Greeting.o SayHello.o
rm -f SayHello
thales$
```

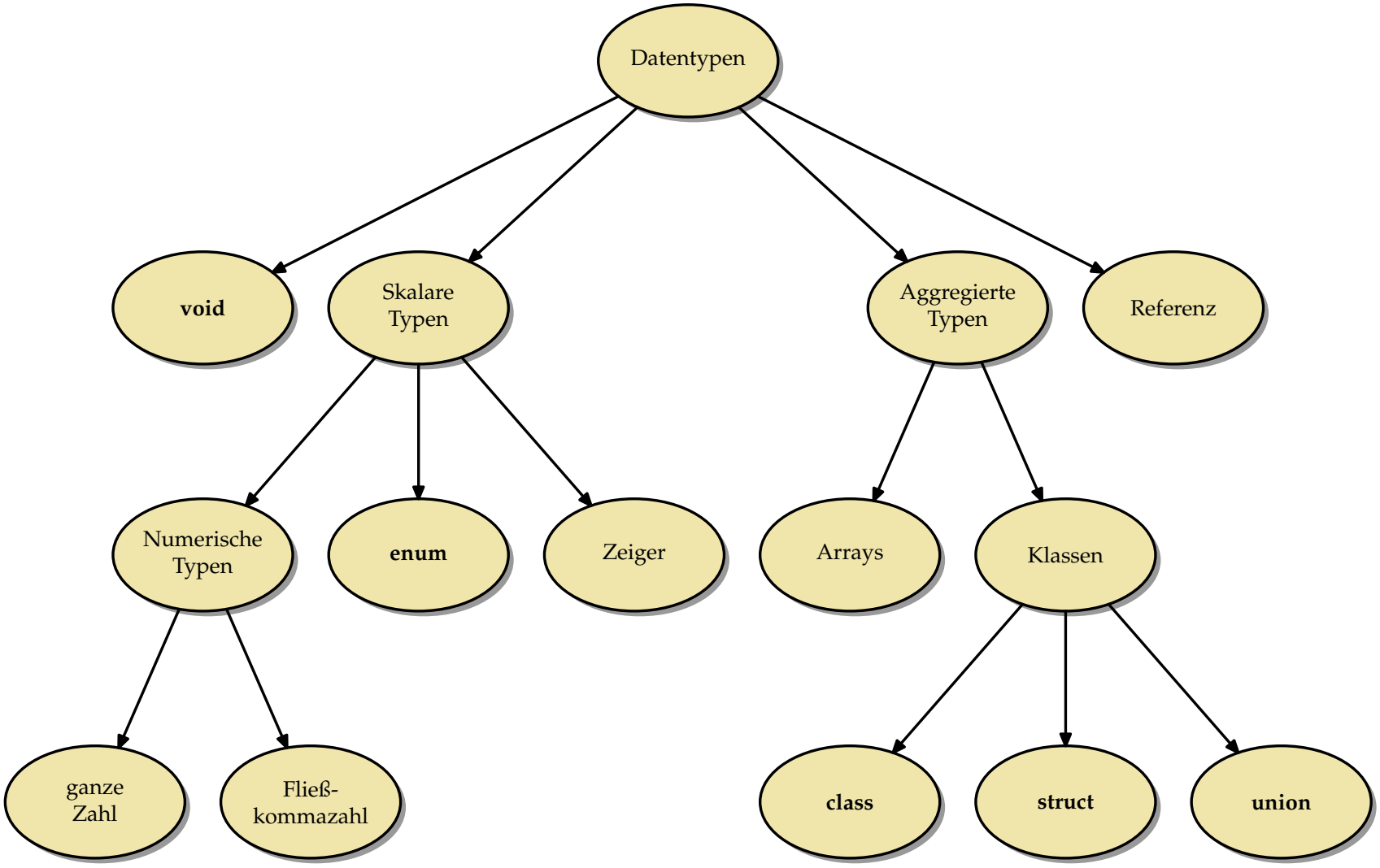
- *make* ist ein Werkzeug, das eine Datei namens *Makefile* (oder *makefile*) im aktuellen Verzeichnis erwartet, in der Methoden zur Generierung bzw. Regenerierung von Dateien beschrieben werden und die zugehörigen Abhängigkeiten.
- *make* ist dann in der Lage festzustellen, welche Zieldateien fehlen bzw. nicht mehr aktuell sind, um diese dann mit den spezifizierten Kommandos neu zu erzeugen.
- *make* wurde von Stuart Feldman 1979 für das Betriebssystem UNIX entwickelt. 2003 wurde er hierfür von der ACM mit dem Software System Award ausgezeichnet.

```
thales$ wget --quiet \
> http://www.mathematik.uni-ulm.de/sai/ws12/cpp/cpp/makefile
thales$ sed 's/PleaseRenameMe/SayHello/' <makefile >makefile.tmp &&
> mv makefile.tmp makefile
```

- Unter der genannten URL steht eine Vorlage für ein für C++ geeignetes *makefile* zur Verfügung.
- Das Kommando *wget* lädt Inhalte von einer gegebenen URL in das lokale Verzeichnis.
- In der Vorlage fehlt noch die Angabe, wie Ihr Programm heißen soll. Das wird hier mit dem Kommando *sed* nachgeholt, indem der Text »PleaseRenameMe« entsprechend ersetzt wird.

```
thales$ make depend
```

- Das heruntergeladene *makefile* geht davon aus, dass Sie den g++ verwenden (GNU C++ Compiler) und die regulären C++-Quellen in ».cpp« enden und die Header-Dateien in ».hpp«.
- Mit dem Aufruf von »make depend« werden die Abhängigkeiten neu bestimmt und im *makefile* eingetragen. Dies muss zu Beginn mindestens einmal aufgerufen werden.
- Wenn Sie dies nicht auf unseren Rechnern probieren, sollten Sie das hier implizit verwendete Skript *gcc-makedepend* von uns klauen. Sie finden es auf einem beliebigen unserer Rechner unter »/usr/local/bin/gcc-makedepend«. Es ist in Perl geschrieben und sollte mit jeder üblichen Perl-Installation zurechtkommen.


Datentypen
void
Skalare Typen
Aggregierte Typen
Referenz
Numerische Typen
enum
Zeiger
Arrays
Klassen
ganze Zahl
Fließ-kommazahl
class
struct
union

- Zu den skalaren Datentypen gehören alle elementaren Typen, die entweder numerisch sind oder sich zu einem numerischen Typ konvertieren lassen.
- Ein Wert eines skalaren Datentyps kann beispielsweise ohne weitere Konvertierung in einer Bedingung verwendet werden.
- Entsprechend wird die 0 im entsprechenden Kontext auch als Null-Zeiger interpretiert oder umgekehrt ein Null-Zeiger ist äquivalent zu *false* und ein Nicht-Null-Zeiger entspricht innerhalb einer Bedingung *true*.
- Ferner liegt die Nähe zwischen Zeigern und ganzen Zahlen auch in der von C++ unterstützten Adressarithmetik begründet.

- Die Spezifikation eines ganzzahligen Datentyps besteht aus einem oder mehreren Schlüsselworten, die die Größe festlegen, und dem optionalen Hinweis, ob der Datentyp vorzeichenbehaftet ist oder nicht.
- Fehlt die Angabe von **signed** oder **unsigned**, so wird grundsätzlich **signed** angenommen.
- Die einzigen Ausnahmen hiervon sind **char** und **bool**.
- Bei **char** darf der Übersetzer selbst eine Voreinstellung treffen, die sich am effizientesten auf der Zielarchitektur umsetzen lässt.

Auch wenn Angaben wie **short** oder **long** auf eine gewisse Größe hindeuten, so legt keiner der C++-Standards die damit verbundenen tatsächlichen Größen fest. Stattdessen gelten nur folgende Regeln:

- Der jeweilige „größere“ Datentyp in der Reihe **char**, **short**, **int**, **long**, **long long** umfasst den Wertebereich der kleineren Datentypen, d.h. **char** ist nicht größer als **short**, **short** nicht größer als **int** usw.
- **wchar_t** basiert auf einem der anderen ganzzahligen Datentypen und übernimmt die entsprechenden Eigenschaften.
- Die korrespondierenden Datentypen mit und ohne Vorzeichen (etwa **signed int** und **unsigned int**) belegen exakt den gleichen Speicherplatz und verwenden die gleiche Zahl von Bits. (Entsprechende Konvertierungen erfolgen entsprechend der Semantik des Zweier-Komplements.)

In C++ werden alle ganzzahligen Datentypen durch Bitfolgen fester Länge repräsentiert: $\{a_i\}_{i=1}^{n}$ mit $a_i \in \{0, 1\}$. Bei ganzzahligen Datentypen ohne Vorzeichen ergibt sich der Wert direkt aus der binären Darstellung:

$$a = \sum_{i=1}^{n} a_i 2^{i-1}$$

Daraus folgt, dass der Wertebereich bei n Bits im Bereich von 0 bis $2^n - 1$ liegt.

Bei ganzzahligen Datentypen mit Vorzeichen übernimmt a_n die Rolle des Vorzeichenbits. Für die Repräsentierung gibt es bei C++ nur drei zugelassene Varianten:

- **Zweier-Komplement**:

$$a = \sum_{i=1}^{n-1} a_i 2^{i-1} - a_n 2^n$$

Wertebereich: $\left[-2^{n-1}, 2^{n-1} - 1\right]$
Diese Darstellung hat sich durchgesetzt und wird von fast allen Prozessor-Architekturen unterstützt.

- **Einer-Komplement**:

$$a = \sum_{i=1}^{n-1} a_i 2^{i-1} - a_n (2^n - 1)$$

Wertebereich: $\left[-2^{n-1}+1, 2^{n-1}-1\right]$
Vorsicht: Es gibt zwei Repräsentierungen für die Null. Es gilt:
$-a == \sim a$
Diese Darstellung gibt es auf einigen historischen Architekturen wie etwa der PDP-1, der UNIVAC 1100/2200 oder der 6502-Architektur.

- **Trennung zwischen Vorzeichen und Betrag**:

$$a = (-1)^{a_n} \sum_{i=1}^{n-1} a_i 2^{i-1}$$

Wertebereich: $\left[-2^{n-1}+1, 2^{n-1}-1\right]$
Vorsicht: Es gibt zwei Repräsentierungen für die Null.
Diese Darstellung wird ebenfalls nur von historischen Architekturen verwendet wie etwa der IBM 7090.

- Leider verzichtet der C++-Standard (anders als bei C) auf Angaben hierzu.
- Die Header-Dateien <*limits*>, <*climits*> und <*cfloat*> liefern die unterstützten Wertebereiche und weitere Eigenschaften der Basistypen der lokalen C++-Implementierung.

links	Postfix-Operatoren: ++, −−, −>, ., etc
rechts	Unäre Operatoren: ++, −−, ∗, &, +, −, !, ∼, **new**, **delete**, **sizeof**, **alignof**, **noexcept**
links	Multiplikative Operatoren: ∗, /, %
links	Additive Operatoren: +, −
links	Schiebe-Operatoren: <<, >>
links	Vergleichs-Operatoren: <, >, <=, >=
links	Gleichheits-Operatoren: ==, !=
links	Bitweises Und: &
links	Bitweises Exklusiv-Oder: ^
links	Bitweises Inklusiv-Oder: |
links	Logisches Und: &&
links	Logisches Oder: ||
rechts	Bedingungs-Operator: ?:
rechts	Zuweisungs-Operatoren: =, ∗=, /=, %=, +=, −=, >>=, <<=, &=, ^=, |=
links	Komma-Operator: ,

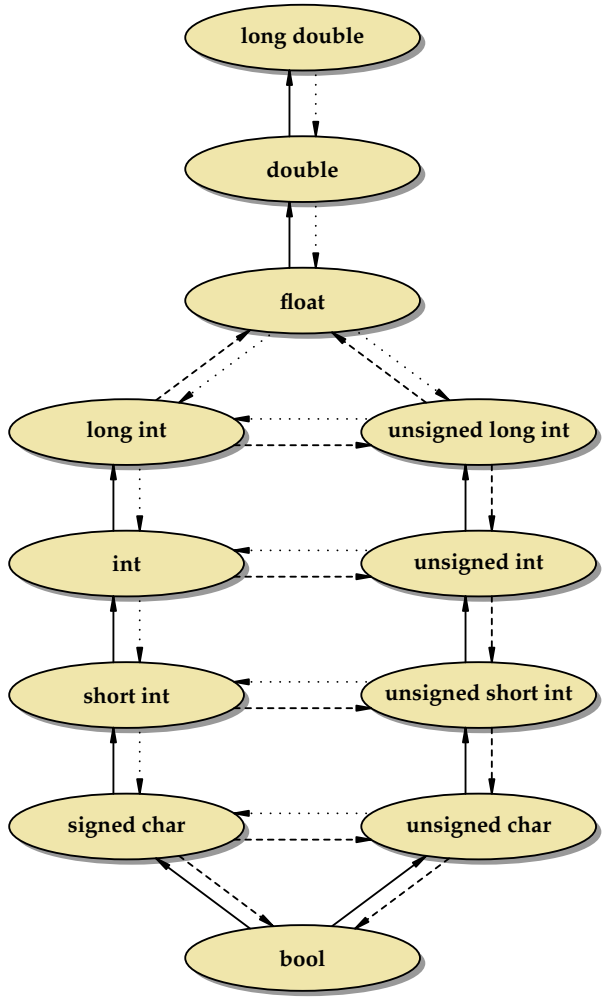
- Die Operatoren sind in der Reihenfolge ihres Vorrangs aufgelistet, beginnend mit den Operatoren höchster Priorität.
- Klammern können verwendet werden, um Operatoren mit Operanden auf andere Weise zu verknüpfen.
- Da nur wenige die gesamte Tabelle auswendig wissen, ist es gelegentlich ratsam, auch dann Klammern aus Gründen der Lesbarkeit einzusetzen, wenn sie nicht strikt notwendig wären.
- Sofern die Operatoren auch von C unterstützt werden, gibt es keine Änderungen der Prioritäten.

- Typ-Konvertierungen können in C sowohl implizit als auch explizit erfolgen.
- Implizite Konvertierungen werden angewendet bei Zuweisungs-Operatoren, Parameterübergaben und Operatoren. Letzteres schließt auch die unären Operatoren mit ein.
- Explizite Konvertierungen erfolgen durch die Cast-Operatoren.

Bei einer Konvertierung zwischen numerischen Typen gilt der Grundsatz, dass – wenn irgendwie möglich – der Wert zu erhalten ist. Falls das jedoch nicht möglich ist, gelten folgende Regeln:

- Bei einer Konvertierung eines vorzeichenbehafteten ganzzahligen Datentyps zum Datentyp ohne Vorzeichen *gleichen Ranges* (also etwa von **int** zu **unsigned int**) wird eine ganze Zahl $a < 0$ zu b konvertiert, wobei gilt, dass $a \bmod 2^n = b \bmod 2^n$ mit n der Anzahl der verwendeten Bits, wobei hier der mod-Operator entsprechend der F-Definition bzw. Euklid gemeint ist. Dies entspricht der Repräsentierung des Zweier-Komplements.
- Der umgekehrte Weg, d.h. vom ganzzahligen Datentyp ohne Vorzeichen zum vorzeichenbehafteten Datentyp gleichen Ranges (also etwa von **unsigned int** zu **int**) hinterlässt ein *undefiniertes* Resultat, falls der Wert nicht darstellbar ist.

- Bei einer Konvertierung von größeren ganzzahligeren Datentypen zu entsprechenden kleineren Datentypen werden die nicht mehr darstellbaren höherwertigen Bits weggeblendet, d.h. es gilt wiederum $a \bmod 2^n = b \bmod 2^n$, wobei n die Anzahl der Bits im kleineren Datentyp ist. (Das Resultat ist aber nur bei ganzzahligen Datentypen ohne Vorzeichen wohldefiniert.)
- Bei Konvertierungen zu *bool* ist das Resultat 0 (**false**), falls der Ausgangswert 0 ist, ansonsten immer 1 (**true**).
- Bei Konvertierungen von Gleitkommazahlen zu ganzzahligen Datentypen wird der ganzzahlige Anteil verwendet. Ist dieser im Zieltyp nicht darstellbar, so ist das Resultat undefiniert.
- Umgekehrt (beispielsweise auf dem Wege von **long int** zu **float**) ist einer der beiden unmittelbar benachbarten darstellbaren Werte zu nehmen, d.h. es gilt entweder $a = b$ oder $a < b \wedge \nexists x : a < x < b$ oder $a > b \wedge \nexists x : a > x > b$ mit x aus der Menge des Zieltyps.


long double
double
float
long int
unsigned long int
int
unsigned int
short int
unsigned short int
signed char
unsigned char
bool

Ausdrücke:	Ein Ausdruck, gefolgt von einem terminierenden Semikolon. Der Ausdruck wird bewertet und das Resultat nicht weiter verwendet. Typische Fälle sind Zuweisungen und Funktions- und Methodenaufrufe.
Blöcke:	Erlaubt die Zufammenfassung mehrerer Anweisungen und eröffnet einen lokalen lexikalisch begrenzten Sichtbereich.
Verzweigungen:	**if** (*condition*) *statement* **if** (*condition*) *statement* **else** *statement* **switch** (*condition*) *statement*

Wiederholungen:	**while** (*condition*) *statement* **do** *statement* **while** (*condition*); **for** (*for-init* ; *condition* ; *expression*) *statement* **for** (*for-range-declaration* : *for-range-initializer*) *statement*
Sprünge:	**return**; **return** *expression* ; **break**;, **continue**;, **goto** *identifier* ;
Ausnahmen:	**throw** *expression* ; **try** *compound-statement handler-seq*

- Deklarationen sind überall zulässig. Die so deklarierten Objekte sind innerhalb des umgebenden lexikalischen Blocks sichtbar, jedoch nicht vor der Deklaration.
- Im Rahmen der (später vorzustellenden) Ausnahmenbehandlungen gibt es **try**-Blöcke.
- Anweisungen können Sprungmarken vorausgehen. Da **goto**-Anweisungen eher vermieden werden, sind Sprungmarken typischerweise nur im Kontext von **switch**-Anweisungen zu sehen unter Verwendung der Schlüsselworte **case** und **default**.

MetaChars.cpp

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

int main() {
   char ch;
   int meta_count(0);
   bool within_range(false);
   bool escape(false);
   while ((ch = cin.get()) != EOF) {
      // counting ...
   }
   if (meta_count == 0) {
      cout << "No";
   } else {
      cout << meta_count;
   }
   cout << " meta characters were found.\n";
} // main()
```

- **int** *meta_count*(0); ist eine Deklaration, die eine Variable namens *meta_count* anlegt, die mit 0 initialisiert wird.
- Alternativ wäre auch **int** *meta_count* = 0 korrekt gewesen. Die Notation mit den Klammern deutet aber die Initialisierung mit einem Konstruktor an.
- Ohne Initialisierungen bleibt der Wert einer lokalen Variable solange undefiniert, bis ihr explizit ein Wert zugewiesen wird.
- Zu beachten ist hier der Unterschied zwischen = (Zuweisung) und == (Vergleich).
- Innerhalb der Bedingung der **while**-Schleife wird von *cin.get*() das nächste Zeichen von der Eingabe geholt, an *ch* zugewiesen und dann mit *EOF* verglichen.

MetaChars.cpp

```
      if (escape) {
         escape = false;
      } else {
         switch (ch) {
         case '*':
         case '?':
         case '\\':
            meta_count += 1; escape = true;
            break;
         case '[':
            meta_count += 1;
            if (!within_range) within_range += 1;
            break;
         case ']':
            if (within_range) meta_count += 1;
            within_range = 0;
            break;
         default:
            if (within_range) meta_count += 1;
            break;
         }
      }
```

MetaChars.cpp

```
switch (ch) {
case '*':
case '?':
case '\\':
   meta_count += 1; escape = true;
   break;
// cases '[' and ']' ...
default:
   if (within_range) {
      meta_count += 1;
   }
   break;
}
```

- Zu Beginn wird der Ausdruck innerhalb der **switch**-Anweisung ausgewertet.
- Dann erfolgt ein Sprung zu der Sprungmarke, die dem berechneten Wert entspricht.
- Falls keine solche Sprungmarke existiert, wird die Sprungmarke **default** ausgewählt, sofern sie existiert.

MetaChars.cpp

```
switch (ch) {
case '*':
case '?':
case '\\':
   meta_count += 1; escape = true;
   break;
// cases '[' and ']' ...
default:
   if (within_range) {
      meta_count += 1;
   }
   break;
}
```

- Beginnend von der ausgesuchten Sprungmarke wird die Ausführung bis zur nächsten *break*-Anweisung fortgesetzt oder eben bis zum Ende der **switch**-Anweisung.
- Zu beachten ist hier, dass *break*-Anweisungen jeweils nur die innerste **switch**-, **for**-, **while**- oder **do**-Anweisung verlassen.

Squares.cpp

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

int main() {
   int limit;
   cout << "Limit: "; cin >> limit;
   for (int n(1); n <= limit; n += 1) {
      cout << setw(4) << n << setw(11) << n*n << endl;
   }
}
```

- **for** (*initialization* ; *condition* ; *expression*) *statement*
  ist nahezu äquivalent zu

```
{
        initialization;
        while (condition){
                statement
                expression;
        }
}
```

- Die Initialisierung, die Bedingung und der Ausdruck dürfen leer sein. Wenn die Bedingung fehlt, wird es zur Endlosschleife.
- Der Sichtbereich von *n* wird lexikalisch auf den Bereich der **for**-Anweisung begrenzt.

Point1.hpp

```
class Point {
   public: // access
      void set_x(float x_coord);
      void set_y(float y_coord);
      float get_x();
      float get_y();

   private: // data
      float x;
      float y;
}; // Point
```

- Datenfelder sollten normalerweise privat gehalten werden, um den direkten Zugriff darauf zu verhindern. Stattdessen ist es üblich, entsprechende Zugriffsmethoden (*accessors*, *mutators*) zu definieren.

Quelle: Die Beispielserie der *Point*-Klassen ist von Jim Heliotis, RIT, geklaut worden.

- Abfragemethoden (*accessors*) wie etwa *get_x* und *get_y* in diesem Beispiel eröffnen der Außenwelt einen Blick auf den Zustand des Objekts. Ein Aufruf eines Akzessors sollte den von außen einsehbaren Objektzustand nicht verändern.
- Änderungsmethoden (wie etwa *set_x* und *set_y*) ermöglichen eine Veränderung des von außen beobachtbaren Objektzustands.
- Der Verzicht auf den direkten Zugang ermöglicht das Durchsetzen semantischer Bedingungen wie etwa der Klasseninvarianten.

Point1.cpp

```
void Point::set_x(float x_coord) {
   x = x_coord;
}

void Point::set_y(float y_coord) {
   y = y_coord;
}

float Point::get_x() {
   return x;
}

float Point::get_y() {
   return y;
}
```

- Im einfachsten Falle können Zugriffsmethoden direkt durch entsprechende **return**-Anweisungen und Zuweisungen implementiert werden.
- Dies mag umständlich erscheinen. Es erlaubt jedoch die einfache Änderung der internen Repräsentierung, ohne dass dabei die Schnittstelle angepasst werden muss.

Point1.cpp

```
#include <iostream>

// class declaration and definition ...

int main() {
   Point p;

   p.set_x(3.0);
   p.set_y(4.0);

   cout << "p=(" << p.get_x() << ','
      << p.get_y() << ')' << endl;

   return 0;
} // main
```

- Zu beachten ist hier, dass p solange undefiniert bleibt, bis beide Änderungsmethoden *set_x* und *set_y* aufgerufen worden sind.

```
dublin$ Point1
p=(3,4)
dublin$
```

Point2.hpp

```
class Point {
   public: // access
      void set_x(float x_coord);
      void set_y(float y_coord);
      float get_x();
      float get_y();

   private: // data
      float radius;
      float angle;
}; // Point
```

- Da die Datenfelder privat sind, ist ein Wechsel von kartesischen zu Polar-Koordinaten möglich. Diese Änderung ist für die Klienten dieser Klasse nicht zu ersehen, da diese weiterhin mit kartesischen Koordinaten arbeiten können.
- Hier ergibt sich ein Unterschied zwischen dem abstrakten Zustand (von den Klienten beobachtbar) und dem internen Zustand (aus der Sicht der Implementierung).

Point2.cpp

```
#include <cmath>
// ...
void Point::set_x(float x_coord) {
   float new_radius(sqrt(x_coord * x_coord +
      get_y() * get_y()));
   angle = atan2(get_y(), x_coord);
   radius = new_radius;
} // set_x

void Point::set_y(float y_coord) {
   float new_radius(sqrt(get_x() * get_x() +
      y_coord * y_coord));
   angle = atan2(y_coord, get_x());
   radius = new_radius;
} // set_y

float Point::get_x() {
   return cos(angle) * radius;
} // get_x

float Point::get_y() {
   return sin(angle) * radius;
} // get_y
```

Point2.cpp

```
#include <iostream>
// ...
int main() {
   Point p;

   p.set_x(3.0);
   p.set_y(4.0);

   cout << "p=(" << p.get_x() << ','
      << p.get_y() << ')' << endl;

   return 0;
}
```

- Leider führt der gleiche benutzende Programmtext zu einem anderen Resultat:

```
dublin$ Point2
p=(-NaN,-NaN)
dublin$
```

- *NaN* steht hier für »not-a-number«, d.h. für eine undefinierte Gleitkommazahl.

Point3.hpp

```
class Point {
   public: // construction
      Point(float x, float y);

   public: // access
      void set_x(float x_coord);
      void set_y(float y_coord);
      float get_x();
      float get_y();

   private: // data
      float radius;
      float angle;
}; // Point
```

- Konstruktoren erlauben es, von Anfang an einen wohldefinierten Zustand zu haben. Der Name einer Konstruktor-Methode ergibt sich immer aus dem Namen der Klasse.
- Wenn mindestens ein Konstruktor in der Klassendeklaration spezifiziert wird, dann ist es nicht mehr möglich, Objekte dieser Klasse zu deklarieren, ohne einen der Konstruktoren zu verwenden.

Point3.cpp

```
Point::Point(float x_coord, float y_coord) {
   radius = sqrt(x_coord * x_coord + y_coord * y_coord);
   angle = atan2(y_coord, x_coord);
} // Point::Point

// ...

int main() {
   Point p(18, -84.2);

   p.set_x(3.0);
   p.set_y(4.0);

   cout << "p=(" << p.get_x() << ',' <<
      p.get_y() << ')' << endl;

   return 0;
} // main
```

- Wenn mit einem wohldefinierten Zustand begonnen wird, dann lassen sich die kartesischen Koordinaten problemlos einzeln ändern.

Point4.hpp

```
class Point {
   public: // creation
      Point(const Point &p); // "const" is TBD
      Point(float x, float y);

   public: // access
      void set_x(float x_coord);
      void set_y(float y_coord);
      float get_x();
      float get_y();

   private: // data
      float radius;
      float angle;
}; // Point
```

- Der gleiche Name darf mehrfach für Methoden der gleichen Klasse vergeben werden, wenn die Signaturen sich bei den Typen der Parameter voneinander unterscheiden.
- Bei einer mehrfachen Verwendung eines Namens wird von einer Überladung gesprochen (*overloading*).

Point4.hpp

```
class Point {
   public: // creation
      Point(const Point &p); // "const" is TBD
      Point(float x, float y);

   public: // access
      void set_x(float x_coord);
      void set_y(float y_coord);
      float get_x();
      float get_y();

   private: // data
      float radius;
      float angle;
}; // Point
```

- Entsprechend kann ein zweiter Konstruktor definiert werden, der in diesem Beispiel die Koordinaten von einem existierenden Punkt-Objekt bezieht.
- **const** *Point*& *p* vermeidet im Vergleich zu *Point p* das Kopieren des Parameters und lässt (dank dem **const**) Änderungen nicht zu.

```
int i(0);
int& j(i); // j ist eine Referenz auf i
j = 3; // i hat jetzt den Wert 3
```

- Referenzen sind konstante Zeiger, die von Anfang an mit einem anderen Objekt fest verknüpft sind.
- Die Verknüpfung ergibt sich entweder
  - aus der Parameterübergabe,
  - aus der Rückgabe einer Funktion oder Methode oder
  - aus der Initialisierung
- Im Unterschied zu Zeigern entfällt die explizite Dereferenzierung.
- Referenzen werden durch die Verwendung eines & deklariert.

Überladen von Methoden

Point4.cpp

```
Point::Point(const Point &p) {
   radius = p.radius;
   angle = p.angle;
} // Point::Point
// ...
int main() {
   Point p(18, -84.2);
   Point q(p);

   p.set_x(3.0);
   p.set_y(4.0);

   cout << "p=(" << p.get_x() << ',' <<
      p.get_y() << ')' << endl;
   cout << "q=(" << q.get_x() << ',' <<
      q.get_y() << ')' << endl;

   return 0;
} // main
```

```
dublin$ Point4
p=(3,4)
q=(18,-84.2)
dublin$
```

Point5.cpp

```
Point::Point(float x_coord, float y_coord):
   radius(sqrt(x_coord * x_coord + y_coord * y_coord)),
   angle(atan2(y_coord, x_coord)) {
}

Point::Point(const Point &p):
   radius(p.radius), angle(p.angle) {
}
```

- Vor dem eigentlichen Block können bei Konstruktoren hinter dem Doppelpunkt Initialisierungssequenzen spezifiziert werden, die abgearbeitet werden, bevor die eigentliche Methode des Konstruktors aufgerufen wird.
- Dabei ist zu beachten, dass die Initialisierungsreihenfolge abgeleitet wird von der Reihenfolge in der Klassendeklaration und nicht der Reihenfolge der Initialisierungen.

Point5.cpp

```
Point::Point(float x_coord, float y_coord):
   radius(sqrt(x_coord * x_coord + y_coord * y_coord)),
   angle(atan2(y_coord, x_coord)) {
}

Point::Point(const Point &p):
   radius(p.radius), angle(p.angle) {
}
```

- Die Verwendung von Konstruktoren der Basisklasse ist dabei zulässig.
- Da in C++ grundsätzlich keine voreingestellten Initialisierungen stattfinden (anders als in Java oder vielen anderen OO-Sprachen), empfiehlt es sich, diese Initialisierungsmöglichkeit konsequent zur Vermeidung von Überraschungen einzusetzen.

- Die Lebenszeit eines Objekts beginnt, wenn
  - Speicher für das Objekt zur Verfügung steht (kann global, auf dem Heap oder lokal sein) und
  - bei einer nicht-trivialen Initialisierung (durch Konstruktoren) diese abgeschlossen ist.
- Das bedeutet, dass im Normalfall ein Objekt erst verwendet werden darf, sobald der Konstruktor abgeschlossen ist.
- Die Lebenszeit eines Objekts endet, sobald
  - der Aufruf des Destruktors beginnt oder spätestens wenn
  - der Speicher für das Objekt freigegeben oder andersweitig genutzt wird.

```
const double PI = 3.141592653589793238 46;
```

- Hier wird *PI* als Konstante deklariert, d.h. *PI* sollte nachträglich verändert werden. Dies wird jedoch nicht immer hart durchgesetzt.

```
Point(const Point& p);
```

- Hier wird *p* über eine Referenz (d.h. einem nicht veränderbaren Zeiger) übergeben und gleichzeitig sichergestellt, dass *p* nicht vom Aufrufer verändert wird. Dies verbessert die Effizienz (da das Kopieren vermieden wird) ohne dies zu Lasten der Sicherheit zu tun.

```
float get_x() const;
```

- Durch das Schlüsselwort **const** am Ende der Signatur einer Methode wird diese zu einer reinen auslesenden Methode, d.h. sie darf den Zustand des Objekts nicht verändern.

```
const Key& get_key();
```

- Hier wird der Aufrufer dieser Funktion daran gehindert, den Wert hinter der zurückgelieferten Referenz zu verändern.

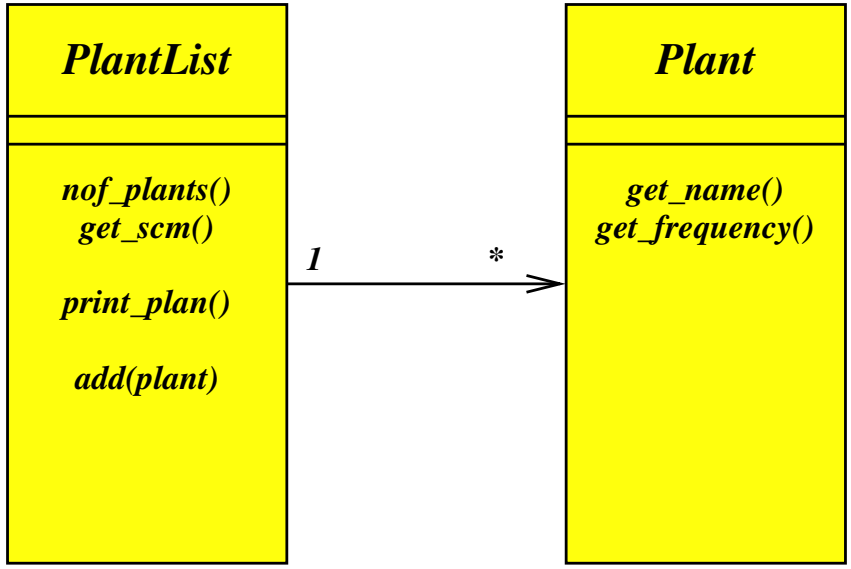
```
constexpr double PI = 3.14159265358979323846;
```

- Im Unterschied zu **const** beinhaltet **constexpr** die Zusicherung, dass der Wert sich nicht nur zur Laufzeit nicht ändert, sondern auch bereits zur Übersetzzeit bestimmbar ist, wenn alle Parameter ebenfalls konstant sind.

```
constexpr unsigned int factorial(unsigned int n) {
   return n > 0? n * factorial(n - 1): 1;
}

char a[factorial(5)];
```

- Entsprechend können **constexpr**-Resultate auch dort verwendet werden, wo der Übersetzer Konstanten verlangt (wie bei der Dimensionierung eines Arrays). Entsprechende Funktionen dürfen aber nur aus einer **return**-Anweisung bestehen.



- Die Aufgabenstellung ist die Generierung eines tageweisen Bewässerungsplans für eine Menge von Pflanzen, von denen jede unterschiedliche Bewässerungsfrequenzen bevorzugt.
- *scm* steht für das *kleinste gemeinsame Vielfache* (engl. *smallest common multiple*; kurz: kgV) und *get_scm* liefert das kgV aller erfasster Bewässerungsfrequenzen zurück. Nach dieser Zahl von Tagen wiederholt sich der Bewässerungsplan.

Plant.hpp

```
#ifndef PLANT_H
#define PLANT_H

#include <string>

class Plant {
   public:
      // constructors
      Plant(std::string plantName, int wateringFrequency);
         // PRE: wateringFrequency >= 1
      Plant(const Plant &plant);

      // accessors
      std::string get_name() const;
      int get_frequency() const;

   private:
      std::string name;
      int frequency;
};

#endif
```

Plant.cpp

```
#include <cassert>
#include "Plant.hpp"

Plant::Plant(std::string plantName, int wateringFrequency) :
   name(plantName),
   frequency(wateringFrequency) {
   assert(wateringFrequency >= 1);
} // Plant::Plant

Plant::Plant(const Plant &plant) :
   name(plant.name),
   frequency(plant.frequency) {
} // Plant::Plant

std::string Plant::get_name() const {
   return name;
} // Plant::get_name

int Plant::get_frequency() const {
   return frequency;
} // Plant::get_frequency
```

Plant.cpp

```
Plant::Plant(const Plant& plant) :
   name(plant.name),
   frequency(plant.frequency) {
} // Plant::Plant
```

- Dies ist ein kopierender Konstruktor (*copy constructor*), der eine Initialisierung mit Hilfe eines bereits existierenden Objekts durchführt.
- Dieser Konstruktor wird in vielen Fällen implizit aufgerufen. Dazu ist beispielsweise der Fall, wenn dieser Datentyp im Rahmen der (noch vorzustellenden) Template-Klasse für Listen verwendet wird, da diese die Werte grundsätzlich kopiert.

Plant.cpp

```
Plant::Plant(const Plant& plant) :
   name(plant.name),
   frequency(plant.frequency) {
} // Plant::Plant
```

- Deswegen ist dieser Konstruktor nicht nur eine Bereicherung der *Plant*-Klasse, sondern auch generell eine Voraussetzung, um Objekte dieser Klasse in Listen aufnehmen zu können.
- Wenn kein kopierender Konstruktor zur Verfügung gestellt wird (und auch sonst keine anderen Konstruktoren explizit deklariert werden) gibt es eine Voreinstellung für den kopierenden Konstruktor, der alle Felder einzeln kopiert. Dies wäre hier kein Problem, aber das kann zur bösen Falle werden, wenn Zeiger auf diese Weise vervielfältigt und dann später möglicherweise mehrfach freigegeben werden.

Plant.cpp

```
std::string Plant::get_name() const {
using namespace std;
   return name;
} // Plant::get_name

int Plant::get_frequency() const {
   return frequency;
} // Plant::get_frequency
```

- Zu beachten ist hier das Schlüsselwort **const** am Ende der Signatur. Dies legt fest, dass diese Methode den (abstrakten!) Status des Objekts nicht verändert.

PlantList.hpp

```
#ifndef PLANTLIST_H
#define PLANTLIST_H

#include <list>
#include "Plant.hpp"

class PlantList {
   public:
      // constructors
      PlantList();
      // accessors
      int nof_plants() const;
      int get_scm() const; // PRE: nof_plants() > 0
      // printing
      void print_plan(int day); // PRE: day >= 0
      void print_plan();
      // mutators
      void add(Plant plant);
   private:
      std::list< Plant > plants;
      int scm; // of watering frequencies
};
#endif
```

PlantList.h

```
#include <list>
#include "Plant.hpp"

// ...

std::list< Plant > plants;
```

- Dies deklariert *plants* als eine Liste von Elementen des Typs *Plant*.
- *list* ist ein Template, das einen Typparameter als Elementtyp erwartet.
- Entsprechend wird hier nicht nur *plants* deklariert, sondern auch implizit ausgehend von dem Template *list* eine neue Klasse erzeugt. Dies wird auch als Instantiierung eines Templates bezeichnet (*template instantiation*).
- Das Listen-Template gehört zur STL (*standard template library*), die Bestandteil von ISO C++ ist.

PlantList.C

```
#include <cassert>
#include "PlantList.hpp"

PlantList::PlantList() :
   scm(0) {
} // PlantList::PlantList

int PlantList::nof_plants() const {
   return plants.size();
} // PlantList::nof_plants

int PlantList::get_scm() const {
   assert(scm > 0);
   return scm;
} // PlantList::get_scm
```

- *scm* ist dank des Konstruktors immer wohldefiniert.
- Später bestehen wir (entsprechend der Vorbedingung) darauf, dass mindestens eine Pflanze eingetragen ist, bevor ein Aufruf von *get_scm* zulässig ist.

PlantList.cpp

```
void PlantList::print_plan(int day) {
   assert(day >= 0);
   for (Plant& plant: plants) {
      if (day % plant.get_frequency() == 0) {
         std::cout << plant.get_name() << std::endl;
      }
   }
}
```

- Seit C++11 gibt es eine spezielle Form der **for**-Schleife, die die Iteration einer Datenstruktur auf elegantem Wege erlaubt.
- In jedem Schleifendurchlauf ist *plant* eine Referenz auf das aktuelle Element aus der Liste *plants*.

PlantList.cpp

```
void PlantList::add(Plant plant) {
   int frequency( plant.get_frequency() );
   if (scm == 0) {
      scm = frequency;
   } else if (scm % frequency != 0) {
      // computing smallest common multiple using Euclid
      int x0(scm), x(scm), y0(frequency), y(frequency);
      while (x != y) {
         if (x > y) {
            y += y0;
         } else {
            x += x0;
         }
      }
      scm = x;
   }
   plants.push_back(plant);
} // PlantList::add
```

- *plants.push_back*(*plant*) belegt Speicher für eine Kopie von *plant* und hängt diese Kopie an das Ende der Liste ein.

WateringPlan.cpp

```
#include <iostream>
#include <string>
#include "Plant.hpp"
#include "PlantList.hpp"

using namespace std;

int main() {
   PlantList plants;
   std::string name; int frequency;

   while (std::cin >> name && std::cin >> frequency) {
      plants.add(Plant(name, frequency));
   }
   plants.print_plan();
}
```

WateringPlan.cpp

```
while (std::cin >> name && std::cin >> frequency) {
   plants.add(Plant(name, frequency));
}
```

- Normalerweise liefert *cin* >> *name* den Wert von *cin* zurück, um eine Verkettung von Eingabe-Operationen für den gleichen Eingabestrom zu ermöglichen.
- Hier jedoch findet implizit eine Konvertierung statt, da ein **bool**-Wert benötigt wird. Dies gelingt u.a. mit Hilfe eines sogenannten Konvertierungs-Operators der entsprechenden Klasse.
- Entsprechend ist die gesamte Bedingung genau dann wahr, falls beide Lese-Operationen erfolgreich sind.
- *Plant*(*name*, *frequency*) erzeugt ein sogenanntes temporäres Objekt des Typs *Plant*, das vollautomatisch wieder aufgeräumt wird, sobald die Ausführung der zugehörigen Anweisung beendet ist.

- In allen bisherigen Beispielen belegten die Objekte entweder statischen Speicherplatz oder sie lebten auf dem Stack.
- Dies vermied bislang völlig den Aufwand einer dynamischen Speicherverwaltung. Es gehört zu den Vorteilen von C++ (und einigen anderen hybriden OO-Sprachen), dass nicht für alle Objekte der Speicherplatz dynamisch belegt werden muss.

- Auf der anderen Seite ist die Beachtung einiger Richtlinien unerlässlich, wenn Klassen Zeiger auf dynamische Datenstrukturen verwenden, da
  - wegen der fehlenden automatischen Speicherbereinigung (*garbage collection*) es in der Verantwortung der Klassenimplementierung liegt, referenzierte Datenstrukturen wieder freizugeben und da
  - Konstruktoren und Zuweisungs-Operatoren per Voreinstellung nur die Datenfelder kopieren (*shallow copy*) und somit Zeigerwerte implizit vervielfältigt werden können.

Integer.hpp

```
#ifndef INTEGER_H
#define INTEGER_H

class Integer {
   public:
      // constructor
      Integer(int initval);
      // destructor
      ~Integer();
      // accessor
      int get_value() const;
      void set_value(int newval);
   private:
      int value;
}; // class Integer

#endif
```

- Die Signatur eines Destruktors besteht aus einer Tilde „~“, dem Namen der Klasse (analog zu den Konstruktoren) und einer leeren Parameterliste.

Integer.cpp

```
#include <iostream>
#include "Integer.hpp"

using namespace std;

Integer::Integer(int intval) :
   value(intval) {
   cout << "Integer constructor: value = " <<
      value << endl;
} // Integer::Integer

Integer::~Integer() {
   cout << "Integer destructor: value = " <<
      value << endl;
} // Integer::~Integer

int Integer::get_value() const {
   return value;
} // Integer::get_value

void Integer::set_value(int newval) {
   value = newval;
} // Integer::set_value
```

- Prinzipiell können (wie in diesem Beispiel) beliebige Anweisungen wie auch Ausgaben in Destruktoren aufgenommen werden. Das kann aber zusätzliche Komplikationen mit sich bringen, wenn etwa eine Ausnahmenbehandlung in Gang gesetzt werden sollte.
- Wenn kein Destruktor angegeben wird, dann kommt eine Voreinstellung zum Zuge, die die Destruktoren für alle einzelnen Datenfelder aufruft.
- Aber auch wenn ein Destruktor spezifiziert wird, dann bleibt immer noch der automatische Aufruf der Destruktoren aller Datenfelder.
- Bei elementaren Datentypen (einschließlich den Zeigern) passiert hier jedoch nichts.

TestInteger1.cpp

```
#include <iostream>
#include "Integer.hpp"

using namespace std;

int main() {
   cout << "main starts" << endl;
   {
      Integer i(1);
      cout << "working on i = " << i.get_value() << endl;
   }
   cout << "main ends" << endl;
} // main
```

- Durch eine Deklaration wie hier mit *Integer i*(1) wird der passende Konstruktor aufgerufen.

```
dublin$ TestInteger1
main starts
Integer constructor: value = 1
working on i = 1
Integer destructor: value = 1
main ends
dublin$
```

- Der Sichtbereich von *i* ist statisch begrenzt auf den umgebenden Block. Die Lebenszeit beginnt und endet mit der Laufzeit des umgebenden Blocks.
- Der Destruktor wird implizit beim Verlassen des umgebenden Blocks aufgerufen.

TestInteger2.cpp

```
#include <iostream>
#include "Integer.hpp"
using namespace std;

int main() {
   cout << "main starts" << endl;

   Integer* ip (new Integer(1));
   cout << "working on ip = " << ip->get_value() << endl;
   delete ip;

   cout << "main ends" << endl;
} // main
```

- Mit *Integer∗ ip* wird *ip* als Zeiger auf *Integer* deklariert.
- Der Ausdruck **new** *Integer*(1) veranlasst das dynamische Belegen von Speicher für ein Objekt des Typs *Integer* und ruft den passenden Konstruktor auf.

```
dublin$ TestInteger2
main starts
Integer constructor: value = 1
working on ip = 1
Integer destructor: value = 1
main ends
dublin$
```

- Die Verantwortung für die Speicherfreigabe verbleibt beim Aufrufer des **new**-Operators.
- Mit **delete** *ip* wird der Destruktor von *ip* aufgerufen und danach der belegte Speicherplatz freigegeben.

TestInteger3.cpp

```
#include <iostream>
#include "Integer.hpp"

using namespace std;

int main() {
   cout << "main starts" << endl;

   Integer* ip (new Integer(1));
   Integer& ir (*ip);
   cout << "working on ip = " << ip->get_value() << endl;
   ir.set_value(2);
   cout << "working on ip = " << ip->get_value() << endl;
   delete ip;

   cout << "main ends" << endl;
} // main
```

- Mit *Integer& ir* (∗*ip*) wird *ir* als Referenz für ein *Integer*-Objekt deklariert.

- Im Vergleich zu Zeigern gibt es bei Referenzen einige Unterschiede:
  - Im Rahmen ihrer Deklaration müssen sie mit einem Objekt des referenzierten Typs verbunden werden.
  - Referenzen bleiben konstant, d.h. sie können während ihrer Lebenszeit nicht ein anderes Objekt referenzieren.
  - Referenzen werden syntaktisch wie das Objekt, das sie referenzieren, behandelt. Entsprechend wird etwa ».« an Stelle von »−>« verwendet.

ListOfFriends.hpp

```
class ListOfFriends {
   public:
      // constructor
      ListOfFriends();
      ListOfFriends(const ListOfFriends& list);
      ~ListOfFriends();

      // overloaded operators
      ListOfFriends& operator=(const ListOfFriends& list);

      // printing
      void print();

      // mutator
      void add(const Friend& f);

   private:
      struct Node* root;
      void addto(Node*& p, Node* newNode);
      void visit(const Node* const p);
}; // class ListOfFriends
```

- Ein Objekt der Klasse *ListOfFriends* verwaltet eine Liste von Freunden und ermöglicht die sortierte Ausgabe (alphabetisch nach dem Namen).
- Die Implementierung beruht auf einem sortierten binären Baum. Der Datentyp **struct** *Node* repräsentiert einen Knoten dieses Baums.
- Zu beachten ist hier, dass eine Deklaration eines Objekts des Typs **struct** *Node*$*$ auch dann zulässig ist, wenn **struct** *Node* noch nicht bekannt ist, da der benötigte Speicherplatz bei Zeigern unabhängig vom referenzierten Datentyp ist.

ListOfFriends.cpp

```
struct Node {
   struct Node* left;
   struct Node* right;
   Friend f;
   Node(const Friend& newFriend);
   Node(const Node* const& node);
   ~Node();
}; // struct Node

Node::Node(const Friend& newFriend) :
   left(0), right(0), f(newFriend) {
} // Node::Node
```

- Im Vergleich zu **class** sind bei **struct** alle Komponenten implizit **public**. Da hier die Datenstruktur nur innerhalb der Implementierung deklariert wird, stört dies nicht, da sie von außen nicht einsehbar ist.
- Der hier gezeigte Konstruktor legt ein Blatt an.

ListOfFriends.cpp

```
Node::Node(const Node* const& node) :
   left(0), right(0), f(node->f) {
   if (node->left) {
      left = new Node(node->left);
   }
   if (node->right) {
      right = new Node(node->right);
   }
} // Node::Node
```

- Der zweite Konstruktor für **struct** *Node* akzeptiert einen Zeiger auf *Node* als Parameter. Die beiden **const** in der Signatur stellen sicher, dass nicht nur der (als Referenz übergebene) Zeiger nicht verändert werden darf, sondern auch nicht der Knoten, auf den dieser verweist.
- Hier ist es sinnvoll, einen Zeiger als Parameter zu übergeben, da in diesem Beispiel Knoten ausschließlich über Zeiger referenziert werden.

- Hier werden die Felder *left* und *right* zunächst in der Initialisierungssequenz auf 0 initialisiert und nachher bei Bedarf auf neu angelegte Knoten umgebogen. So ist garantiert, dass die Zeiger immer wohldefiniert sind.
- Tests wie **if** (*node*−>*left*) überprüfen, ob ein Zeiger ungleich 0 ist.
- Zu beachten ist hier, dass der Konstruktor sich selbst rekursiv für die Unterbäume *left* und *right* von *node* aufruft, sofern diese nicht 0 sind.
- Auf diese Weise erhalten wir hier eine tiefe Kopie (*deep copy*), die den gesamten Baum beginnend bei *node* dupliziert.

ListOfFriends.cpp

```
Node::~Node() {
   if (left) {
      delete left;
   }
   if (right) {
      delete right;
   }
} // Node::~Node
```

- Wie beim Konstruieren muss hier die Destruktion bei *Node* rekursiv arbeiten.
- Diese Lösung geht davon aus, dass ein Unterbaum niemals mehrfach referenziert wird.
- Nur durch die Einschränkung der Sichtbarkeit kann dies auch garantiert werden.

ListOfFriends.cpp

```
ListOfFriends::ListOfFriends() :
   root(0) {
} // ListOfFriends::ListOfFriends

ListOfFriends::ListOfFriends(const ListOfFriends& list) :
   root(0) {
   Node* r(list.root);
   if (r) {
      root = new Node (r);
   }
} // ListOfFriends::ListOfFriends
```

- Der Konstruktor ohne Parameter (*default constructor*) ist trivial: Wir setzen nur *root* auf 0.
- Der kopierende Konstruktor ist ebenso hier recht einfach, da die entscheidende Arbeit an den rekursiven Konstruktor für *Node* delegiert wird.
- Es ist hier nur darauf zu achten, dass der Konstruktor für *Node* nicht in dem Falle aufgerufen wird, wenn *list.root* gleich 0 ist.

ListOfFriends.cpp

```
ListOfFriends::~ListOfFriends() {
   if (root) {
      delete root;
   }
} // ListOfFriends::~ListOfFriends
```

- Analog delegiert der Destruktor für *ListOfFriends* die Arbeit an den Destruktor für *Node*.
- Es ist nicht schlimm, wenn der **delete**-Operator für 0-Zeiger aufgerufen wird. Das wird vom ISO-Standard für C++ ausdrücklich erlaubt. Die **if**-Anweisung spart aber Ausführungszeit.

ListOfFriends.cpp

```
ListOfFriends& ListOfFriends::operator=
      (const ListOfFriends& list) {
   if (this != &list) { // protect against self-assignment
      if (root) {
         delete root;
      }
      if (list.root) {
         root = new Node (list.root);
      } else {
         root = 0;
      }
   }
   return *this;
} // ListOfFriends::operator=
```

- Ein rekursiv arbeitender kopierender Konstruktur und zugehöriger Destruktor genügen alleine nicht, da der voreingestellte Zuweisungs-Operator nur den Wurzelzeiger kopieren würde (*shallow copy*) und eine rekursive Kopie (*deep copy*) unterbleiben würde.

- Dies würde die wichtige Annahme (des Destruktors) verletzen, dass der selbe Baum nicht von mehreren Objekten des Typs *ListOfFriends* referenziert werden darf.
- Entsprechend ist die Implementierung unvollständig, solange eine simple Zuweisung von *ListOfFriends*-Objekten diese wichtige Annahme verletzen kann.
- Bei der Implementierung des Zuweisungs-Operators ist darauf zu achten, dass Objekte an sich selbst zugewiesen werden können. **this** repräsentiert einen Zeiger auf das Objekt, auf der die aufgerufene Methode arbeitet. &*list* ermittelt die Adresse von *list* und erlaubt somit einen Vergleich von Zeigerwerten.

ListOfFriends.cpp

```
void ListOfFriends::addto(Node*& p, Node* newNode) {
   if (p) {
      if (newNode->f.get_name() < p->f.get_name()) {
         addto(p->left, newNode);
      } else {
         addto(p->right, newNode);
      }
   } else {
      p = newNode;
   }
} // ListOfFriends::addto

void ListOfFriends::add(const Friend& f) {
   Node* node( new Node(f) );
   addto(root, node);
} // ListOfFriends::add
```

- Wenn ein neuer Freund in die Liste aufgenommen wird, ist ein neues Blatt anzulegen, das auf rekursive Weise in den Baum mit Hilfe der privaten Methode *addto* eingefügt wird.

ListOfFriends.cpp

```
void ListOfFriends::visit(const Node* const p) {
   if (p) {
      visit(p->left);
      cout << p->f.get_name() << ": " <<
         p->f.get_info() << endl;
      visit(p->right);
   }
} // ListOfFriends::visit

void ListOfFriends::print() {
   visit(root);
} // ListOfFriends::print
```

- Analog erfolgt die Ausgabe rekursiv mit Hilfe der privaten Methode *visit*.

TestFriends.cpp

```
ListOfFriends list1;
```

- Diese Deklaration ruft implizit den Konstruktor von *ListOfFriends* auf, der keine Parameter verlangt (*default constructor*). In diesem Falle wird *root* einfach auf 0 gesetzt werden.

TestFriends.cpp

```
ListOfFriends list2(list1);
```

- Diese Deklaration führt zum Aufruf des kopierenden Konstruktors, der den vollständigen Baum von *list1* für *list2* dupliziert.

TestFriends.cpp

```
ListOfFriends list3;
list3 = list1;
```

- Hier wird zunächst der Konstruktor von *ListOfFriends* ohne Parameter aufgerufen (*default constructor*).
- Danach kommt es zur Ausführung des Zuweisungs-Operators, der den Baum von *list1* dupliziert und bei *list3* einhängt.

Function.hpp

```
#include <string>

class Function {
   public:
      virtual ~Function() {};
      virtual std::string get_name() const = 0;
      virtual double execute(double x) const = 0;
}; // class Function
```

- Polymorphe Methoden einer Basis-Klasse können in einer abgeleiteten Klasse überdefiniert werden.
- Eine Methode wird durch das Schlüsselwort **virtual** als polymorph gekennzeichnet.
- Dies wird auch als *dynamischer Polymorphismus* bezeichnet, da die auszuführende Methode zur Laufzeit bestimmt wird,

Function.hpp

```
    virtual std::string get_name() const = 0;
```

- Die Angabe von $= 0$ am Ende einer Signatur einer polymorphen Methode ermöglicht den Verzicht auf eine zugehörige Implementierung.
- In diesem Falle gibt es nur Implementierungen in abgeleiteten Klassen und nicht in der Basis-Klasse.
- So gekennzeichnete Methoden werden *abstrakt* genannt.
- Klassen mit mindestens einer solchen Methode werden *abstrakte Klassen* genannt.
- Abstrakte Klassen können nicht instantiiert werden.

Function.hpp

```
#include <string>

class Function {
   public:
      virtual ~Function() {};
      virtual std::string get_name() const = 0;
      virtual double execute(double x) const = 0;
}; // class Function
```

- Wenn wie in diesem Beispiel alle Methoden abstrakt sind (oder wie beim Dekonstruktor innerhalb der Klassendeklaration implementiert werden), kann die zugehörige Implementierung vollständig entfallen. Entsprechend gibt es keine zugehörige Datei namens *Function.cpp*.
- Implizit definierte Destruktoren und Operatoren müssen explizit als abstrakte Methoden deklariert werden, wenn die Möglichkeit erhalten bleiben soll, sie in abgeleiteten Klassen überzudefinieren.

Sinus.hpp

```
#include <string>
#include "Function.hpp"

class Sinus: public Function {
   public:
      virtual std::string get_name() const;
      virtual double execute(double x) const;
}; // class Sinus
```

- *Sinus* ist eine von *Function* abgeleitete Klasse.
- Das Schlüsselwort **public** bei der Ableitung macht diese Beziehung öffentlich. Alternativ wäre auch **private** zulässig. Dies ist aber nur in seltenen Fällen sinnvoll.
- Die Wiederholung des Schlüsselworts **virtual** bei den Methoden ist nicht zwingend notwendig, erhöht aber die Lesbarkeit.
- Da $= 0$ nirgends mehr innerhalb der Klasse *Sinus* verwendet wird, ist die Klasse nicht abstrakt und somit ist eine Instantiierung zulässig.

Sinus.cpp

```
#include <cmath>
#include "Sinus.hpp"

std::string Sinus::get_name() const {
   return "sin";
} // Sinus::get_name

double Sinus::execute(double x) const {
   return std::sin(x);
} // Sinus::execute
```

- Alle Methoden, die nicht abstrakt sind und nicht in einer der Basisklassen definiert worden sind, müssen implementiert werden.
- Hier wird auf die Definition eines Dekonstruktors verzichtet. Stattdessen kommt der leere Dekonstruktor der Basisklasse zum Zuge.

TestSinus.cpp

```
#include <iostream>
#include "Sinus.hpp"

using namespace std;

int main() {
   Function* f(new Sinus());
   double x;

   while (cout << f->get_name() << ": " &&
          cin >> x) {
      cout << f->execute(x) << endl;
   }
   return 0;
} // main
```

- Variablen des Typs *Function* können nicht deklariert werden, weil *Function* eine abstrakte Klasse ist.
- Stattdessen ist es aber zulässig, Zeiger oder Referenzen auf *Function* zu deklarieren, also *Function*∗ oder *Function*&.

TestSinus.cpp

```
#include <iostream>
#include "Sinus.hpp"

using namespace std;

int main() {
   Function* f(new Sinus());
   double x;

   while (cout << f->get_name() << ": " &&
          cin >> x) {
      cout << f->execute(x) << endl;
   }
   return 0;
} // main
```

- Zeiger auf Instantiierungen abgeleiteter Klassen (wie etwa hier das Resultat von **new** *Sinus*()) können an Zeiger der Basisklasse (hier: *Function∗ f*) zugewiesen werden.
- Umgekehrt gilt dies jedoch nicht!

TestSinus.cpp

```
#include <iostream>
#include "Sinus.hpp"

using namespace std;

int main() {
   Function* f(new Sinus());
   double x;

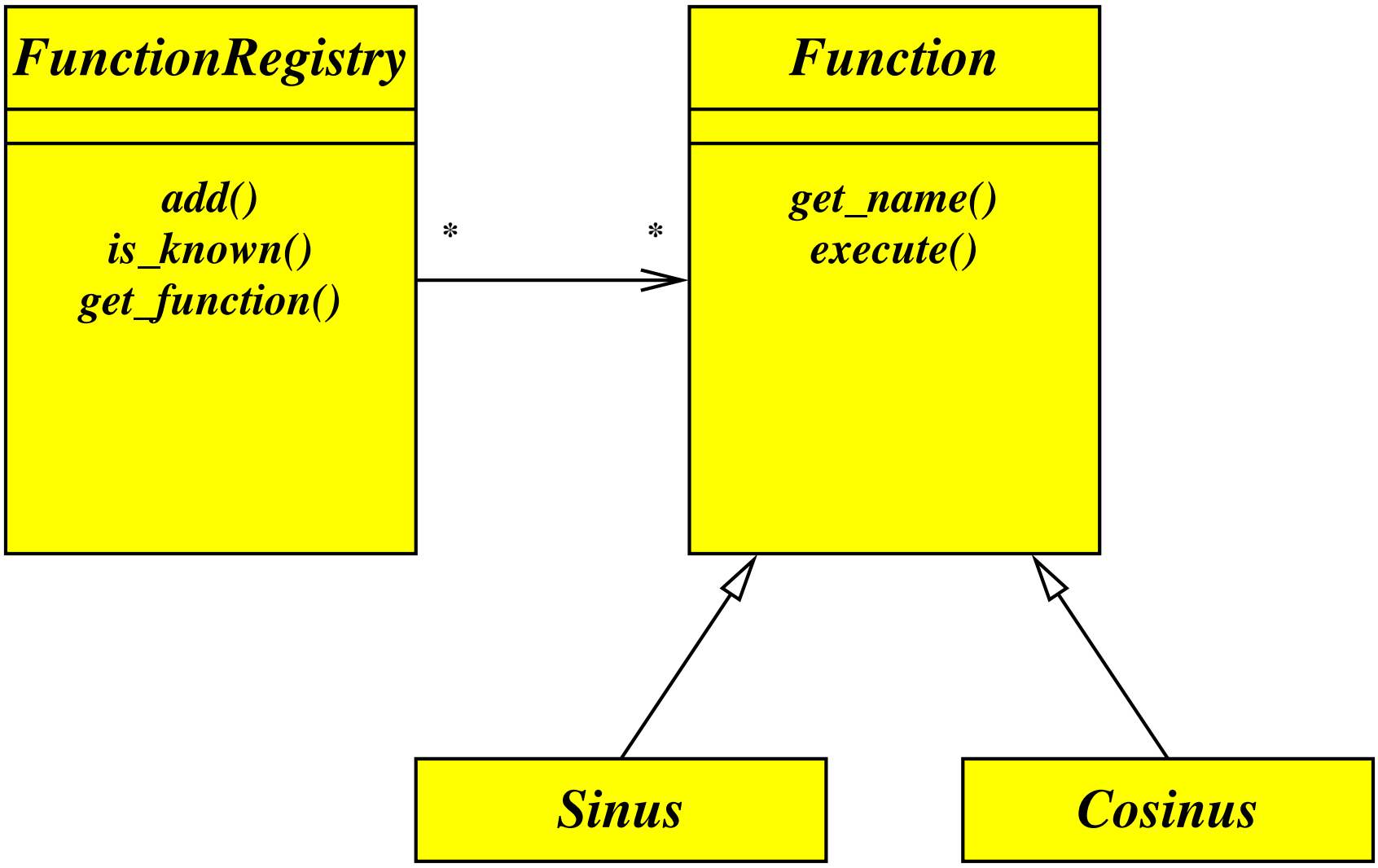
   while (cout << f->get_name() << ": " &&
          cin >> x) {
      cout << f->execute(x) << endl;
   }
   return 0;
} // main
```

- Wenn eine Methode mit dem Schlüsselwort **virtual** versehen ist, dann erfolgt die Bestimmung der zugeordneten Methodenimplementierung *erst zur Laufzeit* in Abhängigkeit vom dynamischen Typ, der bei Zeigern und Referenzen eine beliebige Erweiterung des deklarierten Typs sein kann.

TestSinus.cpp

```
Function* f(new Sinus());
```

- Fehlt das Schlüsselwort **virtual**, so steht bereits zur Übersetzzeit fest, welche Implementierung aufzurufen ist.
- In diesem Beispiel hat die Variable f den statischen Typ *Function*$*$, während zur Laufzeit der dynamische Typ hier *Sinus*$*$ ist.


FunctionRegistry
add()
is_known()
get_function()
*
*
Function
get_name()
execute()
Sinus
Cosinus

- Die Einführung einer Klasse *FunctionRegistry* erlaubt es, Funktionen über ihren Namen auszuwählen.
- Hiermit ist es beispielsweise möglich, den Namen einer Funktion einzulesen und dann mit dem gegebenen Namen ein zugehöriges Funktionsobjekt zu erhalten.
- Dank der Kompatibilität einer abgeleiteten Klasse zu den Basisklassen ist es möglich, heterogene Listen (d.h. Listen mit Objekten unterschiedlicher Typen) zu verwalten, sofern eine gemeinsame Basisklasse zur Verfügung steht. In diesem Beispiel ist das *Function*.

FunctionRegistry.hpp

```
#include <map>
#include <string>
#include "Function.hpp"

class FunctionRegistry {
   public:
      void add(Function* f);
      bool is_known(std::string fname) const;
      Function* get_function(std::string fname);
   private:
      std::map< std::string, Function* > registry;
}; // class FunctionRegistry
```

- *map* ist eine Implementierung für assoziative Arrays und gehört zu den generischen Klassen der Standard-Template-Library (STL)
- *map* erwartet zwei Typen als Parameter: den Index- und den Element-Typ.
- Hier werden Zeichenketten als Indizes verwendet (Datentyp *string*) und die Elemente sind Zeiger auf Funktionen (Datentyp *Function*$*$).

- Generell können heterogene Datenstrukturen nur Zeiger oder Referenzen auf den polymorphen Basistyp aufnehmen, da
  - abstrakte Klassen nicht instantiiert werden können und
  - das Kopieren eines Objekts einer erweiterten Klasse zu einem Objekt der Basisklasse (falls überhaupt zulässig) die Erweiterungen ignorieren würde. Dies wird im Englischen *slicing* genannt. (In Oberon nannte dies Wirth eine Projektion.)

FunctionRegistry.cpp

```
#include <string>
#include "FunctionRegistry.hpp"

void FunctionRegistry::add(Function* f) {
   registry[f->get_name()] = f;
} // FunctionRegistry::add

bool FunctionRegistry::is_known(std::string fname) const {
   return registry.find(fname) != registry.end();
} // FunctionRegistry::is_known

Function* FunctionRegistry::get_function(std::string fname) {
   return registry[fname];
} // FunctionRegistry::get_function
```

- Instantiierungen der generischen Klasse *map* können analog zu regulären Arrays verwendet werden, da der []-Operator für sie überladen wurde.
- *registry.find* liefert einen Iterator, der auf *registry.end* verweist, falls der gegebene Index bislang noch nicht belegt wurde.

FunctionRegistry.cpp

```
bool FunctionRegistry::is_known(std::string fname) const {
   return registry.find(fname) != registry.end();
} // FunctionRegistry::is_known
```

- Die STL-Container-Klassen wie *map* arbeiten mit Iteratoren.
- Iteratoren werden weitgehend wie Zeiger behandelt, d.h. sie können dereferenziert werden und vorwärts oder rückwärts zum nächsten oder vorherigen Element gerückt werden.
- Die *find*-Methode liefert nicht das gewünschte Objekt, sondern einen Iterator darauf.
- Die *end*-Methode liefert einen Iterator-Wert, der für das Ende steht.
- Durch einen Vergleich kann dann festgestellt werden, ob das gewünschte Objekt gefunden wurde.

TestFunctions.cpp

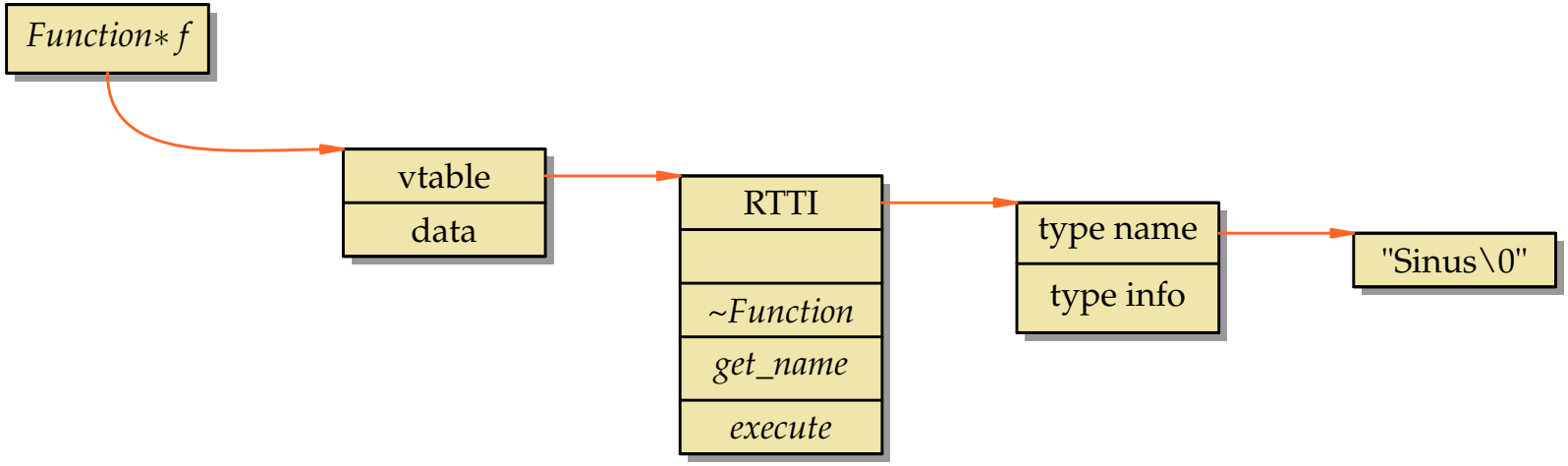
```
#include <iostream>
#include "Sinus.hpp"
#include "Cosinus.hpp"
#include "FunctionRegistry.hpp"

using namespace std;

int main() {
   FunctionRegistry registry;
   registry.add(new Sinus());
   registry.add(new Cosinus());

   string fname; double x;
   while (cout << ": " &&
          cin >> fname >> x) {
      if (registry.is_known(fname)) {
         Function* f(registry.get_function(fname));
         cout << f->execute(x) << endl;
      } else {
         cout << "Unknown function name: " << fname << endl;
      }
   }
} // main
```

Wie teuer ist dynamischer Polymorphismus? 192



- Nicht-polymorphe Methoden und reguläre Funktionen können in C++ direkt aufgerufen werden, d.h. die Sprungadresse ist direkt im Maschinen-Code verankert.
- Bei polymorphen Methoden muss zunächst hinter dem Objektzeiger, der sogenannte *vtable*-Zeiger geladen werden, hinter dem sich wiederum eine Liste mit Funktionszeigern zu den einzelnen Methoden verbirgt.
- Die Kosten einer polymorphen Methode belaufen sich entsprechend auf zwei nicht parallelisierbare Speicherzugriffe. Im ungünstigsten Falle (d.h. nichts davon ist im Cache) kostet dies bei aktuellen Systemen ca. 200 ns.

- Da der Aufruf polymorpher Methoden (also solcher Methoden, die mit **virtual** ausgezeichnet sind) zusätzliche Kosten während der Laufzeit verursacht, stellt sich die Frage, wann dieser Aufwand gerechtfertigt ist.
- Sinnvoll ist dynamischer Polymorphismus insbesondere, wenn
  - Container mit Zeiger oder Referenzen auf heterogene Objekte gefüllt werden, die alle eine Basisklasse gemeinsam haben oder
  - unbekannte Erweiterungen einer Basisklasse erst zur Laufzeit geladen werden.

```
Sinus* sf = dynamic_cast<Sinus*>(f);
if (sf) {
   cout << "appeared to be sin" << endl;
} else {
   cout << "appeared to be something else" << endl;
}
```

- Typ-Konvertierungen von Zeigern bzw. Referenzen abgeleiteter Klassen in Richtung zu Basisklassen ist problemlos möglich. Dazu wird kein besonderer Operator benötigt.
- In der umgekehrten Richtung kann eine Typ-Konvertierung mit Hilfe des **dynamic_cast**-Operators versucht werden.
- Diese Konvertierung ist erfolgreich, wenn es sich um einen Zeiger oder Referenz des gegebenen Typs handelt (oder eine Erweiterung davon).
- Im Falle eines Misserfolgs liefert **dynamic_cast** einen Nullzeiger.

```
#include <typeinfo>
// ...
const std::type_info& ti(typeid(*f));
cout << "type of f = " << ti.name() << endl;
```

- Seit C++11 gibt es im Rahmen des Standards *first-class*-Objekte für Typen.
- Der **typeid**-Operator liefert für einen Ausdruck oder einen Typen ein Typobjekt vom Typ *std*::**type_info**.
- *std*::**type_info** kann als Index für diverse Container-Klassen benutzt werden und es ist auch möglich, den Namen abzufragen.
- Wie der Name aber tatsächlich aussieht, ist der Implementierung überlassen. Dies muss nicht mit dem Klassennamen übereinstimmen.

- Die bisher zu C++ erschienen ISO-Standards (bis einschließlich ISO 14882-2012) sehen das dynamische Laden von Klassen nicht vor.
- Der POSIX-Standard (IEEE Standard 1003.1) schließt einige C-Funktionen ein, die das dynamische Nachladen von speziell übersetzten Modulen (*shared objects*) ermöglichen.
- Diese Schnittstelle kann auch von C++ aus genutzt werden, da grundsätzlich C-Funktionen auch von C++ aus verwendbar sind.
- Es sind hierbei allerdings Feinheiten zu beachten, da wegen des Überladens in C++ Symbolnamen auf der Ebene des Laders nicht mehr mit den in C++ verwendeten Namen übereinstimmen. Erschwerend kommt hinzu, dass die Abbildung von Namen in C++ in Symbolnamen – das sogenannte *name mangling* – nicht standardisiert ist.

```
#include <dlfcn.h>
#include <link.h>

void* dlopen(const char* pathname, int mode);
char* dlerror(void);
```

- *dlopen* lädt ein Modul (*shared object*, typischerweise mit der Dateiendung „.so“), dessen Dateiname bei *pathname* spezifiziert wird.
- Der Parameter *mode* legt zwei Punkte unabhängig voneinander fest:
  - Wann werden die Symbole aufgelöst? Entweder sofort (*RTLD_NOW*) oder so spät wie möglich (*RTLD_LAZY*). Letzteres wird normalerweise bevorzugt.
  - Sind die geladenen globalen Symbole für später zu ladende Module sichtbar (*RTLD_GLOBAL*) oder wird ihre Sichtbarkeit lokal begrenzt (*RTLD_LOCAL*)? Hier wird zur Vermeidung von Konflikten typischerweise *RTLD_LOCAL* gewählt.
- Wenn das Laden nicht klappt, dann kann *dlerror* aufgerufen werden, um eine passende Fehlermeldung abzurufen.

```
#include <dlfcn.h>

void* dlsym(void* restrict handle, const char* restrict name);
int dlclose(void* handle);
```

- Die Funktion *dlsym* erlaubt es, Symbolnamen in Adressen zu konvertieren. Im Falle von Funktionen lässt sich auf diese Weise ein Funktionszeiger gewinnen. Zu beachten ist hier, dass nur bei C-Funktionen davon ausgegangen werden kann, dass der C-Funktionsname dem Symbolnamen entspricht. Bei C++ ist das ausgeschlossen. Als *handle* wird der **return**-Wert von *dlopen* verwendet, *name* ist der Symbolname.
- Mit *dlclose* kann ein nicht mehr benötigtes Modul wieder entfernt werden.

```
extern "C" void do_something() {
   // beliebiger C++-Programmtext
}
```

- In C++ kann eine Funktion mit **extern** "C" ausgezeichnet werden.
- Diese Funktion ist dann von C aus unter ihrem Namen aufrufbar.
- Ein Überladen solcher Funktionen ist naturgemäß nicht möglich, da C dies nicht unterstützt.
- Innerhalb dieser Funktion sind allerdings beliebige C++-Konstrukte möglich.
- Ein solche C-Funktion kann benutzt werden, um ein Objekt der C++-Klasse zu konstruieren oder ein Objekt einer passenden Factory-Klasse zu erzeugen, mit der Objekte der eigentlichen Klasse konstruiert werden können.

Sinus.cpp

```
extern "C" Function* construct() {
   return new Sinus();
}
```

- Im Falle sogenannter Singleton-Objekte (d.h. Fälle, bei denen typischerweise pro Klasse nur ein Objekt erzeugt wird), genügt eine einfache Konstruktor-Funktion.
- Diese darf sogar einen global nicht eindeutigen Namen tragen – vorausgesetzt, wir laden das Modul mit der Option *RTLD_LOCAL*. Dann ist das entsprechende Symbol nur über den von *dlopen* zurückgelieferten Zeiger in Verbindung mit der *dlsym*-Funktion zugänglich.

DynFunctionRegistry.hpp

```
class DynFunctionRegistry {
   public:
      // constructors
      DynFunctionRegistry();
      DynFunctionRegistry(const std::string& dirname);

      void add(Function* f);
      bool is_known(const std::string& fname);
      Function* get_function(const std::string& fname);
   private:
      const std::string dir;
      std::map< std::string, Function* > registry;
      Function* dynload(const std::string& fname);
}; // class DynFunctionRegistry
```

- Neben dem Default-Konstruktor gibt es jetzt einen weiteren, der einen Verzeichnisnamen erhält, in dem die zu ladenden Module gesucht werden.
- Ferner kommt noch die private Methode *dynload* hinzu, deren Aufgabe es ist, ein Modul, das die angegebene Funktion implementiert, dynamisch nachzuladen und ein entsprechendes Singleton-Objekt zu erzeugen.

Dynamisches Nachladen einer Klasse

DynFunctionRegistry.cpp

```
typedef Function* FunctionConstructor();

Function* DynFunctionRegistry::dynload(const std::string& name) {
   std::string path(dir);
   if (path.size() > 0) path += "/";
   path += name; path += ".so";
   void* handle = dlopen(path.c_str(), RTLD_LAZY | RTLD_LOCAL);
   if (!handle) return 0;
   FunctionConstructor* constructor =
      (FunctionConstructor*) dlsym(handle, "construct");
   if (!constructor) {
      dlclose(handle); return 0;
   }
   return constructor();
}
```

- Zunächst wird aus *name* ein Pfad bestimmt, unter der das passende Modul abgelegt sein könnte.
- Dann wird mit *dlopen* versucht, es zu laden.
- Wenn dies erfolgreich war, wird mit Hilfe von *dlsym* die Adresse der *construct*-Funktion ermittelt und diese im Erfolgsfalle aufgerufen.

Abrufen einer Funktion

DynFunctionRegistry.cpp

```
Function* DynFunctionRegistry::get_function(const std::string& fname) {
   auto it(registry.find(fname));
   Function* f;
   if (it == registry.end()) {
      f = dynload(fname);
      if (f) {
         add(f);
         if (f->get_name() != fname) registry[fname] = f;
      }
   } else {
      f = it->second;
   }
   return f;
} // FunctionRegistry::get_function
```

- Innerhalb der *map*-Template-Klasse gibt es ebenfalls einen *iterator*-Typ, der hier mit dem Resultat von *find* initialisiert wird.
- Wenn dieser Iterator dereferenziert wird, liefert ein Paar mit den Komponenten *first* (Index) und *second* (eigentlicher Wert hinter dem Index).
- Falls der Name bislang nicht eingetragen ist, wird mit Hilfe von *dynload* versucht, das zugehörige Modul dynamisch nachzuladen.

DynFunctionRegistry.cpp

```
auto it(registry.find(fname));
```

- Beginnend mit C++11 kann bei einer Deklaration auf die Spezifikation eines Typs mit Hilfe des Schlüsselworts **auto** verzichtet werden, wenn sich der gewünschte Typ von der Initialisierung ableiten lässt.
- In diesem Beispiel muss nicht der lange Typname *std*::*map*< *std*::*string*, *Function*∗ >::*iterator* hingeschrieben werden, weil der Übersetzer das selbst automatisiert von dem Rückgabetyp von *registry*.*find*() ableiten kann.

- Generische Klassen und Funktionen, in C++ *templates* genannt, sind unvollständige Deklarationen bzw. Definitionen, die von nicht deklarierten Typparametern abhängen.
- Sie können nur in instantiierter Form verwendet werden, wenn alle Typparameter gegeben und deklariert sind.
- Unter bestimmten Umständen ist auch eine implizite Festlegung der Typparameter möglich, wenn sich dies aus dem Kontext ergibt.
- Generische Module wurden zuerst von Ada unterstützt (nicht in Kombination mit OO-Techniken) und später in Eiffel, einer statisch getypten OO-Sprache.
- Generische Klassen werden primär für Container-Klassen verwendet wie etwa in der STL, zunehmend aber auch für andere Anwendungen wie etwa der Metaprogrammierung.

- Templates ähneln teilweise den Makros, da
  - der Übersetzer den Programmtext des generischen Moduls erst bei einer Instantiierung vollständig analysieren und nach allen Fehlern durchsuchen kann und
  - für jede Instantiierung (mit unterschiedlichen Typparametern) Code zu generieren ist.
- Anders als bei Makros
  - müssen sich generische Module sich an die üblichen Regeln halten (korrekte Syntax, Sichtbarkeit, Typverträglichkeiten),
  - können entsprechend einige Fehler schon vor einer Instantiierung festgestellt werden und es
  - lässt sich die Code-Duplikation im Falle zweier Instanzen mit den gleichen Typparametern vermeiden.

```
class ListOfElements {
   // ...
   private:
      struct Linkable {
         Element element;
         Linkable* next;
      };
      Linkable* list;
};
```

- Diese Listenimplementierung speichert Objekte des Typs *Element*.
- Objekte, die einer von *Element* abgeleiteten Klasse angehören, können nur partiell (eben nur der Anteil von *Element*) abgesichert werden.
- Entsprechend müsste die Implementierung dieser Liste textuell dupliziert werden für jede zu unterstützende Variante des Datentyps *Element*.

```
class List {
   // ...
   private:
      struct Linkable {
         Element* element;
         Linkable* next;
      };
      Linkable* list;
};
```

- Wenn Zeiger oder Referenzen zum Einsatz kommen, können beliebige Erweiterungen von *Element* unterstützt werden.
- Generell stellt sich dann aber immer die Frage, wer für das Freigeben der Objekte hinter den Zeigern verantwortlich ist: Die Listenimplementierung oder der die Liste benutzende Klient?
- Die Anwendung der Liste für elementare Datentypen wie etwa **int** ist nicht möglich. Für Klassen, die keine Erweiterung von *Element* sind, müssten sogenannte Wrapper-Klassen konstruiert werden, die von *Element* abgeleitet werden und Kopien des gewünschten Typs aufnehmen können.

- Generell haben polymorphe Container-Klassen den Nachteil der mangelnden statischen Typsicherheit.
- Angenommen wir haben eine polymorphe Container-Klasse, die Zeiger auf Objekte unterstützt, die der Klasse A oder einer davon abgeleiteten Klasse unterstützen.
- Dann sei angenommen, dass wir nur Objekte der von A abgeleiteten Klasse B in dem Container unterbringen möchten. Ferner sei C eine andere von A abgeleitete Klasse, die jedoch nicht von B abgeleitet ist.
- Dann gilt:
  - Objekte der Klassen A und C können neben Objekten der Klasse B versehentlich untergebracht werden, ohne dass dies zu einem Fehler führt.
  - Wenn wir ein Objekt der Klasse B aus dem Container herausholen, ist eine Typkonvertierung unverzichtbar. Diese ist entweder prinzipiell unsicher oder kostet einen Test zur Laufzeit.
  - Entsprechend fatal wäre es, wenn Objekte der Klasse B erwartet werden, aber Objekte der Klassen A oder C enthalten sind.

Deklaration eines Templates


```
template<typename Element>
class List {
   public:
      // ...
      void add(const Element& element);
   private:
      struct Linkable {
         Element element;
         Linkable* next;
      };
      Linkable* list;
};
```

- Wenn der Klassendeklaration eine Template-Parameterliste vorangeht, dann wird daraus insgesamt die Deklaration eines Templates.
- Parameter bei Templates sind typischerweise von der Form **typename** *T*, aber C++ unterstützt auch andere Parameter, die beispielsweise die Dimensionierung eines Arrays bestimmen.

```
List< int > list; // select int as Element type
list.add(7);
```

- Templates werden instantiiert durch die Angabe des Klassennamens und den Parametern in gewinkelten Klammern.

Tail.cpp

```
#include <iostream>
#include <string>
#include "History.hpp"

using namespace std;

int main() {
   History< string > tail(10);
   string line;
   while (getline(cin, line)) {
      tail.add(line);
   }
   for (int i = tail.size() - 1; i >= 0; --i) {
      cout << tail[i] << endl;
   }
   return 0;
}
```

- Diese Anwendung gibt die letzten 10 Zeilen der Standardeingabe aus.
- *History* ist eine Container-Klasse, die sich nur die letzten n hinzugefügten Objekte merkt. Alle vorherigen Einträge werden rausgeworfen.

Tail.cpp

```
History< string > tail(10);
string line;
while (getline(cin, line)) {
   tail.add(line);
}
for (int i = tail.size() - 1; i >= 0; --i) {
   cout << tail[i] << endl;
}
return 0;
```

- Mit *History*< *string* > *tail*(10) wird die Template-Klasse *History* mit *string* als Typparameter instantiiert. Der Typparameter legt hier den Element-Typ des Containers fest.
- Der Konstruktor erwartet eine ganze Zahl als Parameter, der die Zahl zu speichernden Einträge bestimmt.
- Der []-Operator wurde hier überladen, um eine Notation analog zu Arrays zu erlauben. So steht *tail*[0] für das zuletzt hinzugefügte Objekt, *tail*[1] für das vorletzte usw.

History.hpp

```
#include <vector>
template<typename Item>
class History {
   public:
      // constructor
      History(unsigned int nitems);
      // accessors
      unsigned int max_size() const; // returns capacity
      unsigned int size() const; // returns # of items in buffer
      const Item& operator[](unsigned int i) const;
         // PRE: i >= 0 && i < size()
         // i = 0: return item added last
         // i = 1: return item before last item
      // mutators
      void add(const Item& item);
   private:
      std::vector< Item > items; // ring buffer with the last n items
      unsigned int index; // next item will be stored at items[index]
      unsigned int nof_items; // # of items in ring buffer so far
};
```

History.hpp

```
template<typename Item>
class History {
   // ...
};
```

- Typparameter bei Templates werden immer in der Form **typename** *T* spezifiziert. Zugelassen sind nicht nur Klassen, sondern auch elementare Datentypen wie etwa **int**.

History.hpp

```
    const Item& operator[](unsigned int i) const;
       // PRE: i >= 0 && i < size()
       // i = 0: return item added last
       // i = 1: return item before last item
```

- Per Typparameter eingeführte Klassen können innerhalb des Templates so verwendet werden, als wären sie bereits vollständig deklariert worden.
- Der []-Operator erhält einen Index als Parameter und liefert hier eine konstante Referenz zurück, die Veränderungen des Objekts nicht zulassen. Dies ist hier beabsichtigt, da eine *History* Objekte nur aufzeichnen, jedoch nicht verändern sollte.

History.hpp

```
private:
   std::vector< Item > items; // ring buffer with the last n items
   unsigned int index; // next item will be stored at items[index]
   unsigned int nof_items; // # of items in ring buffer so far
```

- Template-Klassen steht es frei, andere Templates zu verwenden und ggf. auch hierbei die eigenen Parameter zu verwenden.
- In diesem Beispiel wird ein Vektor mit dem Template-Parameter *Item* angelegt.

History.hpp

```
private:
   std::vector< Item > items; // ring buffer with the last n items
   unsigned int index; // next item will be stored at items[index]
   unsigned int nof_items; // # of items in ring buffer so far
```

- *vector* ist eine Template-Klasse aus der STL, die anders als die regulären Arrays in C++
  - nicht wie Zeiger behandelt werden,
  - sich die Dimensionierung merken und
  - in der Lage sind, die Zulässigkeit der Indizes zu überprüfen.
- Auf Basis der Template-Klasse *vector* lassen sich leicht andere Zuordnungen von Indizes zu zugehörigen Objekten umsetzen.

History.tpp

```
#include <cassert>

template<typename Item>
History<Item>::History(unsigned int nitems) :
      items(nitems), index(0), nof_items(0) {
   assert(nitems > 0);
} // History<Item>::History

template<typename Item>
unsigned int History<Item>::max_size() const {
   return items.size();
} // History<Item>::max_size

template<typename Item>
unsigned int History<Item>::size() const {
   return nof_items;
} // History<Item>::size
```

- Allen Methodendeklarationen, die zu einer Template-Klasse gehören, muss die Template-Deklaration vorangehen und der Klassenname ist mit der Template-Parameterliste zu erweitern.

History.tpp

```
template<typename Item>
void History<Item>::add(const Item& item) {
   items[index] = item;
   index = (index + 1) % items.size();
   if (nof_items < items.size()) {
      nof_items += 1;
   }
} // History<Item>::add
```

- *add* legt eine Kopie des übergebenen Objekts in der aktuellen Position im Ringpuffer ab.
- Die Template-Klasse *vector* aus der STL unterstützt ebenfalls den []-Operator.

History.tpp

```
template<typename Item>
const Item& History<Item>::operator[](unsigned int i) const {
   assert(i >= 0 && i < nof_items);
   // we are adding items.size to the left op of % to avoid
   // negative operands (effect not defined by ISO C++)
   return items[(items.size() + index - i - 1) % items.size()];
}; // History<Item>::operator[]
```

- Indizierungsoperatoren sollten die Gültigkeit der Indizes überprüfen, falls dies möglich und sinnvoll ist.
- *items.size*() liefert die Größe des Vektors, die vom *nitems*-Parameter beim Konstruktor abgeleitet wird.
- Da es sich bei *items* um einen Ringpuffer handelt, verwenden wir den Modulo-Operator, um den richtigen Index relativ zur aktuellen Position zu ermitteln.

- Template-Klassen können nicht ohne weiteres mit beliebigen Typparameter instantiiert werden.
- C++ verlangt, dass *nach* der Instantiierung die gesamte Template-Deklaration und alle zugehörigen Methoden zulässig sein müssen in C++.
- Entsprechend führt jede neuartige Instantiierung zur völligen Neuüberprüfung der Template-Deklaration und aller zugehörigen Methoden unter Verwendung der gegebenen Parameter.
- Daraus ergeben sich Abhängigkeiten, die ein Typ, der als Parameter bei der Instantiierung angegeben wird, einzuhalten sind.

- Folgende Abhängigkeiten sind zu erfüllen für den Typ-Parameter der Template-Klasse *History*:
  - *Default Constructor*: Dieser wird implizit von der Template-Klasse *vector* verwendet, um das erste Element im Array zu initialisieren.
  - *Copy Constructor*: Dieser wird ebenfalls implizit von *vector* verwendet, um alle weiteren Elemente in Abhängigkeit vom ersten Element zu initialisieren.
  - Zuweisungs-Operator: Ist notwendig, damit Elemente in und aus der Template-Klasse *History* kopiert werden können.
  - Destruktor: Dieser wird von der Template-Klasse *vector* verwendet für Elemente, die aus dem Ringpuffer fallen bzw. bei der Auflösung des gesamten Ringpuffers.

TemplateFailure.cpp

```
#include "History.hpp"

class Integer {
   public:
      Integer(int i) : integer(i) {};
   private:
      int integer;
};

int main() {
   History< Integer > integers(10);
}
```

- Hier fehlt ein Default-Konstruktor. Dieser wird auch nicht implizit erzeugt, da mit *Integer*(**int** *i*) ein Konstruktor bereits gegeben ist.
- Damit wird eine der Template-Abhängigkeiten von *History* nicht erfüllt.

```
thales$ make 2>&1 | fold -sw 80
g++ -Wall -g -std=gnu++11   -c -o TemplateFailure.o TemplateFailure.cpp
In file included from
/usr/local/gcc47/lib/gcc/i386-pc-solaris2.10/4.7.1/../../../../include/c++/4.7.1
/vector:63:0,
                 from History.hpp:8,
                 from TemplateFailure.cpp:1:
/usr/local/gcc47/lib/gcc/i386-pc-solaris2.10/4.7.1/../../../../include/c++/4.7.1
/bits/stl_construct.h: In instantiation of 'void std::_Construct(_T1*, _Args&&
...) [with _T1 = Integer; _Args = {}]':
/usr/local/gcc47/lib/gcc/i386-pc-solaris2.10/4.7.1/../../../../include/c++/4.7.1
/bits/stl_uninitialized.h:497:3:   required from 'static void
std::__uninitialized_default_n_1<_TrivialValueType>::__uninit_default_n(_Forward
Iterator, _Size) [with _ForwardIterator = Integer*; _Size = unsigned int; bool
_TrivialValueType = false]'
/usr/local/gcc47/lib/gcc/i386-pc-solaris2.10/4.7.1/../../../../include/c++/4.7.1
/bits/stl_uninitialized.h:545:7:   required from 'void
std::__uninitialized_default_n(_ForwardIterator, _Size) [with _ForwardIterator
= Integer*; _Size = unsigned int]'
/usr/local/gcc47/lib/gcc/i386-pc-solaris2.10/4.7.1/../../../../include/c++/4.7.1
/bits/stl_uninitialized.h:607:7:   required from 'void
std::__uninitialized_default_n_a(_ForwardIterator, _Size, std::allocator<_Tp>&)
[with _ForwardIterator = Integer*; _Size = unsigned int; _Tp = Integer]'
/usr/local/gcc47/lib/gcc/i386-pc-solaris2.10/4.7.1/../../../../include/c++/4.7.1
```

```
/bits/stl_vector.h:1191:2:   required from 'void std::vector<_Tp,
_Alloc>::_M_default_initialize(std::vector<_Tp, _Alloc>::size_type) [with _Tp =
Integer; _Alloc = std::allocator<Integer>; std::vector<_Tp, _Alloc>::size_type
= unsigned int]'
/usr/local/gcc47/lib/gcc/i386-pc-solaris2.10/4.7.1/../../../../include/c++/4.7.1
/bits/stl_vector.h:268:9:   required from 'std::vector<_Tp,
_Alloc>::vector(std::vector<_Tp, _Alloc>::size_type) [with _Tp = Integer;
_Alloc = std::allocator<Integer>; std::vector<_Tp, _Alloc>::size_type =
unsigned int]'
History.tpp:9:43:   required from 'History<Item>::History(unsigned int) [with
Item = Integer]'
TemplateFailure.cpp:11:34:   required from here
/usr/local/gcc47/lib/gcc/i386-pc-solaris2.10/4.7.1/../../../../include/c++/4.7.1
/bits/stl_construct.h:77:7: error: no matching function for call to
'Integer::Integer()'
/usr/local/gcc47/lib/gcc/i386-pc-solaris2.10/4.7.1/../../../../include/c++/4.7.1
/bits/stl_construct.h:77:7: note: candidates are:
TemplateFailure.cpp:5:7: note: Integer::Integer(int)
TemplateFailure.cpp:5:7: note:   candidate expects 1 argument, 0 provided
TemplateFailure.cpp:3:7: note: constexpr Integer::Integer(const Integer&)
TemplateFailure.cpp:3:7: note:   candidate expects 1 argument, 0 provided
TemplateFailure.cpp:3:7: note: constexpr Integer::Integer(Integer&&)
TemplateFailure.cpp:3:7: note:   candidate expects 1 argument, 0 provided
make: *** [TemplateFailure.o] Error 1
thales$
```

- Bei der Übersetzung von Templates gibt es ein schwerwiegendes Problem:
  - Dort, wo die Methoden des Templates stehen (hier etwa in *History.tpp*), ist nicht bekannt, welche Instanzen benötigt werden.
  - Dort, wo das Template instantiiert wird (hier etwa in *Tail.cpp*), sind die Methodenimplementierungen des Templates unbekannt, da zwar *History.hpp* reinkopiert wurde, aber eben nicht ohne weiteres *History.tpp*.
- Folgende Fragen stellen sich:
  - Wie kann der Übersetzer die benötigten Template-Instanzen generieren?
  - Wie kann vermieden werden, dass die gleiche Template-Instanz mehrfach generiert wird?

- Beim Inclusion-Modell wird mit Hilfe einer **#include**-Anweisung auch die Methoden-Implementierung hereinkopiert, so dass sie beim Übersetzung der instantiierenden Module sichtbar ist.
- Entsprechend wird in *History.hpp* am Ende auch noch *History.tpp* mit **#include** hereinkopiert. (Deswegen auch die Datei-Endung „.tpp“ anstelle von „.cpp“.)
- Das funktioniert grundsätzlich bei allen C++-Übersetzern, aber es führt im Normalfall zu einer Code-Vermehrung, wenn das gleiche Template in unterschiedlichen Quellen in gleicher Weise instantiiert wird.
- Das Borland-Modell sieht hier eine zusätzliche Verwaltung vor, die die Mehrfach-Generierung unterbindet.
- Der *gcc* unterstützt das Borland-Modell, wenn jeweils die Option *-frepo* gegeben wird, die dann die Verwaltungsinformationen in Dateien mit der Endung *rpo* unterbringt. Dies erfordert die Zusammenarbeit mit dem Linker und funktioniert beim *gcc* somit nur mit dem GNU-Linker.

- Der elegantere Ansatz vermeidet zusätzliche **#include**-Anweisungen. Entsprechend muss der Übersetzer selbst die zugehörige Quelle finden.
- Hierfür gibt es kein standardisiertes Vorgehen. Jeder Übersetzer, der dieses Modell unterstützt, hat dafür eigene Verwaltungsstrukturen.
- *gcc* unterstützt dieses Modell jedoch nicht.
- Der von Sun ausgelieferte C++-Übersetzer (bei uns mit *CC* aufzurufen) folgt diesem Modell.
- Im C++-Standard von 2003 wurde dies explizit über das Schlüsselwort **export** unterstützt.
- Da dies jedoch von kaum jemanden implementiert worden ist, wurde dies bei C++11 gestrichen. Entsprechend ist das Inclusion-Modell das einzige, das sich in der Praxis durchgehend etabliert hat.

```
template class History<string>;
```

- Die Kontrolle darüber, genau wann und wo der Code für eine konkrete Template-Instantiierung zu erzeugen ist, kann mit Hilfe expliziter Instantiierungen kontrolliert werden.
- Eine explizite Instantiierung wiederholt die Template-Deklaration ohne das Innenleben, nennt aber die Template-Parameter.
- Dann wird an dieser Stelle der entsprechende Code erzeugt.
- Das darf dann aber nur einmal im gesamten Programm erfolgen. Sonst gibt es Konflikte beim Zusammenbau.
- Seit C++11 ist es möglich, so eine explizite Instantiierung mit dem Schlüsselwort **extern** zu versehen. Dann wird die Generierung des entsprechenden Codes unterdrückt und stattdessen die anderswo explizit instantiierte Fassung verwendet.

```
extern template class History<string>;
```

- Der Vorteil expliziter Instantiierungen liegt in der Vermeidung redundanten Codes, ohne sich auf entsprechende implementierungsabhängige Unterstützungen des Übersetzers verlassen zu müssen.
- Ein weiterer Vorzug ist die kürzere Übersetzungszeit, da die Template-Implementierung dann nur noch dort benötigt wird, wo explizite Instantiierungen vorgenommen werden.
- Diese Vorgehensweise nötigt den Programmierer jedoch, selbst einen Überblick zu behalten, welche Instantiierungen alle benötigt werden. Das wird sehr schnell sehr unübersichtlich.
- Das liegt an der sogenannten *one-definition-rule* (ODR), d.h. Objekte dürfen beliebig oft deklariert, aber global nur einmal definiert werden. Bei impliziten Instantiierungen ist das ein Problem des Übersetzters, bei expliziten Instantiierungen übernimmt der Programmierer die Verantwortung.
- Diese Technik wird daher typischerweise nur in isolierten Fällen benutzt.

History.hpp

```
template<typename Item>
class History {
   public:
      History(unsigned int nitems) : items(nitems), index(0), nof_items(0) {
         assert(nitems > 0);
      }
      unsigned int max_size() const { // returns capacity
         return items.size();
      }
      unsigned int size() const { // returns # of items in buffer
         return nof_items;
      }
      const Item& operator[](unsigned int i) const {
         assert(i >= 0 && i < nof_items);
         return items[(items.size() + index - i - 1) % items.size()];
      }
      void add(const Item& item) {
         items[index] = item;
         index = (index + 1) % items.size();
         if (nof_items < items.size()) {
            nof_items += 1;
         }
      }
   private:
      std::vector< Item > items; // ring buffer with the last n items
      unsigned int index; // next item will be stored at items[index]
      unsigned int nof_items; // # of items in ring buffer so far
};
```

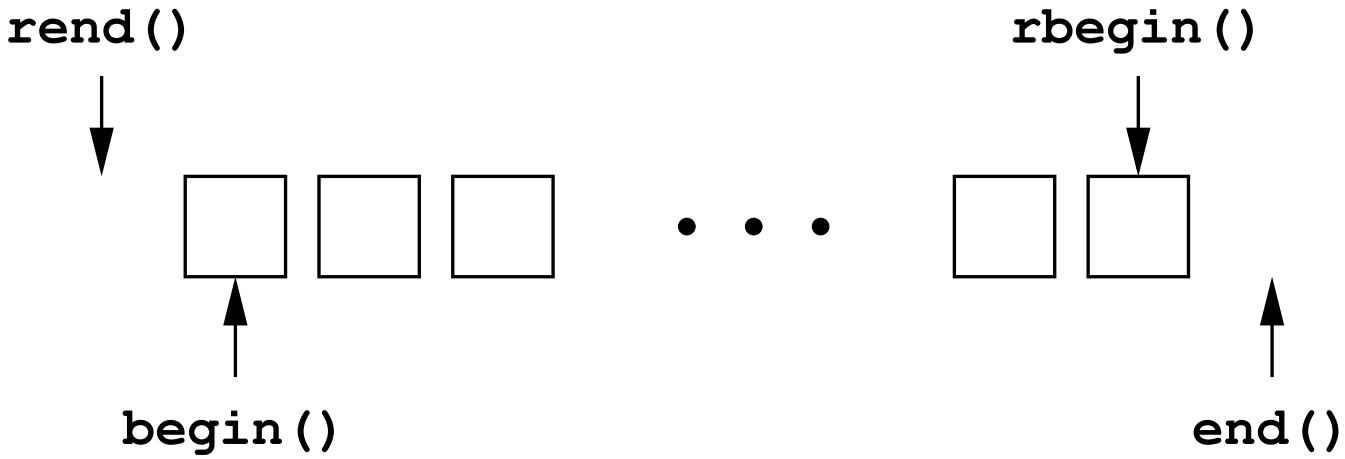
- Wenn eine Methode direkt in einer Klassendeklaration definiert wird, d.h. ihre Implementierung integriert ist, dann ist sie automatisch als **inline** deklariert. (Das ist äquivalent zu einer getrennten Implementierung, bei der das Schlüsselwort **inline** angegeben wird.)
- **inline**-Methoden geben dem Übersetzer die Möglichkeit, den Code unmittelbar beim Aufruf einer Methode zu expandieren. (Voraussetzung ist hier, dass sich alles statisch ableiten lässt.)
- Wenn alle Methoden einer Template-Klasse **inline** sind, entfällt die Notwendigkeit einer „.tpp“-Datei und entsprechend fällt die gesamte Problematik weg.
- Ferner wird die Laufzeiteffizienz des erzeugten Codes verbessert (Methodenaufruf und Parameterübergabe fallen weg).
- Allerdings wird der erzeugte Code umfangreicher und die Übersetzungszeiten nehmen deutlich zu.
- Große Teile der Standard-Bibliothek von C++ sind auf diese Weise realisiert.

Die Standard-Template-Library (STL) bietet eine Reihe von Template-Klassen für Container, eine allgemeine Schnittstelle für Iteratoren und eine Sammlung von Algorithmen an.

Container-Klassen der STL:

Implementierungstechnik	Name der Template-Klasse	
Lineare Listen	*deque*	*queue*
	list	*stack*
Dynamische Arrays	*string*	*vector*
Balancierte binäre	*set*	*multiset*
sortierte Bäume	*map*	*multimap*
Hash-Verfahren	*unordered_set*	*unordered_multiset*
	unordered_map	*unordered_multimap*

- All die genannten Container-Klassen mit Ausnahme der *unordered*-Varianten besitzen eine Ordnung. Eine weitere Ausnahme sind hier noch die Template-Klassen *multiset* und *multimap*, die keine definierte Ordnung für mehrfach vorkommende Schlüssel haben.
- Die Unterstützung von Hash-Tabellen (*unordered_map* etc.) ist erst mit C++11 gekommen. Zuvor sah die Standard-Bibliothek keine Hash-Tabellen vor.
- Gelegentlich gab es früher den Standard ergänzende Bibliotheken, die dann andere Namen wie etwa *hash_set*, *hash_map* usw. haben.



iterator	bidirektionaler Iterator, der sich an der Ordnung des Containers orientiert (soweit eine Ordnung existiert)
const_iterator	analog zu *iterator*, jedoch sind schreibende Zugriffe auf die referenzierten Elemente nicht möglich
reverse_iterator	bidirektionaler Iterator, dessen Richtung der Ordnung des Containers entgegen gesetzt ist
const_reverse_iterator	analog zu *reverse_iterator*, jedoch sind schreibende Zugriffe auf die referenzierten Elemente nicht möglich

- *iterator* und *reverse_iterator* erlauben Schreibzugriffe auf die referenzierten Elemente. Der Schlüssel ist jedoch davon ausgenommen im Falle assoziativer Container wie *set*, *map*, *unordered_set* usw.
- Keiner der Iteratoren ist *robust*, d.h. Entfernungen oder Einfügungen aus oder in den Container führen zu ungültigen Iteratoren (mehr dazu im ISO-Standard selbst).
- Es gibt noch weitere Iteratoren, die hier nicht vorgestellt werden: Unidirektionale Iteratoren, Einfüge-Iteratoren und Stream-Iteratoren.

Obwohl Iteratoren keine Zeiger im eigentlichen Sinne sind, werden die entsprechenden Operatoren weitgehend unterstützt, so dass sich Zeiger und Iteratoren in der Anwendung ähneln.
Sei *Iterator* der Typ eines Iterators, *it* ein Iterator dieses Typs und *Element* der Element-Typ des Containers, mit dem *Iterator* und *it* verbunden sind. Ferner sei *member* ein Datenfeld von *Element*.

Operator	Rückgabe-Typ	Beschreibung
$*it$	*Element&*	Zugriff auf ein Element
$it->member$	Typ von *member*	Zugriff auf ein Datenfeld
$++it$	*Iterator*	Iterator vorwärts weitersetzen
$--it$	*Iterator*	Iterator rückwärts weitersetzen

- Es ist zu beachten, dass ein Iterator *in* einen Container zeigen muss, damit auf ein Element zugegriffen werden kann, d.h. die Rückgabe-Werte von *end()* und *rend()* dürfen nicht dereferenziert werden.
- Analog ist es auch nicht gestattet, Zeiger mehr als einen Schritt jenseits der Container-Grenzen zu verschieben.
- $--it$ darf den Container nicht verlassen, nicht einmal um einen einzelnen Schritt.
- Iteratoren unterstützen Default-Konstruktoren, Kopierkonstruktoren, Zuweisungen, $==$ und $!=$.

Iteratoren, die von *vector*, *deque* und *string* geliefert werden, erlauben einen indizierten Zugriff:

Operator	Rückgabe-Typ	Beschreibung
it+*n*	*Iterator*	liefert einen Iterator zurück, der *n* Schritte relativ zu *it* vorangegangen ist
it−*n*	*Iterator*	liefert einen Iterator zurück, der *n* Schritte relativ zu *it* zurückgegangen ist
it[*n*]	*Element*&	äquivalent zu $*(it+n)$
$it1 < it2$	**bool**	äquivalent zu $it2 - it1 > 0$
$it2 < it1$	**bool**	äquivalent zu $it1 - it2 > 0$
$it1 <= it2$	**bool**	äquivalent zu $!(it1 > it2)$
$it1 >= it2$	**bool**	äquivalent zu $!(it1 < it2)$
$it1 - it2$	*Distance*	Abstand zwischen *it1* und *it2*; dies liefert einen negativen Wert, falls $it1 < it2$

Eine gute Container-Klassenbibliothek strebt nach einer übergreifenden Einheitlichkeit, was sich auch auf die Methodennamen bezieht:

Methode	Beschreibung
begin()	liefert einen Iterator, der auf das erste Element verweist
end()	liefert einen Iterator, der hinter das letzte Element zeigt
rbegin()	liefert einen rückwärts laufenden Iterator, der auf das letzte Element verweist
rend()	liefert einen rückwärts laufenden Iterator, der vor das erste Element zeigt
empty()	ist wahr, falls der Container leer ist
size()	liefert die Zahl der Elemente
clear()	leert den Container
erase(*it*)	wirft das Element aus dem Container heraus, auf das *it* zeigt

Methode	Beschreibung	unterstützt von
front()	liefert das erste Element eines Containers	*vector*, *list*, *deque*
back()	liefert das letzte Element eines Containers	*vector*, *list*, *deque*
push_front()	fügt ein Element zu Beginn ein	*list*, *deque*
push_back()	hängt ein Element an das Ende an	*vector*, *list*, *deque*
pop_front()	entfernt das erste Element	*list*, *deque*
pop_back()	entfernt das letzte Element	*vector*, *list*, *deque*
[*n*]	liefert das *n*-te Element	*vector*, *deque*
at(*n*)	liefert das *n*-te Element	*vector*, *deque*

Vorteile:

- Erlaubt indizierten Zugriff in konstanter Zeit
- Einfüge- und Lösch-Operationen an den Enden mit konstanten Aufwand.

Nachteile:

- Einfüge- und Lösch-Operationen in der Mitte haben einen linearen Aufwand.
- Kein Aufteilen, kein Zusammenlegen (im Vergleich zu Listen).

Vorteile:

- Überall konstanter Aufwand beim Einfügen und Löschen. (Dies schließt nicht das Finden eines Elements in der Mitte ein.)
- Unterstützung des Zusammenlegens von Listen, des Aufteilens und des Umdrehens.

Nachteile:

- Kein indizierter Zugriff. Entsprechend ist der Suchaufwand linear.

Vorteile:

- Schneller indizierter Zugriff (theoretisch kann dies gleichziehen mit den eingebauten Arrays).
- Konstanter Aufwand für Einfüge- und Löschoperationen am Ende.

Nachteile:

- Weder *push_front* noch *pop_front* werden unterstützt.

Operation	Rückgabe-Typ	Beschreibung
empty()	**bool**	liefert *true*, falls der Container leer ist
size()	*size_type*	liefert die Zahl der enthaltenen Elemente
top()	*value_type*&	liefert das letzte Element; eine **const**-Variante wird ebenfalls unterstützt
push(*element*)	**void**	fügt ein Element hinzu
pop()	**void**	entfernt ein Element

Es gibt vier sortierte assoziative Container-Klassen in der STL:

	Schlüssel/Werte-Paare	Nur Schlüssel
Eindeutige Schlüssel	*map*	*set*
Mehrfache Schlüssel	*multimap*	*multiset*

- Der Aufwand der Suche nach einem Element ist logarithmisch.
- Kandidaten für die Implementierung sind AVL-Bäume oder Red-Black-Trees.
- Voreinstellungsgemäß wird $<$ für Vergleiche verwendet, aber es können auch andere Vergleichs-Operatoren spezifiziert werden. Der $==$-Operator wird nicht verwendet. Stattdessen wird die Äquivalenzrelation von $<$ abgeleitet, d.h. a und b werden dann als äquivalent betrachtet, falls $!(a < b) \&\& !(b < a)$.
- Alle assoziativen Container haben die Eigenschaft gemeinsam, dass vorwärts laufende Iteratoren die Schlüssel in monotoner Reihenfolge entsprechend des Vergleichs-Operators durchlaufen. Im Falle von Container-Klassen, die mehrfach vorkommende Schlüssel unterstützen, ist diese Reihenfolge nicht streng monoton.

- Assoziative Container mit eindeutigen Schlüsseln akzeptieren Einfügungen nur, wenn der Schlüssel bislang noch nicht verwendet wurde.
- Im Falle von *map* und *multimap* ist jeweils ein Paar, bestehend aus einem Schlüssel und einem Wert zu liefern. Diese Paare haben den Typ *pair*<**const** *Key*, *Value*>, der dem Typ *value_type* der instanziierten Template-Klasse entspricht.
- Der gleiche Datentyp für Paare wird beim Dereferenzieren von Iteratoren bei *map* und *multimap* geliefert.
- Das erste Feld des Paares (also der Schlüssel) wird über den Feldnamen *first* angesprochen; das zweite Feld (also der Wert) ist über den Feldnamen *second* erreichbar. Ja, die Namen sind unglücklich gewählt.

- Die Template-Klasse *map* unterstützt den []-Operator, der den Datentyp für Paare vermeidet, d.h. Zuweisungen wie etwa *mymap*[*key*] = *value* sind möglich.
- Jedoch ist dabei Vorsicht geboten: Es gibt keine **const**-Variante des []-Operators und ein Zugriff auf *mymap*[*key*] führt zum Aufruf des Default-Konstruktors für das Element, wenn es bislang noch nicht existierte. Entsprechend ist der []-Operator nicht zulässig in **const**-Methoden und stattdessen erfolgt der Zugriff über einen *const_iterator*.

Methode	Beschreibung
insert(*t*)	Einfügen eines Elements: *pair*<*iterator*,**bool**> wird von *map* und *set* geliefert, wobei der Iterator auf das Element mit dem Schlüssel verweist und der **bool**-Wert angibt, ob die Einfüge-Operation erfolgreich war oder nicht. Bei *multiset* und *multimap* wird nur ein Iterator auf das neu hinzugefügte Element geliefert.
insert(*it*,*t*)	Analog zu *insert*(*t*). Falls das neu einzufügende Element sich direkt hinter *t* einfügen lässt, erfolgt die Operation mit konstantem Aufwand.
erase(*k*)	Entfernt alle Elemente mit dem angegebenen Schlüssel.
erase(*it*)	Entfernt das Element, worauf *it* zeigt.
erase(*it1*, *it2*)	Entfernt alle Elemente aus dem Bereich $[it1, it2)$.

Methode	Beschreibung
find(k)	Liefert einen Iterator, der auf ein Element mit dem gewünschten Schlüssel verweist. Falls es keinen solchen Schlüssel gibt, wird *end()* zurückgeliefert.
count(k)	Liefert die Zahl der Elemente mit einem zu k äquivalenten Schlüssel. Dies ist insbesondere bei *multimap* und *multiset* sinnvoll.
lower_bound(k)	Liefert einen Iterator, der auf das erste Element verweist, dessen Schlüssel nicht kleiner als k ist.
upper_bound(k)	Liefert einen Iterator, der auf das erste Element verweist, dessen Schlüssel größer als k ist.

Es gibt vier unsortierte assoziative Container-Klassen in der STL:

	Schlüssel/Werte-Paare	Nur Schlüssel
Eindeutige Schlüssel	*unordered_map*	*unordered_set*
Mehrfache Schlüssel	*unordered_multimap*	*unordered_multiset*

- Die unsortierten assoziativen Container werden mit Hilfe von Hash-Organisationen implementiert.
- Der Standard sichert zu, dass der Aufwand für das Suchen und das Einfügen im Durchschnittsfall konstant sind, im schlimmsten Fall aber linear sein können (wenn etwa alle Objekte den gleichen Hash-Wert haben).
- Für den Schlüsseltyp muss es eine Hash-Funktion geben und einen Operator, der auf Gleichheit testet.
- Die Größe der Bucket-Tabelle wird dynamisch angepasst. Eine Umorganisation hat linearen Aufwand.

- Für die elementaren Datentypen einschließlich der Zeigertypen darauf und einigen von der Standard-Bibliothek definierten Typen wie *string* ist eine Hash-Funktion bereits definiert.
- Bei selbst definierten Schlüsseltypen muss dies nachgeholt werden. Der Typ der Hash-Funktion muss dann als dritter Template-Parameter angegeben werden und die Hash-Funktion als weiterer Parameter beim Konstruktor.
- Hierzu können aber die bereits vordefinierten Hash-Funktionen verwendet und typischerweise mit dem „$\widehat{\ }$“-Operator verknüpft werden.

Persons.cpp

```
struct Name {
   string first;
   string last;
   Name(const string first_, const string last_) :
      first(first_), last(last_) {
   }
   bool operator==(const Name& other) const {
      return first == other.first && last == other.last;
   }
};

struct NameHash {
   size_t operator()(const Name& name) const {
      return hash<string>()(name.first) ^ hash<string>()(name.last);
   }
};
```

- Damit ein Datentyp als Schlüssel für eine Hash-Organisation genutzt werden kann, müssen der „==“-Operator und eine Hash-Funktion gegeben sein.

Persons.cpp

```
size_t operator()(const Name& name) const {
   return hash<string>()(name.first) ^ hash<string>()(name.last);
}
```

- *hash*<*string*>() erzeugt ein temporäres Hash-Funktionsobjekt, das einen Funktions-Operator mit einem Parameter (vom Typ *string*) anbietet, der den Hash-Wert (Typ *size_t*) liefert.
- Hash-Werte werden am besten mit dem XOR-Operator „^“ verknüpft.
- Eine Hash-Funktion muss immer den gleichen Wert für den gleichen Schlüssel liefern.
- Für zwei verschiedene Schlüssel *k1* und *k2* sollte die Wahrscheinlichkeit, dass die entsprechenden Hash-Werte gleich sind, sich 1.0 / *numeric_limits*<*size_t*>::*max*() nähern.

Persons.cpp

```
int main() {
   unordered_map<Name, string, NameHash> address(32, NameHash());
   address[Name("Marie", "Maier")] = "Ulm";
   address[Name("Hans", "Schmidt")] = "Neu-Ulm";
   address[Name("Heike", "Vogel")] = "Geislingen";
   string first; string last;
   while (cin >> first >> last) {
      auto it = address.find(Name(first, last));
      if (it != address.end()) {
         cout << it->second << endl;
      } else {
         cout << "Not found." << endl;
      }
   }
}
```

- Der erste Parameter beim Konstruktor für Hash-Organisationen legt die initiale Größe der Bucket-Tabelle fest, der zweite spezifiziert die gewünschte Hash-Funktion.

- Template-Container-Klassen benutzen implizit viele Methoden und Operatoren für ihre Argument-Typen.
- Diese ergeben sich nicht aus der Klassendeklaration, sondern erst aus der Implementierung der Template-Klassenmethoden.
- Da die implizit verwendeten Methoden und Operatoren für die bei dem Template als Argument übergebenen Klassen Voraussetzung sind, damit diese verwendet werden können, wird von Template-Abhängigkeiten gesprochen.
- Da diese recht unübersichtlich sind, erlauben Test-Templateklassen wie die nun vorzustellende *TemplateTester*-Klasse eine Analyse, welche Operatoren oder Methoden wann aufgerufen werden.

TemplateTester.hpp

```
template<class BaseType>
class TemplateTester {
   public:
      // constructors
      TemplateTester();
      TemplateTester(const TemplateTester& orig);
      TemplateTester(const BaseType& val);

      // destructor
      ~TemplateTester();

      // operators
      TemplateTester& operator=(const TemplateTester& orig);
      TemplateTester& operator=(const BaseType& val);
      bool operator<(const TemplateTester& other) const;
      bool operator<(const BaseType& val) const;
      operator BaseType() const;

   private:
      static int instanceCounter; // gives unique ids
      int id; // id of this instance
      BaseType value;
}; // class TemplateTester
```

TemplateTester.tpp

```
template<class BaseType>
TemplateTester<BaseType>::TemplateTester() :
      id(instanceCounter++) {
   std::cerr << "TemplateTester: CREATE #" << id <<
      " (default constructor)" << std::endl;
} // default constructor
```

- Alle Methoden und Operatoren von *TemplateTester* geben Logmeldungen auf *cerr* aus.
- Die *TemplateTester*-Klasse ist selbst eine Wrapper-Template-Klasse um *BaseType* und bietet einen Konstruktor an, der einen Wert des Basistyps akzeptiert und einen dazu passenden Konvertierungs-Operator.
- Die Klassen-Variable *instanceCounter* erlaubt die Identifikation individueller Instanzen in den Logmeldungen.

Test der *list*-Klasse

TestList.cpp

```
typedef TemplateTester<int> Test;
list<Test> myList;
// put some values into the list
for (int i = 0; i < 2; ++i) {
   myList.push_back(i);
}
// iterate through the list
for (int val: myList) {
   cout << "Found " << val << " in the list." << endl;
}
```

```
thales$ TestList >/dev/null
TemplateTester: CREATE #0 (constructor with parameter 0)
TemplateTester: CREATE #1 (copy constructor of 0)
TemplateTester: DELETE #0
TemplateTester: CREATE #2 (constructor with parameter 1)
TemplateTester: CREATE #3 (copy constructor of 2)
TemplateTester: DELETE #2
TemplateTester: CONVERT #1 to 0
TemplateTester: CONVERT #3 to 1
TemplateTester: DELETE #1
TemplateTester: DELETE #3
thales$
```

Test der *vector*-Klasse

TestVector.cpp

```
typedef TemplateTester<int> Test;
vector<Test> myVector(2);
// put some values into the vector
for (int i = 0; i < 2; ++i) {
   myVector[i] = i;
}
// print all values of the vector
for (int i = 0; i < 2; ++i) {
   cout << myVector[i] << endl;
}
```

```
dublin$ testVector >/dev/null
TemplateTester: CREATE #0 (default constructor)
TemplateTester: CREATE #1 (copy constructor of 0)
TemplateTester: CREATE #2 (copy constructor of 0)
TemplateTester: DELETE #0
TemplateTester: ASSIGN value 0 to #1
TemplateTester: ASSIGN value 1 to #2
TemplateTester: CONVERT #1 to 0
TemplateTester: CONVERT #2 to 1
TemplateTester: DELETE #1
TemplateTester: DELETE #2
dublin$
```

Test der *map*-Klasse

TestMap.cpp

```
typedef TemplateTester<int> Test;
map<int, Test> myMap;

// put some values into the map
for (int i = 0; i < 2; ++i) {
   myMap[i] = i;
}
```

```
TemplateTester: CREATE #0 (default constructor)
TemplateTester: CREATE #1 (copy constructor of 0)
TemplateTester: CREATE #2 (copy constructor of 1)
TemplateTester: DELETE #1
TemplateTester: DELETE #0
TemplateTester: ASSIGN value 0 to #2
TemplateTester: CREATE #3 (default constructor)
TemplateTester: CREATE #4 (copy constructor of 3)
TemplateTester: CREATE #5 (copy constructor of 4)
TemplateTester: DELETE #4
TemplateTester: DELETE #3
TemplateTester: ASSIGN value 1 to #5
TemplateTester: CONVERT #2 to 0
TemplateTester: CONVERT #5 to 1
TemplateTester: DELETE #5
TemplateTester: DELETE #2
```

Test der Indizierung bei *map*

TestMapIndex.cpp

```
typedef TemplateTester<int> Test;
typedef map<Test, int> MyMap; MyMap myMap;
for (int i = 0; i < 2; ++i) myMap[i] = i;
for (const auto& pair: myMap) {
   cout << pair.second << endl;
}
```

```
TemplateTester: CREATE #0 (constructor with parameter 0)
TemplateTester: CREATE #1 (copy constructor of 0)
TemplateTester: CREATE #2 (copy constructor of 1)
TemplateTester: DELETE #1
TemplateTester: DELETE #0
TemplateTester: CREATE #3 (constructor with parameter 1)
TemplateTester: COMPARE #2 with #3
TemplateTester: CREATE #4 (copy constructor of 3)
TemplateTester: COMPARE #2 with #4
TemplateTester: COMPARE #4 with #2
TemplateTester: CREATE #5 (copy constructor of 4)
TemplateTester: DELETE #4
TemplateTester: DELETE #3
TemplateTester: DELETE #5
TemplateTester: DELETE #2
```

- Gelegentlich wird über die Verwendung von Kopierkonstruktoren oder Zuweisungen der Inhalt eines Objekts aufwendig kopiert (*deep copy*), worauf kurz danach das Original eliminiert wird.
- Das Problem tritt bei temporären Objekten auf. Da auch bei STL-Containern nicht wenig mit temporäre Objekte zum Einsatz kommen, ist dies u.U. recht teuer.
- Hier wäre es besser, wenn wir einfach die (möglicherweise sehr umfangreiche) interne Datenstruktur einfach „umhängen“ könnten.
- Beginnend mit C++11 gibt es einen weiteren Referenztyp, der auch temporäre Objekte unterstützt. Dieser Referenztyp verwendet zwei Und-Zeichen statt einem: „&&“.
- Diese Technik kommt auch bei *std*::*swap* zum Einsatz.

Trie.hpp

```
Trie(Trie&& other) :
      root(other.root), number_of_objects(other.number_of_objects) {
   // unlink tree from rvalue-referenced object other
   other.root = 0;
   other.number_of_objects = 0;
}
```

- Anders als beim Kopierkonstruktor übernimmt der Verlagerungskonstruktor den Objekt-Inhalt von dem übergebenen Objekt und initialisiert das referenzierte Objekt auf den leeren Zustand.
- Letzteres ist zwingend notwendig, da in jedem Falle anschließend noch der Dekonstruktor für das referenzierte Objekt aufgerufen wird.
- Da das referenzierte Objekt hier verändert wird, entfällt die Angabe von **const**.

Trie.hpp

```
Trie& operator=(Trie&& other) {
   delete root;
   root = other.root;
   number_of_objects = other.number_of_objects;
   other.root = 0;
   other.number_of_objects = 0;
   return *this;
}
```

- Analog kann bei der Zuweisung auch der Fall unterstützt werden, dass wir den Inhalt eines temporären Objekts übernehmen.
- Da das temporäre Objekt nicht das eigene sein kann, entfällt hier der entsprechende Test.

Trie.hpp

```
friend void swap(Trie& first, Trie& other) {
   std::swap(first.root, other.root);
   std::swap(first.number_of_objects, other.number_of_objects);
}
```

- Noch eleganter wird das alles, wenn eine *swap*-Methode eingeführt wird. Ihre Aufgabe ist es, den Inhalt der beiden Argumente auszutauschen.
- Es gibt bereits eine *std*::*swap*-Funktion, die per **#include** <utility> zur Verfügung steht.
- Für die elementaren Datentypen ist sie bereits definiert, für selbst-definierte Klassen kann sie (wie hier) als normale Funktion definiert werden, die dank der **friend**-Deklaration vollen Zugang zu den privaten Daten hat.

Trie.hpp

```
Trie(Trie&& other) : Trie() {
   swap(*this, other);
}
// ...
Trie& operator=(Trie other) {
   swap(*this, other);
   return *this;
}
```

- Wenn die passende *swap*-Funktion zur Verfügung steht, lassen sich der Verlagerungs-Konstruktor und die Zuweisung dramatisch vereinfachen.
- Die eigentlichen Aufgaben werden dann nur noch in *swap* bzw. dem Kopierkonstruktor geleistet.
- Es gibt dann nur noch einen Zuweisungsoperator, der mit einer Kopie(!) arbeitet und nicht mit Referenzen.
- Das eröffnet mehr Möglichkeiten für den C++-Optimierer. Wenn eine tiefe Kopie wirklich notwendig ist, erfolgt sie bei der Parameterübergabe, danach wird diese nicht ein weiteres Mal kopiert, sondern nur noch verlagert. Wenn keine tiefe Kopie notwendig ist, wird auch keine durchgeführt.

OutOfMemory.cpp

```
#include <iostream>
#include <stdexcept>

using namespace std;

int main() {
   try {
      int count(0);
      for(;;) {
         char* megabyte = new char[1048576];
         count += 1;
         cout << " " << count << flush;
      }
   } catch(bad_alloc) {
      cout << " ... Game over!" << endl;
   }
} // main
```

- Ausnahmenbehandlungen sind eine mächtige (und recht aufwendige!) Kontrollstruktur zur Behandlung von Fehlern.

```
dublin$ ulimit -d 8192  # limits max size of heap (in kb)
dublin$ OutOfMemory
 1 2 3 4 5 6 7 ... Game over!
dublin$
```

- Ausnahmenbehandlungen erlauben das Schreiben robuster Software, die wohldefiniert im Falle von Fehlern reagiert.

Crash.cpp

```
#include <iostream>

using namespace std;

int main() {
   int count(0);
   for(;;) {
      char* megabyte = new char[1048576];
      count += 1;
      cout << " " << count << flush;
   }
} // main
```

- Ausnahmen, die nicht abgefangen werden, führen zum Aufruf von *std*::*terminate*(), das voreinstellungsgemäß *abort()* aufruft.
- Unter UNIX führt *abort()* zu einer Terminierung des Prozesses mitsamt einem Core-Dump.

```
dublin$ ulimit -d 8192
dublin$ Crash
 1 2 3 4 5 6 7Abort(coredump)
dublin$
```

- Dies ist akzeptabel für kleine Programme oder Tests. Viele Anwendungen benötigen jedoch eine robustere Behandlung von Fehlern.

Ausnahmen können als Verletzungen von Verträgen zwischen Klienten und Implementierungen im Falle von Methodenaufrufen betrachtet werden, wo

- ein Klient all die Vorbedingungen zu erfüllen hat und umgekehrt
- die Implementierung die Nachbedingung zu erfüllen hat (falls die Vorbedingung tatsächlich erfüllt gewesen ist).

Es gibt jedoch Fälle, bei denen eine der beiden Seiten den Vertrag nicht halten kann.

Gegeben sei das Beispiel einer Matrixinvertierung:

- Vorbedingung: Die Eingabe-Matrix ist regulär.
- Nachbedingung: Die Ausgabe-Matrix ist die invertierte Matrix der Eingabe-Matrix.

Problem: Wie kann festgestellt werden, dass eine Matrix regulär ist? Dies ist in manchen Fällen fast so aufwendig wie die Invertierung selbst.

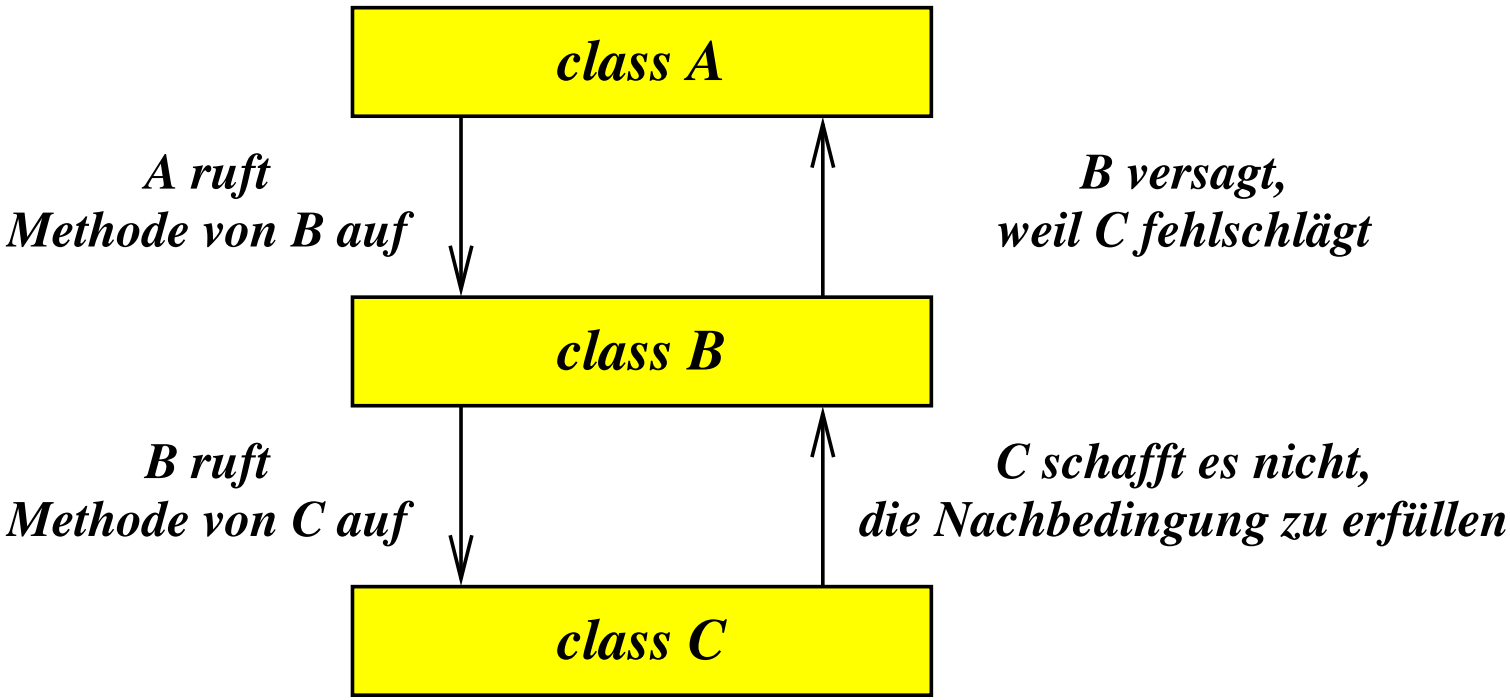
Beispiel: Übersetzer für C++:

- Vorbedingung: Die Eingabedatei ist ein wohldefiniertes Programm in C++.
- Nachbedingung: Die Ausgabedatei enthält eine korrekte Übersetzung des Programms in eine Maschinensprache.

Problem: Wie kann im Voraus sichergestellt werden, dass die Eingabedatei wohldefiniert für C++ ist?

Die Einhaltung der Nachbedingungen kann aus vielerlei Gründen versagt bleiben:

- Laufzeitfehler:
  - Programmierfehler wie z.B. ein Index, der außerhalb des zulässigen Bereiches liegt.
  - Arithmetische Fehler wie Überläufe oder das Teilen durch 0.
- Ausfälle der Systemumgebung wie etwa zu wenig Hauptspeicher, unzureichender Plattenplatz, Hardware-Probleme und unterbrochene Netzwerkverbindungen.


class A
A ruft
Methode von B auf
B versagt,
weil C fehlschlägt
class B
B ruft
Methode von C auf
C schafft es nicht,
die Nachbedingung zu erfüllen
class C

- Eine Software-Komponente ist **robust**, wenn sie nicht nur korrekt ist (d.h. die Nachbedingung wird eingehalten, wenn die Verbedingung erfüllt ist), sondern sie auch Verletzungen der Vorbedingung erkennen und signalisieren kann. Ferner sollte eine robuste Software-Komponente in der Lage sein, alle anderen Probleme zu erkennen und zu signalisieren, die sie daran hindern, die Nachbedingung zu erfüllen.
- Solche Verletzungen oder Nichterfüllungen werden **Ausnahmen** (*exceptions*) genannt.
- Die Signalisierung einer Ausnahme ist so zu verstehen: »Verzeihung, ich muss aufgeben, weil ich dieses Problem nicht selbst weiter lösen kann.«

- Wer ist für die Behandlung einer Ausnahme verantwortlich?
- Welche Informationen sind hierfür weiterzuleiten?
- Welche Optionen stehen einem Ausnahmenbehandler zur Verfügung?

- Es gibt hierfür eine Vielzahl an Konzepten, den zuständigen Ausnahmenbehandler (*exception handler*) zu lokalisieren. Dies hängt jeweils von der Programmiersprache bzw. der verwendeten Bibliothek ab.
- Es wird vielfach gerne gesehen, wenn die Ausnahmenbehandlung vom normalen Programmtext getrennt werden kann, damit der Programmtext nicht mit Überprüfungen nach jedem Methodenaufruf unübersichtlich wird.
- In C++ (und ebenso nicht wenigen anderen Programmiersprachen) liegt die Verantwortung beim Klienten. Wenn kein zuständiger Ausnahmenbehandler definiert ist, dann wird die Ausnahme automatisch durch die Aufrufkette weitergeleitet und dabei der Stack abgebaut. Wenn am Ende nirgends ein Ausnahmenbehandler gefunden wird, terminiert der Prozess mit einem Core-Dump.
- Alternativ gibt es den Ansatz, Ausnahmenbehandler für Objekte zu definieren. Dies ist auch bei C++ möglich, wird aber nicht direkt von der Sprache unterstützt.

Wie können Informationen über eine Ausnahme weitergeleitet werden?

VerboseOutOfMemory.cpp

```
int main() {
   try {
      int count(0);
      for(;;) {
         char* megabyte = new char[1048576];
         count += 1;
         cout << " " << count << flush;
      }
   } catch(bad_alloc& e) {
      cout << " ... Game over!" << endl;
      cout << "This hit me: " << e.what() << endl;
   }
} // main
```

- C++ hat einen recht einfachen und gleichzeitig mächtigen Ansatz: Beliebige Instanzen einer Klasse können verwendet werden, um das Problem zu beschreiben.

Wie können Informationen über eine Ausnahme weitergeleitet werden?


```
dublin$ ulimit -d 8192
dublin$ VerboseOutOfMemory
 1 2 3 4 5 6 7 ... Game over!
This hit me: Out of Memory
dublin$
```

- Alle Ausnahmen, die von der ISO-C++-Standardbibliothek ausgelöst werden, verwenden Erweiterungen der **class** *exception*, die eine virtuelle Methode *what()* anbietet, die eine Zeichenkette für Fehlermeldungen liefert.

exception

```
namespace std {

   class exception {
      public:
         exception() noexcept;
         exception(const exception&) noexcept;
         virtual ~exception() noexcept;
         exception& operator=(const exception&) noexcept;
         virtual const char* what() const noexcept;
   };
}
```

- Hier ist zu beachten, dass Klassen, die für Ausnahmen verwendet werden, einen Kopierkonstruktor anbieten müssen, da dieser implizit bei der Ausnahmenbehandlung verwendet wird.
- Die Signatur einer Funktion oder Methode kann spezifizieren, welche Ausnahmen ausgelöst werden können. **noexcept** bedeutet, dass keinerlei Ausnahmen ausgelöst werden.

Stack.hpp

```
#include "StackExceptions.hpp"
template<class Item>
class Stack {
   public:
      // destructor
      virtual ~Stack() {};
      // accessors
      virtual bool empty() const = 0;
      virtual bool full() const = 0;
      virtual const Item& top() const throw(EmptyStack) = 0;
         // PRE: not empty()
      // mutators
      virtual void push(const Item& item)
            throw(FullStack) = 0;
         // PRE: not full()
      virtual void pop() throw(EmptyStack) = 0;
         // PRE: not empty()
}; // class Stack
```

- Die Menge der potentiell ausgelösten Ausnahmen kann und sollte in eine Signatur aufgenommen werden. Wenn dies erfolgt, dürfen andere Ausnahmen von der Funktion bzw. Methode nicht ausgelöst werden.

Definition von Ausnahmen

StackExceptions.hpp

```
#include <exception>

class StackException : public std::exception {};

class FullStack : public StackException {
   public:
      virtual const char* what() const noexcept {
         return "stack is full";
      };
}; // class FullStack

class EmptyStack : public StackException {
   public:
      virtual const char* what() const noexcept {
         return "stack is empty";
      };
}; // class EmptyStack
```

- Klassen für Ausnahmen sollten hierarchisch organisiert werden.
- Eine **catch**-Anweisung für *StackException* erlaubt das Abfangen der Ausnahmen *FullStack*, *EmptyStack* und aller anderen Erweiterungen von *StackException*.

ArrayedStack.tpp

```
template<class Item>
const Item& ArrayedStack<Item>::top() const throw(EmptyStack) {
   if (index > 0) {
      return items[index-1];
   } else {
      throw EmptyStack();
   }
} // top

template<class Item>
void ArrayedStack<Item>::push(const Item& item) throw(FullStack) {
   if (index < SIZE) {
      items[index] = item;
      index += 1;
   } else {
      throw FullStack();
   }
} // push
```

- **throw** erhält ein Objekt, das die Ausnahme repräsentiert und initiiert die Ausnahmenbehandlung.
- Das Objekt sollte das Problem beschreiben.
- Es ist hierbei erlaubt, temporäre Objekte zu verwenden, da diese bei Bedarf implizit kopiert werden.
- Zu beachten ist, dass alle lokalen Variablen einer Funktion oder Methode, die eine Ausnahme auslöst, jedoch diese nicht abfängt, vollautomatisch im Falle einer Ausnahmenbehandlung dekonstruiert werden.

Calculator.hpp

```
#include <string>
#include "Stack.hpp"
#include "CalculatorExceptions.hpp"

class Calculator {
   public:
      typedef Stack<float> FloatStack;
      // constructor
      Calculator(FloatStack& stack);
      float calculate(const std::string& expr)
         throw(CalculatorException);
         // PRE: expr in RPN syntax
   private:
      FloatStack& opstack;
}; // class Calculator
```

- Diese Klasse bietet einen Rechner an, der Ausdrücke in der umgekehrten polnischen Notation (UPN) akzeptiert.
- Beispiele für gültige Ausdrücke: „1 2 +“, „1 2 3 * +“.
- UPN-Rechner können recht einfach mit Hilfe eines Stacks implementiert werden.

Ausnahmen des Taschenrechners

CalculatorExceptions.hpp

```
#include <exception>
class CalculatorException : public std::exception {};

class SyntaxError : public CalculatorException {
   public:
      virtual const char* what() const noexcept {
         return "syntax error";
      };
}; // class SyntaxError

class BadExpr : public CalculatorException {
   public:
      virtual const char* what() const noexcept {
         return "invalid expression";
      };
}; // class BadExpr

class StackFailure : public CalculatorException {
   public:
      virtual const char* what() const noexcept {
         return "stack failure";
      };
}; // class StackFailure
```

- Die Ausnahmen für *Calculator* sollten entsprechend der Abstraktionsebene dieser Klasse verständlich sein.
- Aus diesem Grunde wird hier die Ausnahme *StackFailure* hinzugefügt, die für den Fall vorgesehen ist, dass der zur Verfügung stehende Stack seine Aufgabe (z.B. wegen mangelnder Kapazität) nicht erfüllt.

Ausnahmenbehandlung des Taschenrechners

Calculator.cpp

```
float Calculator::calculate(const string& expr)
      throw(CalculatorException) {
   istringstream in(expr);
   string token;
   float result;
   try {
      while (in >> token) {
         // ...
      }
      result = opstack.top(); opstack.pop();
      if (!opstack.empty()) {
         throw BadExpr();
      }
   } catch(FullStack) {
      throw StackFailure();
   } catch(EmptyStack) {
      throw BadExpr();
   }
   return result;
} // calculate
```

- Zu beachten ist hier, wie Ausnahmen der *Stack*-Klasse in solche der *Calculator*-Klasse konvertiert werden.

Innenbereich der Schleife des Taschenrechners

Calculator.cpp

```
while (in >> token) {
   if (token == "+" || token == "-" || token == "*" || token == "/") {
      float op2(opstack.top()); opstack.pop();
      float op1(opstack.top()); opstack.pop();
      float result;
      if (token == "+") { result = op1 + op2;
      } else if (token == "-") { result = op1 - op2;
      } else if (token == "*") { result = op1 * op2;
      } else { result = op1 / op2;
      }
      opstack.push(result);
   } else {
      istringstream floatin(token);
      float newop;
      if (floatin >> newop) {
         opstack.push(newop);
      } else {
         throw SyntaxError();
      }
   }
}
result = opstack.top(); opstack.pop();
```

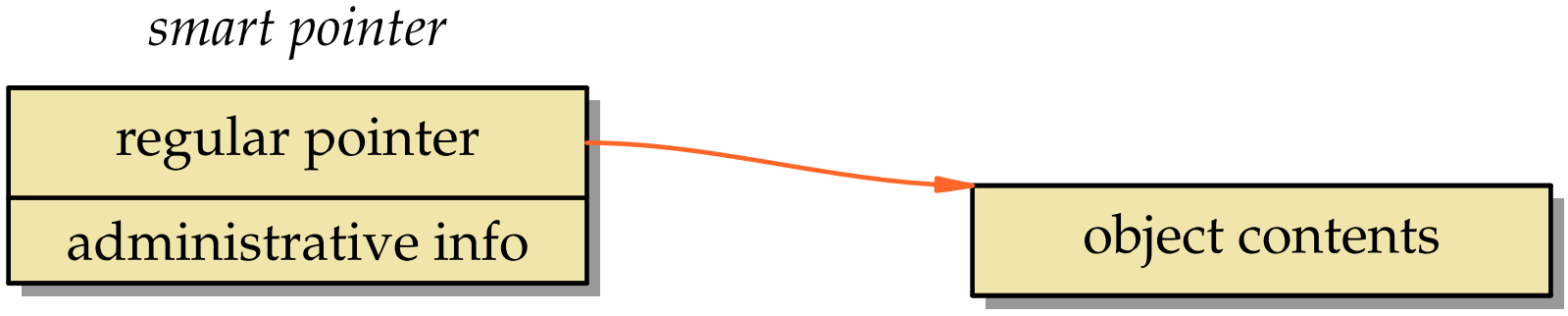
TestCalculator.cpp

```
#include <exception>
#include <iostream>
#include <string>
#include "ArrayedStack.hpp"
#include "Calculator.hpp"
using namespace std;
int main() {
   ArrayedStack<float> stack;
   Calculator calc(stack);
   try {
      string expr;
      while (cout << ": " && getline(cin, expr)) {
         cout << calc.calculate(expr) << endl;
      }
   } catch(exception& exc) {
      cerr << exc.what() << endl;
   }
} // main
```

- Zu beachten ist, dass *expr* automatisch dekonstruiert wird, wenn eine Ausnahme innerhalb des **try**-Blocks ausgelöst wird.
- Hier werden Ausnahmen nur abgefangen und ausgegeben.

```
dublin$ TestCalculator
: 1 2 +
3
: 1 2 3 * +
7
: 1 2 3 4 5 + + + +
stack failure
dublin$ TestCalculator
: 1
1
: 1 2
invalid expression
dublin$ TestCalculator
: +
invalid expression
dublin$ TestCalculator
: x
syntax error
dublin$
```

- Zu beachten ist hier, dass die Implementierung des *ArrayedStack* nur vier Elemente unterstützt.
- „1 2“ ist unzulässig, da der Stack am Ende nach dem Entfernen des obersten Elements nicht leer ist.



- Intelligente Zeiger (*smart pointers*) entsprechend weitgehend normalen Zeigern, haben aber Sonderfunktionalitäten aufgrund weiterer Verwaltungsinformationen.
- Sie werden insbesondere dort eingesetzt, wo die Sprache selbst keine Infrastruktur für die automatisierte Speicherfreigabe anbietet.

- Seit dem C++11-Standard sind intelligente Zeiger Bestandteil der C++-Bibliothek. Zuvor gab es nur den inzwischen abgelösten *auto_ptr* und die Erweiterungen der Boost-Library, die jetzt praktisch übernommen worden sind.
- C++11 bietet folgende Varianten an:

unique_ptr	nur ein Zeiger auf ein Objekt
shared_ptr	mehrere Zeiger auf ein Objekt mit externem Referenzzähler
weak_ptr	nicht das Überleben sichernder „schwacher“ Zeiger auf ein Objekt mit externem Referenzzähler

- Grundsätzlich sollte ein mit **new** erzeugtes Objekt mit **delete** wieder freigegeben werden, sobald der letzte Verweis entfernt wird.
- Unterbleibt dies, haben wir ein Speicherleck.
- Wichtig ist aber auch, dass kein Objekt mehrfach freigegeben wird. Dies kann bei manueller Freigabe leicht geschehen, wenn es mehrere Zeiger auf ein Objekt gibt.
- Intelligente Zeiger können sich auch dann um eine korrekte Freigabe kümmern, wenn eine Ausnahmenbehandlung ausgelöst wird.
- Jedoch können zyklische Datenstrukturen mit der Verwendung von Referenzzählern alleine nicht korrekt aufgelöst werden. Hier sind ggf. Ansätze mit sogenannten „schwachen“ Zeigern denkbar.

- Auf ein Objekt sollten nur Zeiger eines Typs verwendet werden.
- Die einzige Ausnahme davon ist die Mischung von *shared_ptr* und *weak_ptr*.
- Im Normalfall bedeutet dies, dass die entsprechenden Klassen entsprechend angepasst werden müssen, da es dann nicht mehr zulässig ist, **this** zurückzugeben.
- Üblicherweise sollte sogleich bei dem Entwurf einer Klasse geplant werden, welche Art von Zeigern zum Einsatz kommt.

- Wenn es nur einen einzigen Zeiger auf ein Objekt geben soll, dann empfiehlt sich die Verwendung von *unique_ptr*.
- Das ist besonders geeignet für lokale Zeigervariablen oder Zeiger innerhalb einer Klasse.
- Die Freigabe erfolgt dann vollautomatisch, sobald der zugehörige Block bzw. das umgebende Objekt freigegeben werden.
- Bei einer Zuweisung wird der Besitz des Zeigers übertragen. Das funktioniert nur entsprechend mit einem sogenannten *move assignment*, d.h. der Zeigerwert wird von einem anderen *unique_ptr*-Objekt gerettet, der im nächsten Moment ohnehin dekonstruiert wird.
- Andere Zuweisungen dieser Zeiger sind nicht möglich, da dies die Restriktion des exklusiven Zugangs verletzen würde.

ptrex.cpp

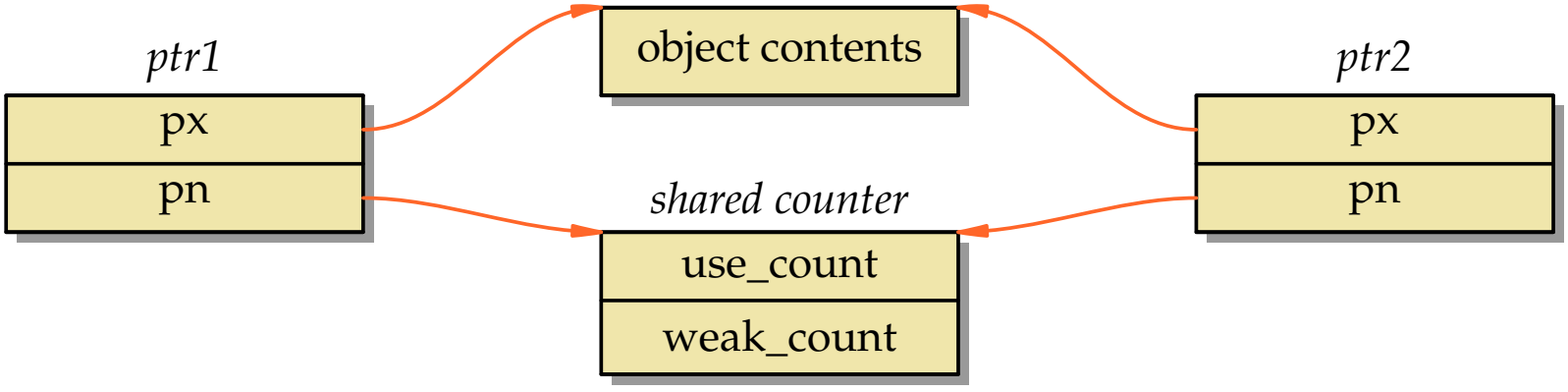
```
void f(int i) {
   Object* ptr = new Object(i);
   if (i == 2) {
      throw something();
   }
   delete ptr;
}
```

- Wenn Objekte in einer Funktion nur lokal erzeugt und verwendet werden, ist darauf zu achten, dass die Freigabe nicht vergessen wird.
- Dies passiert jedoch leicht bei Ausnahmenbehandlungen (möglicherweise durch eine aufgerufene Funktion) oder bei frühzeitigen **return**-Anweisungen.

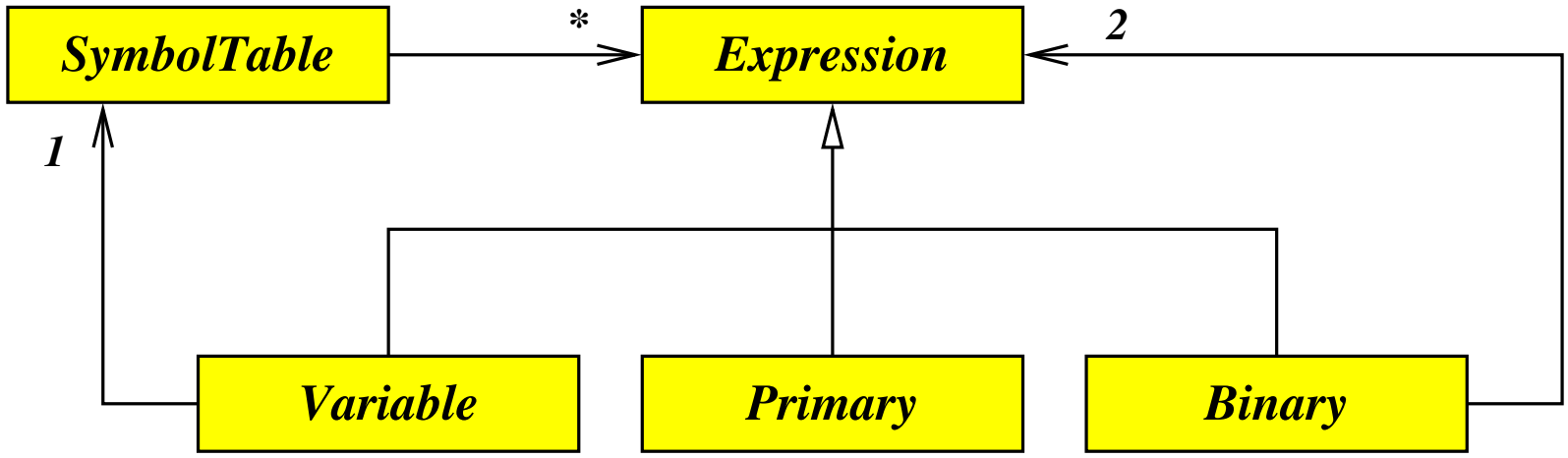
ptrex2.cpp

```
void f(int i) {
   unique_ptr<Object> ptr(new Object(i));
   if (i == 2) {
      throw something();
   }
}
```

- *ptr* kann hier wie ein normaler Zeiger verwendet werden, abgesehen davon, dass eine Zuweisung an einen anderen Zeiger nicht zulässig ist.
- Dann erfolgt die Freigabe des Objekts vollautomatisch über den Dekonstruktor.



- Für den allgemeinen Einsatz empfiehlt sich die Verwendung von *shared_ptr*, das mit Referenzzählern arbeitet.
- Zu jedem referenzierten Objekt gehört ein intern verwaltetes Zählerobjekt, das die Zahl der Verweise zählt. Sobald *use_count* auf 0 sinkt, erfolgt die Freigabe des Objekts.



- Für einen Taschenrechner haben wir eine Datenstruktur für Syntaxbäume (*Expression*) mit den abgeleiteten Klassen *Variable*, *Primary* und *Binary*.
- Einträge in der Symboltabelle können auch auf Syntaxbäume verweisen.

expression.hpp

```
class Expression {
   public:
      virtual ~Expression() {};
      virtual Value evaluate() const = 0;
};

typedef std::shared_ptr<Expression> ExpressionPtr;
```

- Bei Klassenhierarchien, bei denen polymorphe Zeiger eingesetzt werden, ist die Deklaration eines virtuellen Dekonstruktors essentiell.
- Die Methode *evaluate* soll den durch den Baum repräsentierten Ausdruck rekursiv auswerten.
- *ExpressionPtr* wird hier als intelligenter Zeiger auf *Expression* definiert, bei dem beliebig viele Zeiger des gleichen Typs auf ein Objekt verweisen dürfen.

expression.hpp

```
class Binary: public Expression {
   public:
      typedef Value (*BinaryOp)(Value val1, Value val2);
      Binary(BinaryOp _op, ExpressionPtr _expr1, ExpressionPtr _expr2);
      virtual Value evaluate() const;
   private:
      BinaryOp op;
      ExpressionPtr expr1;
      ExpressionPtr expr2;
};
```

- *Binary* repräsentiert einen Knoten des Syntaxbaums mit einem binären Operator und zwei Operanden.
- Statt *Expression*∗ wird dann konsequent *ExpressionPtr* verwendet.

expression.hpp

```
class Variable: public Expression {
   public:
      Variable(SymbolTable& _symtab, const std::string& _varname);
      virtual Value evaluate() const;
      void set(ExpressionPtr expr);
   private:
      SymbolTable& symtab;
      std::string varname;
};
typedef std::shared_ptr<Variable> VariablePtr;
```

- Die Kompatibilität innerhalb der *Expression*-Hierarchie überträgt sich auch auf die zugehörigen intelligenten Zeiger.
- Zwar bilden die intelligenten Zeigertypen keine formale Hierarchie, aber sie bieten Zuweisungs-Operatoren auch für fremde Datentypen an, die nur dann funktionieren, wenn die Kompatibilität für die entsprechenden einfachen Zeigertypen existiert.

parser.cpp

```
ExpressionPtr Parser::parseSimpleExpression() throw(Exception) {
   ExpressionPtr expr = parseTerm();
   while (getToken().symbol == Token::PLUS ||
          getToken().symbol == Token::MINUS) {
      Binary::BinaryOp op;
      switch (getToken().symbol) {
         case Token::PLUS: op = addop; break;
         case Token::MINUS: op = subop; break;
         default: /* does not happen */ break;
      }
      nextToken();
      ExpressionPtr expr2 = parseTerm();
      expr = std::make_shared<Binary>(op, expr, expr2);
   }
   return expr;
}
```

- *make_shared* erzeugt ein Objekt des angegebenen Typs mit **new** und liefert den passenden intelligenten Zeigertyp zurück.
- Das ist in diesem Beispiel *shared_ptr<Binary>*, das entsprechend der Klassenhierarchie an den allgemeinen Zeigertyp *ExpressionPtr* zugewiesen werden kann.

Typentests und Konvertierungen bei intelligenten Zeigern

parser.cpp

```
ExpressionPtr Parser::parseAssignment() throw(Exception) {
   ExpressionPtr expr = parseSimpleExpression();
   if (getToken().symbol == Token::BECOMES) {
      VariablePtr var = std::dynamic_pointer_cast<Variable>(expr);
      if (!var) {
         throw Exception(getToken(), "variable expected");
      }
      nextToken();
      ExpressionPtr expr2 = parseSimpleExpression();
      var->set(expr2);
      return expr2;
   }
   return expr;
}
```

- Statt **dynamic_cast** ist bei intelligenten Zeigern *dynamic_pointer_cast* zu verwenden, um sicherzustellen, dass es bei einem Zählerobjekt bleibt.
- Genauso wie bei **dynamic_cast** wird ein Nullzeiger geliefert, falls der angegebene Zeiger nicht den passenden Typ hat.

- Bei Referenzzyklen bleiben die Referenzzähler positiv, selbst wenn der Zyklus insgesamt nicht mehr von außen erreichbar ist.
- Eine automatisierte Speicherfreigabe (*garbage collection*) würde den Zyklus freigeben, aber mit Zeigern auf Basis von *shared_ptr* gelingt dies nicht.
- Eine Lösung für dieses Problem sind sogenannte schwache Zeiger (*weak pointers*), die bei der Referenzzählung nicht berücksichtigt werden.

Beispiel: Doppelt verkettete Liste

list.hpp

```
template <typename T>
class List {
   private:
      struct Element;
      typedef std::shared_ptr<Element> Link;
      typedef std::weak_ptr<Element> WeakLink;
      struct Element {
         Element(const T& _elem);
         T elem;
         Link next;
         WeakLink prev;
      };
      Link head;
      Link tail;
   public:
      class Iterator {
         // ...
      };
      Iterator begin();
      Iterator end();
      void push_back(const T& object);
};
```

Beispiel: Doppelt verkettete Liste

list.hpp

```
typedef std::shared_ptr<Element> Link;
typedef std::weak_ptr<Element> WeakLink;
struct Element {
   Element(const T& _elem);
   T elem;
   Link next;
   WeakLink prev;
};
```

- Die einzelnen Glieder einer doppelt verketteten Liste verweisen jeweils auf den Nachfolger und den Vorgänger.
- Wenn mindestens zwei Glieder in einer Liste enthalten ist, ergibt dies eine zyklische Datenstruktur.
- Das kann dadurch gelöst werden, dass für die Rückverweise schwache Zeiger verwendet werden.

list.tpp

```
template<typename T>
void List<T>::push_back(const T& object) {
   Link ptr = std::make_shared<Element>(object);
   ptr->prev = tail;
   if (head) {
      tail->next = ptr;
   } else {
      head = ptr;
   }
   tail = ptr;
}
```

- Eine Zuweisung von *shared_ptr* an den korrespondierenden *weak_ptr* ist problemlos möglich wie hier bei: $ptr{->}prev = tail$

Iterator-Klasse mit schwachem Zeiger

list.hpp

```
class Iterator {
   public:
      class Exception: public std::exception {
         public:
            Exception(const std::string& _msg);
            virtual ~Exception() noexcept;
            virtual const char* what() const noexcept;
         private:
            std::string msg;
      };
      bool valid();
      T& operator*();
      Iterator& operator++(); // prefix increment
      Iterator operator++(int); // postfix increment
      Iterator& operator--(); // prefix decrement
      Iterator operator--(int); // postfix decrement
      bool operator==(const Iterator& other);
      bool operator!=(const Iterator& other);
   private:
      friend class List;
      Iterator();
      Iterator(WeakLink _ptr);
      WeakLink ptr;
};
```

list.tpp

```
template<typename T>
T& List<T>::Iterator::operator*() {
   Link p = ptr.lock();
   if (p) {
      return p->elem;
   } else {
      throw Exception("iterator is expired");
   }
}
```

- Ein schwacher Zeiger kann mit Hilfe der *lock*-Methode in einen regulären Zeiger verwandelt werden.
- Wenn das referenzierte Objekt mittlerweile freigegeben wurde, ist der Zeiger 0.

list.tpp

```
template<typename T>
bool List<T>::Iterator::operator==(const Iterator& other) {
   Link p1 = ptr.lock();
   Link p2 = other.ptr.lock();
   return p1 == p2;
}

template<typename T>
bool List<T>::Iterator::operator!=(const Iterator& other) {
   return !(*this == other);
}
```

- Schwache Zeiger können erst dann miteinander verglichen werden, wenn sie zuvor in reguläre Zeiger konvertiert werden.
- Nullzeiger werden hier als äquivalent angesehen.

```
#include <memory>

class Object;
typedef std::shared_ptr<Object> ObjectPtr;
class Object: public std::enable_shared_from_this<Object> {
   public:
      ObjectPtr me() {
         return shared_from_this();
      }
};
```

- Die Grundregel, dass auf ein Objekt nur Zeiger eines Typs verwendet werden sollten, stößt auf ein Problem, wenn statt **this** ein passender intelligenter Zeiger zurückzugeben ist.
- Eine Lösung besteht darin, die Klasse von *std*::*enable_shared_from_this* abzuleiten. Dann steht die Methode *shared_from_this* zur Verfügung. Dies wird implementiert, indem im Objekt zusätzlich ein schwacher Zeiger auf das eigene Objekt verwaltet wird.

```
#include <memory>

class Object;
typedef std::shared_ptr<Object> ObjectPtr;
class Object {
   public:
      class Key {
         friend class Object;
         Key() {}
      };
      static ObjectPtr create() {
         return std::make_shared<Object>(Key());
      }
      Object(Key&& key) {}
};
```

- Um die Grundregel durchzusetzen, erscheint es gelegentlich sinnvoll, die regulären Konstruktoren zu verbergen.
- **private** dürfen Sie jedoch nicht sein, da *std*::*make_shared* einen passenden öffentlichen Konstruktor benötigt.
- Eine Lösung bietet der *pass key*-Ansatz. Der Konstruktor ist zwar öffentlich, aber ohne privaten Schlüssel nicht benutzbar.

- Funktionsobjekte sind Objekte, bei denen der **operator**() definiert ist. (Siehe ISO 14882-2012, Abschnitt 20.8.)
- Viele Algorithmen der STL akzeptieren solche Funktionsobjekte, um aus einer Menge von Objekten (repräsentiert durch Iteratoren) Objekte herauszufiltern oder eine Menge von Objekten zu transformieren.
- Es ist in vielen Fällen nicht notwendig, extra Klassen für Funktionsobjekte zu definieren, da es bereits eine Reihe vorgefertigter Funktionsobjekte gibt und auch Funktionsobjekte entsprechend des λ-Kalküls mit Lambda-Ausdrücken frei konstruiert werden können.

transform.cpp

```
template<typename T>
class SquareIt: public function<T(T)> {
   public:
      T operator()(T x) const noexcept { return x * x; }
};
```

- Die von *function* abgeleitete Klasse *SquareIt* bietet einen das Quadrat seiner Argumente zurückliefernden Funktions-Operator an.
- Die zum ISO-Standard gehörende Template-Klasse *function* dient dazu, die zugehörigen Typen leichter zugänglich zu machen und/oder Funktionen zu verpacken:

```
template<class R, class... ArgTypes>
class function<R(ArgTypes...)> {
   public:
      typedef R result_type;
      typedef T1 argument_type; // defined in case of a unary function
      typedef T1 first_argument_type; // in case of a binary function
      typedef T1 second_argument_type; // in case of a binary function
      // ...
      R operator()(ArgTypes...) const;
}
```

transform.cpp

```
int main() {
   list<int> ints;
   for (int i = 1; i <= 10; ++i) {
      ints.push_back(i);
   }

   list<int> squares;
   transform(ints.begin(), ints.end(),
      back_inserter(squares), SquareIt<int>());

   for (int val: squares) {
      cout << val << endl;
   }
}
```

- *transform* gehört zu den in der STL definierten Operatoren, die auf durch Iteratoren spezifizierten Sequenzen arbeiten.
- Die ersten beiden Parameter von *transform* spezifizieren die zu transformierende Sequenz, der dritte Parameter den Iterator, der die Resultate entgegen nimmt und beim vierten Parameter wird das Funktionsobjekt angegeben, das die gewünschte Abbildung durchführt.

transform.cpp

```
transform(ints.begin(), ints.end(),
   back_inserter(squares), SquareIt<int>());
```

- Wenn die Sequenz-Operatoren der STL einen Iterator für die Ausgabe erhalten, dann gehen sie davon aus, dass hinter dem Ausgabe-Operator bereits Objekte existieren.
- *transform* selbst fügt also keine Objekte irgendwo ein, sondern nimmt Zuweisungen vor.
- Funktionen wie *back_inserter* erzeugen einen speziellen Iterator für einen Container, der neue Objekte einfügt (hier immer an das Ende der Sequenz).
- Bei *transform* wäre auch eine direkte Ersetzung möglich gewesen der ursprünglichen Objekte:

transform.cpp

```
transform(ints.begin(), ints.end(), ints.begin(), SquareIt<int>());
```

- Das Lambda-Kalkül geht auf Alonzo Church und Stephen Kleene zurück, die in den 30er-Jahren damit ein formales System für berechenbare Funktionen entwickelten.
- Zu den wichtigsten Arbeiten aus dieser Zeit gehört der Aufsatz von Alonzo Church: *An Unsolvable Problem of Elementary Number Theory*, *Americal Journal of Mathematics*, Band 58, Nr. 2 (April 1936), S. 345–363.
- Diese Arbeit zeigt, dass es keine berechenbare Funktion gibt, die die Äquivalenz zweier Ausdrücke des Lambda-Kalküls feststellen kann.
- Die Turing-Maschine und das Lambda-Kalkül sind in Bezug auf die Berechenbarkeit äquivalent.
- Das Lambda-Kalkül wurde von funktionalen Programmiersprachen übernommen (etwa von Lisp und Scheme) und wird auch gerne zur formalen Beschreibung der Semantik einer Programmiersprache verwendet (denotationelle Semantik).

Da in C++ Funktionsobjekte wegen der entsprechenden STL-Algorithmen recht beliebt sind, gab es mehrere Ansätze, Lambda-Ausdrücke in C++ einzuführen:

- *boost*::*lambda* von Jaakko Järvi, entwickelt von 1999 bis 2004
- *boost*::*phoenix* von Joel de Guzman und Dan Marsden, entwickelt von 2002 bis 2005
- Integration von Lambda-Ausdrücken im C++-Standard ISO-14882-2012.

Church gibt eine rekursive Definition für Lambda-Ausdrücke, die in eine Grammatik übertragen werden kann:

$$
\begin{aligned}
\langle \text{formula} \rangle &\longrightarrow \langle \text{variable} \rangle \\
&\longrightarrow \text{„}\lambda\text{“ } \langle \text{variable} \rangle \text{ „[“ } \langle \text{formula} \rangle \text{ „]“} \\
&\longrightarrow \text{„\{“ } \langle \text{formula} \rangle \text{ „\}“ „(“ } \langle \text{formula} \rangle \text{ „)“}
\end{aligned}
$$

Bei Variablen werden Namen verwendet wie beispielsweise x oder y.

Später hat sich folgende vereinfachte Grammatik für Lambda-Ausdrücke durchgesetzt:

$$\begin{aligned} \langle \text{formula} \rangle &\longrightarrow \langle \text{variable} \rangle \\ &\longrightarrow \text{„}\lambda\text{“ } \langle \text{variable} \rangle \text{ „.“ } \langle \text{formula} \rangle \\ &\longrightarrow \text{„(“ } \langle \text{formula} \rangle \text{ „)“ } \langle \text{formula} \rangle \end{aligned}$$

Beispiel:

- $\lambda f.\lambda x.(f)(f)x$
  (Traditionelle Schreibweise: $\lambda f\,[\lambda x\,[\{f\}\,(\{f\}\,(x))]]$)

Variablen sind in einem Lambda-Ausdruck entweder frei oder gebunden:

- Bei „λ“ ⟨variable⟩ „[“ ⟨formula⟩ „]“ ist die hinter λ genannte Variable innerhalb der ⟨formula⟩ gebunden.
- Es liegt eine Blockstruktur vor mit entsprechendem lexikalisch bestimmten Sichtbereichen.
- Um die Lesbarkeit zu erhöhen und die textuell definierten Konvertierungen zu vereinfachen, wird normalerweise davon ausgegangen, dass Variablennamen eindeutig sind.
- Variablen, die nicht gebunden sind, sind frei.

Der Textersetzungs-Ausdruck $S_N^x M \mid$ ersetzt x global in M durch N. Hierbei ist x eine Variable.

Beispiele:

- $S_y^x \lambda x.x \mid = \lambda y.y$
- $S_{\lambda x.x}^x \lambda y.(x)(x)\, y \mid = \lambda y.(\lambda x.x)(\lambda x.x)\, y$

Textersetzungen sollten dabei keine Variablenbindungen brechen. Gegebenenfalls sind zuerst Variablennamen zu ersetzen. Das zweite Beispiel war zulässig, weil x nicht gebunden war und die im Ersatztext gebundene Variable x nicht in Konflikt zu bestehenden gebundenen Variablen steht.

- α-**Äquivalenz**: In einem Lambda-Ausdruck dürfen überall Konstrukte der Form $\lambda x.M$ durch $\lambda y.S_y^x M$ | ersetzt werden, vorausgesetzt, dass y innerhalb von M nicht vorkommt.
- Beispiel: $\lambda x.x$ ist α-äquivalent zu $\lambda y.y$
- In $\lambda x.\lambda y.(y)x$ darf y nicht durch x ersetzt werden, da x bereits vorkommt.

- Es dürfen überall Konstrukte der Form $(\lambda x.M)\,N$ durch $S_N^x M \mid$ ersetzt werden, vorausgesetzt, dass die in M gebundenen Variablen sich von den freien Variablen in N unterscheiden.
- Wenn die Voraussetzung nicht erfüllt ist, könnte zuvor bei M oder N eine α-äquivalente Variante gesucht werden, die den Konflikt vermeidet.
- Beispiel:

$$\begin{aligned}(\lambda x.\lambda y.(y)x)y &\rightarrow (\lambda x.\lambda a.(a)x)y \\ &\rightarrow \lambda a.(a)y\end{aligned}$$

- Die β-Reduktion kann mit der Auswertung eines Funktionsaufrufs verglichen werden, bei der der formale Parameter x durch den aktuellen Parameter N ersetzt wird.

- β-Reduktionen (mit ggf. notwendigen α-äquivalenten Ersetzungen) können nacheinander durchgeführt werden, bis sich keine β-Reduktion anwenden lässt.
- Nicht jeder „Funktionsaufruf“ lässt sich dabei auflösen. Beispiel: $(a)b$, wobei a eine ungebundene Variable ist.
- Der Prozess kann halten, muss aber nicht.
- Bei folgenden Beispiel führt die β-Reduktion zum identischen Lambda-Ausdruck, wodurch der Prozess nicht hält:

$$(\lambda x.(x)x)\lambda x.(x)x \quad \rightarrow \quad (\lambda x.(x)x)\lambda x.(x)x$$

Wenn mehrere β-Reduktionen zur Anwendung kommen können, welche ist dann zu nehmen?

- Satz von Church und Rosser (1936): Wenn zwei Prozesse mit dem gleichen Lambda-Ausdruck beginnen und sie beide terminieren, dann haben beide das identische Resultat. Die β-Reduktionen sind somit konfluent.
- Es kann jedoch passieren, dass die Reihenfolge, in der Kandidaten für β-Reduktionen ausgesucht werden, entscheidet, ob der Prozess terminiert oder nicht. Beispiel:

$$(\lambda x.a)(\lambda x.(x)x)\lambda y.(y)y$$

Dieser Ausdruck kann zu a reduziert werden, wenn die am weitesten links stehende Möglichkeit zu einer β-Reduktion gewählt wird.

- Wenn immer die am weitestens links stehende Möglichkeit zu einer β-Reduktion angewendet wird, dann handelt es sich um eine Auswertung in der Normal-Ordnung (*normal-order evaluation* oder auch *lazy evaluation*).
- Wenn der Prozess terminieren kann, dann terminiert auch die Auswertung in der Normal-Ordnung.

- Da der einfache ungetypte Lambda-Kalkül nur Funktionen als Datentypen kennt, werden skalare Werte durch Funktionen repräsentiert. Hierzu haben sich einige Konventionen gebildet.
- Die Boolean-Werte *true* und *false* werden durch Funktionen repräsentiert, die von zwei gegebenen Parametern einen aussuchen:

```
True = Lx.Ly.x
False = Lx.Ly.y
```

- (Die Syntax entspricht der eines kleinen Lambda-Kalkül-Interpreters, bei dem aus Gründen der Einfachheit λ durch `L` repräsentiert wird und Lambda-Ausdrücke über Namen referenziert werden können (hier *True* und *False*).

- Mit den Definitionen für *True* und *False* ergibt sich die Definition einer bedingten Anweisung:

```
If-then-else = La.Lb.Lc.((a)b)c
```

- Der erste Parameter (hier *a*) ist die Bedingung. Wenn sie wahr ist, wird *b* ausgewählt, ansonsten *c*.

```
(((La.Lb.Lc.((a)b)c)Lx.Ly.x)this)that
---> ((Lb.Lc.((Lx.Ly.x)b)c)this)that
---> (Lc.((Lx.Ly.x)this)c)that
---> ((Lx.Ly.x)this)that
---> (Ly.this)that
---> this
```

- Die natürliche Zahl n kann dadurch repräsentiert werden, dass eine beliebige Funktion f n-fach aufgerufen wird. Die Zahl n repräsentiert dann die n-te Potenz einer Funktion. Entsprechend werden natürliche Zahlen als Funktionen definiert, die zwei Parameter erwarten: die anzuwendende Funktion f und der Parameter, der dieser Funktion beim ersten Aufruf zugeführt wird:

```
0 = Lf.Lx.x
1 = Lf.Lx.(f)x
2 = Lf.Lx.(f)(f)x
3 = Lf.Lx.(f)(f)(f)x

> ((3)Lf.(f)hello)Lx.x
((Lf.Lx.(f)(f)(f)x)Lf.(f)hello)Lx.x
---> (Lx.(Lf.(f)hello)(Lf.(f)hello)(Lf.(f)hello)x)Lx.
---> (Lf.(f)hello)(Lf.(f)hello)(Lf.(f)hello)Lx.x
---> ((Lf.(f)hello)(Lf.(f)hello)Lx.x)hello
---> (((Lf.(f)hello)Lx.x)hello)hello
---> (((Lx.x)hello)hello)hello
---> ((hello)hello)hello
```

- Wiederhole x n-mal:

```
Repeat = Ln.Lx.((n)Lg.(g)x)Ly.y
```

- Erhöhe n um 1:

```
Succ = Ln.Lf.Lx.(f)((n)f)x
```

(Es ist zu beachten, dass 3 und (Succ)2 nicht identisch aussehen, aber in der Funktionalität des Wiederholens äquivalent sind.)

- Verkleinere n um 1:

```
Pred = Ln.(((n)Lp.Lz.((z)(Succ)(p)True)
   (p)True)Lz.((z)0)0)False
```

(Das funktioniert nicht für negative Zahlen.)

- Arithmetische Operationen:

```
+ = Lm.Ln.Lf.Lx.((m)f)((n)f)x
* = Lm.Ln.Lf.(m)(n)f
```

- Test, ob eine n 0 ist:

```
Zero? = Ln.((n)(True)False)True
```

- Ein naiver Versuch, eine rekursive Funktion zur Berechnung der Fakultät von n könnte so aussehen:

```
F = Ln.(((If-then-else)(Zero?)n)1)((*)n)(F)(Pred)n
```

- Das ist jedoch nicht zulässig, da dies nicht textuell expandiert werden kann.
- Glücklicherweise lässt sich das Problem mit dem sogenannten Fixpunkt-Operator Y lösen:

```
Y = Ly. (Lx.(y)(x)x) Lx.(y)(x)x
```

- Es gilt $(Y)f \rightarrow (f)(Y)f$.
- Es ist dabei zu beachten, dass der Y-Operator nur in Verbindung mit einer Auswertung in der Normal-Ordnung funktioniert.
- Mit dem Y-Operator lässt sich nun F definieren:

```
F = (Y)Lf.Ln.(((If-then-else)(Zero?)n)1)((*)n)(f)(Pred)n
```

- ```
  > ((Repeat)(F)3)hi
  [..]
  ---> (((((hi)hi)hi)hi)hi)hi
  1114 reductions performed.
  ```
  ```

Es gibt zwei wesentliche Punkte, weswegen Lambda-Ausdrücke auch in nicht-funktionalen Programmiersprachen (wie etwa C++) interessant sind:

- Anonyme Funktionen können lokal konstruiert und übergeben werden. Ein Beispiel dafür wäre das Sortierkriterium bei *sort*. Die lokal definierte anonyme Funktion kann dabei auch die Variablen der sie umgebenden Funktion sehen (*closure*).
- Funktionen können aus anderen Funktionen abgeleitet werden. Beispielsweise kann eine Funktion mit zwei Argumenten in eine Funktion abgebildet werden, bei der der eine Parameter fest vorgegeben und nur noch der andere variabel ist (*currying*).

Grundsätzlich kann das alles auch konventionell formuliert werden durch explizite Klassendefinitionen. Aber dann wird der Code umfangreicher, umständlicher (etwa durch die explizite Übergabe der lokal sichtbaren Variablen) und schwerer lesbarer (zusammenhängender Code wird auseinandergerissen). Allerdings können Lambda-Ausdrücke auch zur Unlesbarkeit beitragen.

transform2.cpp

```
int main() {
   list<int> ints;
   for (int i = 1; i <= 10; ++i) {
      ints.push_back(i);
   }

   list<int> squares;
   transform(ints.begin(), ints.end(),
      back_inserter(squares), [](int val) { return val*val; });

   for (int val: squares) {
      cout << val << endl;
   }
}
```

- Mit Lambda-Ausdrücken werden implizit unbenannte Klassen erzeugt und temporäre Objekte instantiiert.
- In diesem Beispiel ist [](**int** *val*){ **return** *val*∗*val*; } der Lambda-Ausdruck, der ein temporäres unäres Funktionsobjekt erzeugt, das sein Argument quadriert.

⟨lambda-expression⟩ ⟶ ⟨lambda-introducer⟩ [⟨lambda-declarator⟩]
⟨compound-statement⟩

⟨lambda-introducer⟩ ⟶ „[“ [⟨lambda-capture⟩] „]“

⟨lambda-capture⟩ ⟶ ⟨capture-default⟩

⟶ ⟨capture-list⟩

⟶ ⟨capture-default⟩ „,“ ⟨capture-list⟩

⟨capture-default⟩ ⟶ „&“ | „=“

⟨capture-list⟩ ⟶ ⟨capture⟩ [„...“]

⟶ ⟨capture-list⟩ „,“ ⟨capture⟩ [„...“]

⟨capture⟩ ⟶ ⟨identifier⟩

⟶ „&“ ⟨identifier⟩

⟶ **this**

⟨lambda-declarator⟩ ⟶ „(“ ⟨parameter-declaration-clause⟩ „)“
[**mutable**] [⟨exception-specification⟩]
[⟨attribute-specifier-seq⟩]
[⟨trailing-return-type⟩]

- Lambda-Ausdrücke, die in eine Funktion eingebettet sind, „sehen“ die lokalen Variablen aus den umgebenden Blöcken.
- In vielen funktionalen Programmiersprachen überleben die lokalen Variablen selbst dann, wenn der sie umgebende Block verlassen wird, weil es noch überlebende Funktionsobjekte gibt, die darauf verweisen. Dies benötigt zur Implementierung sogenannte *cactus stacks*.
- Da für C++ der Aufwand für diese Implementierung zu hoch ist und auch die Übersetzung normalen Programmtexts ohne Lamda-Ausdrücke verteuern würde, fiel die Entscheidung, einen alternativen Mechanismus zu entwickeln, der über *lambda-capture* spezifiziert wird.

```
template<typename T>
function<T(T)> create_multiplier(T factor) {
   return function<T(T)>([=](T val) { return factor*val; });
}

int main() {
   auto multiplier = create_multiplier(7);
   for (int i = 1; i < 10; ++i) {
      cout << multiplier(i) << endl;
   }
}
```

- *create_multiplier* ist eine Template-Funktion, die ein mit einem vorgegebenen Faktor multiplizierendes Funktionsobjekt erzeugt und zurückliefert.
- Die Standard-Template-Klasse *function* wird hier genutzt, um das Funktionsobjekt in einen bekannten Typ zu verpacken.
- Die *lambda-capture* [=] legt fest, dass die aus der Umgebung referenzierten Variablen beim Erzeugen des Funktionsobjekts kopiert werden.

Erzeugung einer unbenannten Klasse


```
template<typename T>
class Anonymous {
   public:
      Anonymous(const T& factor_) : factor(factor_) {}
      T operator()(T val) const {
         return factor*val;
      }
   private:
      T factor;
};

template<typename T>
function<T(T)> create_multiplier(T factor) {
   return function<T(T)>(Anonymous<T>(factor));
}
```

- Der Lambda-Ausdruck führt implizit zu einer Erzeugung einer unbenannten Klasse (hier einfach *Anonymous* genannt).
- Jeder aus der Umgebung referenzierte Variable, die kopiert wird, findet sich als gleichnamige Variable der Klasse wieder, die bei der Konstruktion übergeben wird.

```
template<typename T>
tuple<function<T()>, function<T()>, function<T()>>
create_counter(T val) {
   shared_ptr<T> p(new T(val));
   auto incr = [=]() { return ++*p; };
   auto decr = [=]() { return --*p; };
   auto getval = [=]() { return *p; };
   return make_tuple(function<T()>(incr),
      function<T()>(decr), function<T()>(getval));
}
```

- In funktionsorientierten Sprachen werden gerne die gemeinsamen Variablen aus der Hülle benutzt, um private Variablen für eine Reihe von Funktionsobjekten zu haben, die wie objekt-orientierte Methoden arbeiten.
- Das ist auch in C++ möglich mit Hilfe von *shared_ptr*.
- Aber normalerweise ist es einfacher, eine entsprechende Klasse zu schreiben.

```
int main() {
   function<int()> incr, decr, getval;
   tie(incr, decr, getval) = create_counter(0);
   char ch;
   while (cin >> ch) {
      switch (ch) {
         case '+': incr(); break;
         case '-': decr(); break;
         default: break;
      }
   }
   cout << getval() << endl;
}
```

- *create_counter* erzeugt ein Tupel (Datenstruktur aus **#include** <tuple>) und *tie* erlaubt es, gleich mehrere Variablen aus einem Tupel zuzuweisen.
- Danach bleibt die gemeinsame private Variable solange bestehen, bis diese von *shared_ptr* freigegeben wird, d.h. sobald die letzte Referenz darauf verschwindet.

```
vector<int> values(10);
int count = 0;
generate(values.begin(), values.end(), [&]() { return ++count; });
```

- Alternativ können Variablen nicht kopiert, sondern per impliziter Referenz benutzt werden.
- Dann darf das Funktionsobjekt aber nicht länger leben bzw. benutzt werden, als die entsprechenden Variablen noch leben. Das liegt in der Verantwortung des Programmierers.
- *generate* steht über **#include** <algorithm> zur Verfügung und weist die von dem Funktionsobjekt erzeugten Werte sukzessiv allen referenzierten Werten zwischen dem ersten Iterator (inklusive) und dem zweiten Iterator (exklusive) zu.

C++ bietet eine Vielfalt weiterer Techniken für Templates:

- Templates können auch außerhalb von Klassen für einzelne Funktionen verwendet werden.
- Templates können implizit in Zuge von Überladungen instantiiert werden. Dabei können sie auch in Konkurrenz zu Nicht-Template-Funktionen des gleichen Namens stehen.
- Templates können innerhalb von Klassen definiert werden.
- Templates können neben Typen auch ganze Zahlen, Zeiger, Referenzen oder andere Templates als Parameter haben. Parameter können optional sein.
- Unterstützung von Spezialfällen und rekursive Templates. (Damit erreichen wir die Mächtigkeit einer Turing-Maschine zur Übersetzzeit!)
- Literatur: David Vandevoorde und Nicolai M. Josuttis: *C++ Templates*

```
template<typename T>
inline void exchange(T& v1, T& v2) {
   T tmp(v1); v1 = v2; v2 = tmp;
}
```

- *exchange* ist hier eine generelle Funktion zum Austausch zweier Variableninhalte.
- (Eine entsprechende Funktion namens *swap* existiert in der Standardbibliothek.)
- Die Funktion kann *ohne* Template-Parameter verwendet werden. In diesem Falle sucht der Übersetzer zunächst nach einer entsprechenden Nicht-Template-Funktion und, falls sich kein entsprechender Kandidat findet, nach einem Template, das sich passend instantiieren lässt.

```
  int i, j;
  // ...
  exchange(i, j);
```

```
template<typename R, typename T>
R eval(R (*f)(T), T x) {
   static map<T, R> cache;
   // attempt to insert default value
   typename map<T, R>::iterator it; bool inserted;
   tie(it, inserted) = cache.insert(make_pair(x, R()));
   // if this was successful, it wasn't computed before
   if (inserted) {
      it->second = f(x); // replace default value by actual value
   }
   return it->second;
}
```

- Die Typen eines Parameters eines Funktions-Templates können von den Template-Typenparametern in beliebiger Weise abgeleitet werden.
- Hier erwartet *eval* zwei Parameter: Eine (möglicherweise nur aufwendig auszuwertende) Funktion f und ein Argument x. In der Variablen *cache* werden bereits ausgerechnete Werte $f(x)$ notiert, um wiederholte Berechnungen zu vermeiden.
- Bei zusammengesetzten Typnamen ist innerhalb von Templates das Schlüsselwort **typename** wichtig, damit eine syntaktische Analyse auch von noch nicht instantiierten Templates möglich ist.

```
int fibonacci(int i) {
   if (i <= 2) return 1;
   return eval(fibonacci, i-1) + eval(fibonacci, i-2);
}

//
cout << eval(fibonacci, 10) << endl;
```

- Hier wird F_i für jedes i nur ein einziges Mal berechnet.
- Die Template-Parameter R und T werden hier ebenfalls vollautomatisch abgeleitet.

```
template<int N, typename T>
T sum(T a[N]) {
   T result = T();
   for (int i = 0; i < N; ++i) {
      result += a[i];
   }
   return result;
}
```

- Ganzzahlige Template-Parameter sind zulässig einschließlich Aufzählungstypen (**enum**).
- Diese können beispielsweise bei Vektoren eingesetzt werden.
- Wenn der Typ unbekannt ist, aber eine explizite Initialisierung gewünscht wird, kann dies durch die explizite Verwendung des Default-Constructors geschehen. Dieser liefert hier auch korrekt auf 0 initialisierte Werte für elementare Datentypen wie **int** oder **float**.

```
   int a[] = {1, 2, 3, 4};
   cout << sum<4>(a) << endl;
```

```
template<typename CONTAINER>
bool is_palindrome(const CONTAINER& cont) {
   if (cont.empty()) return true;
   auto forward(cont.begin());
   auto backward(cont.end());
   --backward;
   for(;;) {
      if (forward == backward) return true;
      if (*forward != *backward) return false;
      ++forward;
      if (forward == backward) return true;
      --backward;
   }
}
```

- Da die Zugriffs-Operatoren für Iteratoren für alle Container einheitlich sind, ist es diesem Template egal, ob es sich um eine *list*, eine *deque*, einen *string* oder was auch immer handelt, sofern alle verwendeten Operatoren unterstützt werden.

- Häufig ist bei Templates für Container und/oder Iteratoren die Deklaration abgeleiteter Typen notwendig wie etwa bei dem Elementtyp des Containers. Hier ist es hilfreich, dass sowohl Container als auch Iteratoren folgende namenstechnisch hierarchisch untergeordnete Typen anbieten:

value_type	Zugehöriger Elemente-Typ
reference	Referenz-Typ zu *value_type*
difference_type	Datentyp für die Differenz zweiter Iteratoren; sinnvoll etwa bei *vector*, *string* oder *deque*
size_type	Passender Typ für die Zahl der Elemente

- Wenn der übergeordnete Typ ein Template-Parameter ist, dann muss jeweils **typename** vorangestellt werden.

```
template<typename ITERATOR>
inline auto
sum(ITERATOR from, ITERATOR to) -> decltype(*from + *from) {
   typedef decltype(*from + *from) Value;
   Value s = Value();
   while (from != to) {
      s += *from++;
   }
   return s;
}
```

- Die Template-Funktion *sum* erwartet zwei Iteratoren und liefert die Summe aller Elemente, die durch diese beiden Iteratoren eingegrenzt werden (inklusiv bei dem ersten Operator, exklusiv bei dem zweiten).
- **decltype** kam durch C++11 hinzu und erlaubt es, einen Datentyp für eine Deklaration aus einem Ausdruck abzuleiten.
- Da dies von den Parametern abhängt, wurde der Rückgabetyp hinter die Parameter verschoben und zum Ausgleich zu Beginn **auto** angegeben.

```
#include <iterator>
#include <iostream>
// ...
int main() {
   typedef int ELEMENT;
   cout << "sum = " <<
      sum(istream_iterator<ELEMENT>(cin), istream_iterator<ELEMENT>())
         << endl;
}
```

- Praktischerweise gibt es Stream-Iteratoren, die wie die bekannten Iteratoren arbeiten, jedoch die gewünschten Elemente jeweils auslesen bzw. herausschreiben.
- *istream_iterator* ist ein Iterator für einen beliebigen *istream*. Der Default-Konstruktor liefert hier einen Endezeiger.

- Polymorphismus bedeutet, dass die jeweilige Methode bzw. Funktion in Abhängigkeit der Parametertypen (u.a. auch nur von einem einzigen Parametertyp) ausgewählt wird.
- Dies kann statisch (also zur Übersetzzeit) oder dynamisch (zur Laufzeit) erfolgen.
- Ferner lässt sich unterscheiden, ob die Typen irgendwelchen Beschränkungen unterliegen oder nicht.
- Dies lässt sich prinzipiell frei kombinieren. C++ unterstützt davon jedoch nur zwei Varianten:

	statisch	dynamisch
beschränkt	(z.B. in Ada)	virtuelle Methoden in C++
unbeschränkt	Templates in C++	(z.B. in Smalltalk oder Perl)

Statischer vs. dynamischer Polymorphismus in C++360

Vorteile dynamischen Polymorphismus in C++:

- Unterstützung heterogener Datenstrukturen, etwa einer Liste von Widgets oder graphischer Objekte.
- Die Schnittstelle ist durch die Basisklasse klarer definiert, da sie dadurch beschränkt ist.
- Der generierte Code ist kompakter.

Vorteile statischen Polymorphismus in C++:

- Erhöhte Typsicherheit.
- Die fehlende Beschränkung auf eine Basisklasse erweitert den potentiellen Anwendungsbereich. Insbesondere können auch elementare Datentypen mit unterstützt werden.
- Der generierte Code ist effizienter.

```
class StdRand {
   public:
      void seed(long seedval) { srand(seedval); }
      long next() { return rand(); }
};

class Rand48 {
   public:
      void seed(long seedval) { srand48(seedval); }
      long next() { return lrand48(); }
};
```

- Gegeben seien zwei Klassen, die nicht miteinander verwandt sind, aber bei einigen relevanten Methoden die gleichen Signaturen offerieren wie hier etwa bei *seed* und *next*.

```
template<typename Rand>
int test_sequence(Rand& rg) {
   const int N = 64;
   int hits[N][N][N] = {{{0}}};
   rg.seed(0);
   int r1 = rg.next() / N % N;
   int r2 = rg.next() / N % N;
   int max = 0;
   for (int i = 0; i < N*N*N*N; ++i) {
      int r3 = rg.next() / N % N;
      int count = ++hits[r1][r2][r3];
      if (count > max) {
         max = count;
      }
      r1 = r2; r2 = r3;
   }
   return max;
}
```

- Dann können beide von der gleichen Template-Funktion behandelt werden.

```
int main() {
   StdRand stdrand;
   Rand48 rand48;
   cout << "result of StdRand: " << test_sequence(stdrand) << endl;
   cout << "result of Rand48: " << test_sequence(rand48) << endl;
}
```

- Hier verwendet *test_sequence* jeweils die passenden Methoden *seed* und *next* in Abhängigkeit des statischen Argumenttyps.
- Die Kosten für den Aufruf virtueller Methoden entfallen hier. Dafür wird hier der Programmtext für *test_sequence* für jede Typen-Variante zusätzlich generiert.

```
template<typename T>
T mod(T a, T b) {
   return a % b;
}

double mod(double a, double b) {
   return fmod(a, b);
}
```

- Explizite Spezialfälle können in Konkurrenz zu implizit instantiierbaren Templates stehen. Sie werden dann, falls sie irgendwo passen, bevorzugt verwendet.
- Auf diese Weise ist es auch möglich, effizientere Algorithmen für Spezialfälle neben dem allgemeinen Template-Algorithmus zusätzlich anzubieten.

```
template<typename T>
const char* tell_type(T* p) { return "is a pointer"; }

template<typename T>
const char* tell_type(T (*f)()) { return "is a function"; }

template<typename T>
const char* tell_type(T v) { return "is something else"; }

int main() {
   int* p; int a[10]; int i;
   cout << "p " << tell_type(p) << endl;
   cout << "a " << tell_type(a) << endl;
   cout << "i " << tell_type(i) << endl;
   cout << "main " << tell_type(main) << endl;
}
```

- Speziell konstruierte Typen können separat behandelt werden, so dass sich etwa Zeiger von anderen Typen unterscheiden lassen.

```
thales$ g++ -c -fpermissive -DLAST=30 Primes.cpp 2>&1 | fgrep 'In instantiation'
Primes.cpp: In instantiation of 'void Prime_print<i>::f() [with int i = 29]':
Primes.cpp: In instantiation of 'void Prime_print<i>::f() [with int i = 23]':
Primes.cpp: In instantiation of 'void Prime_print<i>::f() [with int i = 19]':
Primes.cpp: In instantiation of 'void Prime_print<i>::f() [with int i = 17]':
Primes.cpp: In instantiation of 'void Prime_print<i>::f() [with int i = 13]':
Primes.cpp: In instantiation of 'void Prime_print<i>::f() [with int i = 11]':
Primes.cpp: In instantiation of 'void Prime_print<i>::f() [with int i = 7]':
Primes.cpp: In instantiation of 'void Prime_print<i>::f() [with int i = 5]':
Primes.cpp: In instantiation of 'void Prime_print<i>::f() [with int i = 3]':
Primes.cpp: In instantiation of 'void Prime_print<i>::f() [with int i = 2]':
thales$
```

- Auf einer Sitzung des ISO-Standardisierungskommitees im Jahr 1994 demonstrierte Erwin Unruh die Möglichkeit, Templates zur Programmierung zur Übersetzungszeit auszunutzen.
- Sein Beispiel berechnete die Primzahlen. Die Ausgabe erfolgte dabei über die Fehlermeldungen des Übersetzers.
- Siehe http://www.erwin-unruh.de/Prim.html

```
template<int N>
class Fibonacci {
   public:
      static constexpr int
         result = Fibonacci<N-1>::result +
                  Fibonacci<N-2>::result;
};

template<>
class Fibonacci<1> {
   public: static constexpr int result = 1;
};

template<>
class Fibonacci<2> {
   public: static constexpr int result = 1;
};
```

- Templates können sich selbst rekursiv verwenden. Die Rekursion lässt sich dann durch die Spezifikation von Spezialfällen begrenzen.

```
template<int N>
class Fibonacci {
   public:
      static constexpr int
         result = Fibonacci<N-1>::result +
                  Fibonacci<N-2>::result;
};
```

- Dabei ist es sinnvoll, mit **constexpr**-Konstantenberechnungen zu arbeiten, weil diese zwingend zur Übersetzzeit erfolgen.
- Da **constexpr** erst mit C++11 eingeführt wurde, wurde früher auf **enum** zurückgegriffen.

```
int a[Fibonacci<6>::result];
int main() {
   cout << sizeof(a)/sizeof(a[0]) << endl;
}
```

- Zur Übersetzzeit berechnete Werte können dann auch selbstverständlich zur Dimensionierung globaler Vektoren verwendet werden.

```
thales$ make
gcc-makedepend -std=gnu++11 Fibonacci.cpp
g++ -Wall -g -std=gnu++11  -c -o Fibonacci.o Fibonacci.cpp
g++ -o Fibonacci  Fibonacci.o
thales$ Fibonacci
8
thales$
```

Vermeidung von Schleifen mit rekursiven Templates 370

```
template <int N, typename T>
class Sum {
   public:
      static inline T result(T* a) {
         return *a + Sum<N-1, T>::result(a+1);
      }
};

template <typename T>
class Sum<1, T> {
   public:
      static inline T result(T* a) {
         return *a;
      }
};
```

- Rekursive Templates können verwendet werden, um **for**-Schleifen mit einer zur Übersetzzeit bekannten Zahl von Iterationen zu ersetzen.

```
template <typename T>
inline auto sum(T& a) -> decltype(a[0] + a[0]) {
   return Sum<extent<T>::value,
      typename remove_extent<T>::type>::result(a);
}

int main() {
   int a[] = {1, 2, 3, 4, 5};
   cout << sum(a) << endl;
}
```

- Die Template-Funktion *sum* vereinfacht hier die Nutzung.
- Da der Parameter per Referenz übergeben wird, bleibt hier die Typinformation einschließlich der Dimensionierung erhalten.
- *extent*$<T>$::*value* liefert die Dimensionierung, *remove_extent*$<T>$::*type* den Element-Typ des Arrays.

- Traits sind Charakteristiken, die mit Typen assoziiert werden.
- Die Charakteristiken selbst können durch Klassen repräsentiert werden und die Assoziationen können implizit mit Hilfe von Templates oder explizit mit Template-Parametern erfolgen.

Sum.hpp

```
#ifndef SUM_H
#define SUM_H

template <typename T>
inline T sum(const T* begin, const T* end) {
   T result = T();
   for (const T* it = begin; it < end; ++it) {
      result += *it;
   }
   return result;
}

#endif
```

- Die Template-Funktion *sum* erhält einen Zeiger auf den Anfang und das Ende eines Arrays und liefert die Summe aller enthaltenen Elemente.
- (Dies ließe sich auch mit Iteratoren lösen, darauf wird hier jedoch der Einfachheit halber verzichtet.)

Summe eines Arrays

TestSum.cpp

```
#include "Sum.hpp"
#include <iostream>
#define DIM(vec) (sizeof(vec)/sizeof(vec[0]))
using namespace std;
int main() {
   int numbers[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   cout << "sum of numbers[] = " <<
      sum(numbers, numbers + DIM(numbers)) << endl;
   float floats[] = {1.2, 3.7, 4.8};
   cout << "sum of floats[] = " <<
      sum(floats, floats + DIM(floats)) << endl;
   char text[] = "Hallo zusammen, dies ist etwas Text!!";
   cout << "sum of text[] = " << sum(text, text + DIM(text)) << endl;
}
```

- Bei den ersten beiden Arrays funktioniert das Template recht gut. Weswegen scheitert es im dritten Fall?

```
thales$ testsum
sum of numbers[] = 55
sum of floats[] = 9.7
sum of text[] = ,
thales$
```

SumTraits.hpp

```
#ifndef SUM_TRAITS_H
#define SUM_TRAITS_H

// by default, we use the very same type
template <typename T>
class SumTraits {
   public:
      typedef T SumValue;
};

// special case for char
template <>
class SumTraits<char> {
   public:
      typedef int SumValue;
};

#endif
```

- Die Template-Klasse *SumTraits* liefert als Charakteristik den jeweils geeigneten Datentyp für eine Summe von Werten des Typs T.
- Per Voreinstellung ist das T selbst, aber es können Ausnahmen definiert werden wie hier zum Beispiel für *char*.

TestSum2.cpp

```
#include "Sum2.hpp"
#include <iostream>

using namespace std;

#define DIM(vec) (sizeof(vec)/sizeof(vec[0]))

int main() {
   int numbers[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   cout << "sum of numbers[] = " <<
      sum(numbers, numbers + DIM(numbers)) << endl;
   float floats[] = {1.2, 3.7, 4.8};
   cout << "sum of floats[] = " <<
      sum(floats, floats + DIM(floats)) << endl;
   char text[] = "Hallo zusammen, dies ist etwas Text!!";
   cout << "sum of text[] = " << sum(text, text + DIM(text)) << endl;
}
```

```
thales$ testsum2
sum of numbers[] = 55
sum of floats[] = 9.7
sum of text[] = 3372
thales$
```

Sum3.hpp

```
#ifndef SUM3_H
#define SUM3_H

#include "SumTraits.hpp"

template <typename T, typename ST = SumTraits<T>>
class Sum {
   public:
      typedef typename ST::SumValue SumValue;
      static SumValue sum(const T* begin, const T* end) {
         SumValue result = SumValue();
         for (const T* it = begin; it < end; ++it) {
            result += *it;
         }
         return result;
      }
};

template <typename T>
inline typename SumTraits<T>::SumValue sum(const T* begin, const T* end) {
   return Sum<T>::sum(begin, end);
}

template <typename ST, typename T>
inline typename ST::SumValue sum(const T* begin, const T* end) {
   return Sum<T, ST>::sum(begin, end);
}

#endif
```

Sum3.hpp

```
template <typename T, typename ST = SumTraits<T>>
class Sum {
   public:
      typedef typename ST::SumValue SumValue;
      static SumValue sum(const T* begin, const T* end) {
         SumValue result = SumValue();
         for (const T* it = begin; it < end; ++it) {
            result += *it;
         }
         return result;
      }
};
```

- C++ unterstützt voreingestellte Template-Parameter.
- Leider nur bei Template-Klassen und nicht bei Template-Funktionen. Deswegen muss die Template-Funktion hier in eine Klasse mit einer statischen Funktion verwandelt werden.
- Diese Konstruktion ermöglicht dann einem Nutzer dieser Konstruktion die Voreinstellung zu übernehmen oder bei Bedarf eine eigene Traits-Klasse zu spezifizieren.

Sum3.hpp

```
template <typename T>
inline typename SumTraits<T>::SumValue sum(const T* begin,
       const T* end) {
   return Sum<T>::sum(begin, end);
}

template <typename ST, typename T>
inline typename ST::SumValue sum(const T* begin, const T* end) {
   return Sum<T, ST>::sum(begin, end);
}
```

- Mit diesen beiden Template-Funktionen wird wieder die zuvor gewohnte Bequemlichkeit hergestellt.
- Die erste Variante entspricht der zuvor gewohnten Funktionalität (implizite Wahl der Charakteristik).
- Die zweite Variante erlaubt die Spezifikation der Traits-Klasse. Dies erfolgt praktischerweise über den ersten Template-Parameter, damit der zweite implizit über den Aufruf bestimmt werden kann.

TestSum3.cpp

```
#include "Sum3.hpp"
#include <iostream>
#define DIM(vec) (sizeof(vec)/sizeof(vec[0]))

using namespace std;

class MyTraits {
   public:
      typedef double SumValue;
};

int main() {
   int numbers[] = {2147483647, 10};
   cout << "sum of numbers[] = " <<
      sum(numbers, numbers + DIM(numbers)) << endl;
   cout << "sum of numbers[] = " <<
      sum<MyTraits>(numbers, numbers + DIM(numbers)) << endl;
}
```

```
thales$ testsum3
sum of numbers[] = -2147483639
sum of numbers[] = 2.14748e+09
thales$
```

```
template <typename Derived>
class Base {
   // ...
};

class Derived: public Base<Derived> {
   // ...
};
```

- Es ist möglich, eine Template-Klasse mit einer von ihr abgeleiteten Klasse zu parametrisieren.
- Der Begriff geht auf James Coplien zurück, der diese Technik immer wieder beobachtete.
- Diese Technik nützt aus, dass die Methoden der Basisklasse erst instantiiert werden, wenn der Template-Parameter (d.h. die davon abgeleitete Klasse) dem Übersetzer bereits bekannt sind. Entsprechend kann die Basisklasse von der abgeleiteten Klasse abhängen.

```
// nach Michael Lehn
template <typename Implementation>
class Base {
   public:
      Implementation& impl() {
         return static_cast<Implementation&>(*this);
      }
      void aMethod() {
         impl().aMethod();
      }
};

class Implementation: public Base<Implementation> {
   public:
      void aMethod() {
         // ...
      }
};
```

- Das CRTP ermöglicht hier die saubere Trennung zwischen einem herausfaktorisierten Teil in der Basisklasse von einer Implementierung in der abgeleiteten Klasse, wobei keine Kosten für virtuelle Methodenaufrufe zu zahlen sind.

```
Implementation& impl() {
   return static_cast<Implementation&>(*this);
}
```

- Mit **static_cast** können Typkonvertierungen ohne Überprüfungen zur Laufzeit vorgenommen werden. Insbesondere ist eine Konvertierung von einem Zeiger oder einer Referenz auf einen Basistyp zu einem passenden abgeleiteten Datentyp möglich. Der Übersetzer kann dabei aber nicht sicherstellen, dass das referenzierte Objekt den passenden Typ hat. Falls nicht, ist der Effekt undefiniert.
- In diesem Kontext ist **static_cast** genau dann sicher, wenn es sich bei dem Template-Parameter tatsächlich um die richtige abgeleitete Klasse handelt.
- Die Verwendung von **static_cast** in Verbindung mit CRTP geht auf ein 1994 veröffentlichtes Buch von John J. Barton und Lee R. Nackman zurück.

Einige Anwendungen, die durch CRTP möglich werden:

- Statische Klassenvariablen für jede abgeleiteter Klasse. (Beispiel: Zähler für erzeugte bzw. noch lebende Objekte. Bei klassischer OO-Technik würde dies insgesamt gezählt werden, bei CRTP jedoch getrennt nach den einzelnen Instantiierungen.)
- Die abgeleitete Klasse implementiert einige implementierungsspezifische Methoden, die darauf aufbauenden weiteren Methoden kommen durch die Basis-Klasse. (Beispiel: Wenn die abgeleitete Klasse den Operator == unterstützt, kann die Basisklasse darauf basierend den Operator != definieren.
- Verbessertes Namensraum-Management auf Basis des *argument-dependent lookup* (ADL). In der Basisklasse definierte **friend**-Funktionen können so von den abgeleiteten Klassen importiert werden. (Technik von Abrahams und Gurtovoy.)
- Zur Konfliktauflösung bei überladenen Funktions-Templates. (Technik von Abrahams und Gurtovoy.)

- Was können optimierende Übersetzer erreichen?
- Wie lassen sich optimierende Übersetzer unterstützen?
- Welche Fallen können sich durch den Einsatz von optimierenden Übersetzer eröffnen?

Was wird optimiert?

Es gibt zwei teilweise gegensätzliche Ziele der Optimierung:

- Minimierung der Länge des erzeugten Maschinencodes.
- Minimierung der Ausführungszeit.

Es ist relativ leicht, sich dem ersten Ziel zu nähern. Die zweite Problemstellung ist in ihrer allgemeinen Form nicht vorbestimmbar (wegen potentiell unterschiedlicher Eingaben) bzw. in seiner allgemeinen Form nicht berechenbar.

- Die Problemstellung ist grundsätzlich für sehr kleine Sequenzen lösbar.
- Bei größeren Sequenzen wird zwar nicht das Minimum erreicht, dennoch sind die Ergebnisse beachtlich, wenn alle bekannten Techniken konsequent eingesetzt werden.

- Der GNU-Superoptimizer generiert sukzessive alle möglichen Instruktionssequenzen, bis eine gefunden wird, die die gewünschte Funktionalität umsetzt.
- Die Überprüfung erfolgt durch umfangreiche Tests, ist aber kein Beweis, dass die gefundene Sequenz äquivalent zur gewünschten Funktion ist. In der Praxis sind jedoch noch keine falschen Lösungen geliefert worden.
- Der Aufwand des GNU-Superoptimizers liegt bei $O((mn)^{2n})$, wobei m die Zahl der zur Verfügung stehenden Instruktionen ist und n die Länge der kürzesten Sequenz.
- Siehe http://ftp.gnu.org/gnu/superopt/
  (ist von 1995 und lässt sich leider mit modernen C-Übersetzern nicht mehr übersetzen)

- Problemstellung: Gegeben seien zwei nicht-negative ganze Zahlen in den Registern r_1 und r_2. Gewünscht ist das Minimum der beiden Zahlen in r_1.
- Eine naive Umsetzung erledigt dies analog zu einer **if**-Anweisung mit einem Vergleichstest und einem Sprung.
- Folgendes Beispiel zeigt dies für die SPARC-Architektur und den Registern %l0 und %l1:

```
        subcc   %l0,%l1,%g0
        bleu    endif
        nop
        or      %l1,%g0,%l0
endif:
```

```
        subcc   %l1,%l0,%g1
        subx    %g0,%g0,%g2
        and     %g2,%g1,%l1
        addcc   %l1,%l0,%l0
```

- Der Superoptimizer benötigt (auf Thales) weniger als 5 Sekunden, um 28 Sequenzen mit jeweils 4 Instruktionen vorzuschlagen, die allesamt das Minimum bestimmen, ohne einen Sprung zu benötigen. Dies ist eine der gefundenen Varianten.
- Die Instruktionen entsprechen folgendem Pseudo-Code:

```
%g1 = %l1 - %l0
carry = %l0 > %l1? 1: 0
%g2 = -carry
%l1 = %g2 & %g1
%l0 = %l1 + %l0
```

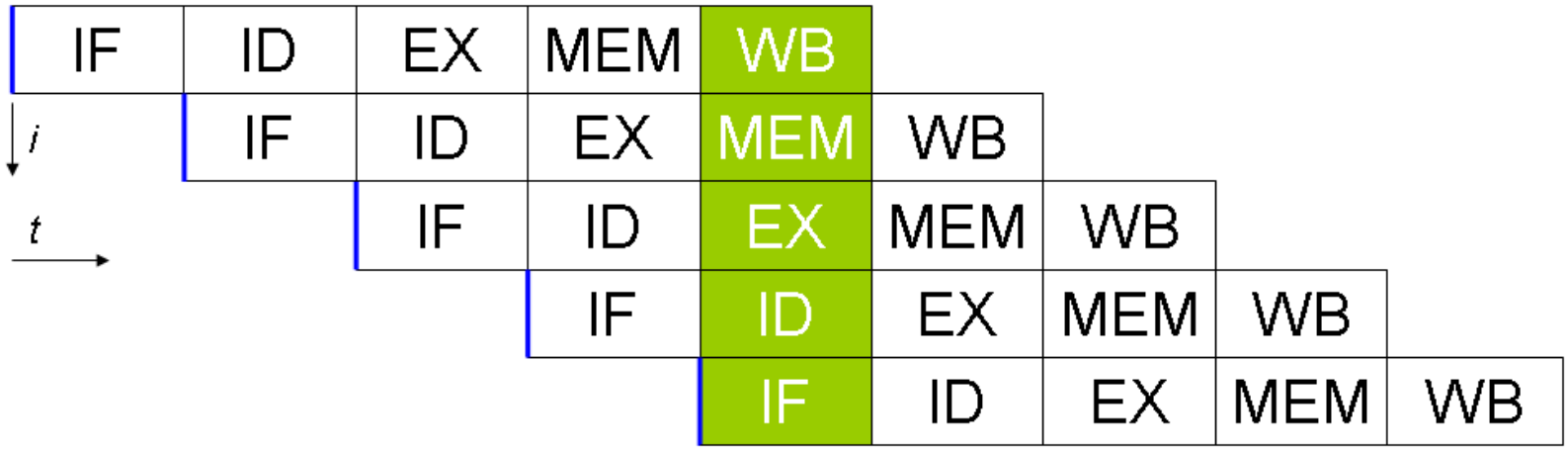
- Generell ist die Vermeidung bedingter Sprünge ein Gewinn, da diese das Pipelining erschweren.

- Moderne Prozessoren arbeiten nach dem Fließbandprinzip: Über das Fließband kommen laufend neue Instruktionen hinzu und jede Instruktion wird nacheinander von verschiedenen Fließbandarbeitern bearbeitet.
- Dies parallelisiert die Ausführung, da unter günstigen Umständen alle Fließbandarbeiter gleichzeitig etwas tun können.
- Eine der ersten Pipelining-Architekturen war die IBM 7094 aus der Mitte der 60er-Jahre mit zwei Stationen am Fließband. Die UltraSPARC-IV-Architektur hat 14 Stationen.
- Die RISC-Architekturen (RISC = *reduced instruction set computer*) wurden speziell entwickelt, um das Potential für Pipelining zu vergrößern.
- Bei der Pentium-Architektur werden im Rahmen des Pipelinings die Instruktionen zuerst intern in RISC-Instruktionen konvertiert, so dass sie ebenfalls von diesem Potential profitieren kann.

Um zu verstehen, was alles innerhalb einer Pipeline zu erledigen ist, hilft ein Blick auf die möglichen Typen von Instruktionen:

- Operationen, die nur auf Registern angewendet werden und die das Ergebnis in einem Register ablegen (wie etwa subcc in den Beispielen).
- Instruktionen mit Speicherzugriff. Hier wird eine Speicheradresse berechnet und dann erfolgt entweder eine Lese- oder eine Schreiboperation.
- Sprünge.

Zerlegung in Arbeitsschritte



Eine einfache Aufteilung sieht folgende einzelne Schritte vor:

- Instruktion vom Speicher laden (IF)
- Instruktion dekodieren (ID)
- Instruktion ausführen, beispielsweise eine arithmetische Operation oder die Berechnung einer Speicheradresse (EX)
- Lese- oder Schreibzugriff auf den Speicher (MEM)
- Abspeichern des Ergebnisses in Registern (WB)

Das Diagramm wurde von Stephane Tsacas geklaut.

- Bedingte Sprünge sind ein Problem für das Pipelining, da unklar ist, wie gesprungen wird, bevor es zur Ausführungsphase kommt.
- RISC-Maschinen führen typischerweise die Instruktion unmittelbar nach einem bedingten Sprung immer mit aus, selbst wenn der Sprung genommen wird. Dies mildert etwas den negativen Effekt für die Pipeline.
- Im übrigen gibt es die Technik der *branch prediction*, bei der ein Ergebnis angenommen wird und dann das Fließband auf den Verdacht hin weiterarbeitet, dass die Vorhersage zutrifft. Im Falle eines Misserfolgs muss dann u.U. recht viel rückgängig gemacht werden.
- Das ist machbar, solange nur Register verändert werden. Manche Architekturen verfolgen die Alternativen sogar parallel und haben für jedes abstrakte Register mehrere implementierte Register, die die Werte für die einzelnen Fälle enthalten.
- Die Vorhersage wird vom Übersetzer generiert. Typisch ist beispielsweise, dass bei Schleifen eine Fortsetzung der Schleife vorhergesagt wird.

Sprungvermeidung am Beispiel einer while-Schleife 395

```
int a[10];
int main() {
   int i = 0;
   while (i < 10 && a[i] != 0) ++i;
}
```

- Eine triviale Umsetzung erzeugt zwei Sprünge: Zwei bedingte Sprünge in der Auswertung der Schleifenbedingung und einen unbedingten Sprung am Ende der Schleife:

```
while:
        mov     %i5,%o2
        subcc   %o2,10,%g0
        bge     endwhile
        nop

        sethi   %hi(a),%o1
        or      %o1,%lo(a),%o1
        sll     %o2,2,%o0
        ld      [%o1+%o0],%o0
        subcc   %o0,0.%g0
        be      endwhile
        nop

        ba      while
        add     %o2,1,%i5
endwhile:
```

Sprungvermeidung am Beispiel einer while-Schleife 396

```
        ba      whilecond
        nop
whilebody:
        add     %l0,1,%l0
whilecond:
        subcc   %l0,10,%g0
        bge     endwhile
        nop
        sethi   %hi(a),%i0
        sll     %l0,2,%i1
        add     %i0,%i1,%i0
        ld      [%i0+%lo(a)],%i0
        subcc   %i0,%g0,%g0
        bne     whilecond
        nop
endwhile:
```

- Die Auswertung der Sprungbedingung erfolgt nun am Ende der Schleife, so dass pro Schleifendurchlauf nur zwei bedingte Sprünge ausgeführt werden. Dafür ist ein zusätzlicher Sprung am Anfang der **while**-Schleife notwendig.

Typische Techniken zur Minimierung des Maschinencodes

- Lokale Variablen und Parameter soweit wie möglich in Registern halten. Dies spart Lade- und Speicherinstruktionen.
- Vereinfachung von Blattfunktionen. Das sind Funktionen, die keine weitere Funktionen aufrufen.
- Auswerten von konstanten Ausdrücken während der Übersetzzeit (*constant folding*).
- Vermeidung von Sprüngen.
- Vermeidung von Multiplikationen, wenn einer der Operanden konstant ist.
- Elimination mehrfach vorkommender Teilausdrücke.
  Beispiel: `a[i + j] = 3 * a[i + j] + 1;`
- Konvertierung absoluter Adressberechnungen in relative.
  Beispiel: `for (int i = 0; i < 10; ++i) a[i] = 0;`
- Datenflussanalyse und Eliminierung unbenötigten Programmtexts

- Ein Übersetzer kann lokale Variablen nur dann permanent in einem Register unterbringen, wenn zu keinem Zeitpunkt eine Speicheradresse benötigt wird.
- Sobald der Adress-Operator & zum Einsatz kommt, muss diese Variable zwingend im Speicher gehalten werden.
- Das gleiche gilt, wenn Referenzen auf die Variable existieren.
- Zwar kann der Übersetzer den Wert dieser Variablen ggf. in einem Register vorhalten. Jedoch muss in verschiedenen Situationen der Wert neu geladen werden, z.B. wenn ein weiterer Funktionsaufruf erfolgt, bei dem ein Zugriff über den Zeiger bzw. die Referenz erfolgen könnte.

```
int index;
while (std::cin >> index) {
   a[index] = f(a[index]);
   a[index] += g(a[index]);
}
```

- In diesem Beispiel wird eine Referenz auf *index* an den >>-Operator übergeben. Der Übersetzer weiß nicht, ob diese Adresse über irgendwelche Datenstrukturen so abgelegt wird, dass die Funktionen *f* und *g* darauf zugreifen. Entsprechend wird der Übersetzer genötigt, immer wieder den Wert von *index* aus dem Speicher zu laden.
- Deswegen ist es ggf. hilfreich, explizit eine weitere lokale Variablen zu verwenden, die eine Kopie des Werts erhält und von der keine Adresse genommen wird:

```
int index;
while (std::cin >> index) {
   int i = index;
   a[i] = f(a[i]);
   a[i] += g(a[i]);
}
```

```
bool find(int* a, int len, int& index) {
   while (index < len) {
      if (a[index] == 0) return true;
      ++index;
   }
   return false;
}
```

- Referenzparameter sollten bei häufiger Nutzung in ausschließlich lokal genutzte Variablen kopiert werden, um einen externen Einfluss auszuschließen.

```
bool find(int* a, int len, int& index) {
   for (int i = index; i < len; ++i) {
      if (a[index] == 0) {
         index = i; return true;
      }
   }
   index = len;
   return false;
}
```

```
void f(int* i, int* j) {
   *i = 1;
   *j = 2;
   // value of *i?
}
```

- Wenn mehrere Zeiger oder Referenzen gleichzeitig verwendet werden, unterbleiben Optimierungen, wenn nicht ausgeschlossen werden kann, dass mehrere davon auf das gleiche Objekt zeigen.
- Wenn der Wert von `*i` verändert wird, dann ist unklar, ob sich auch `*j` verändert. Sollte anschließend auf `*j` zugegriffen werden, muss der Wert erneut geladen werden.
- In C (noch nicht in C++) gibt es die Möglichkeit, mit Hilfe des Schlüsselworts **restrict** Aliasse auszuschließen:

```
void f(int* restrict i, int* restrict j) {
   *i = 1;
   *j = 2;
   // *i == 1 still assumed
}
```

- Mit der Datenflussanalyse werden Abhängigkeitsgraphen erstellt, die feststellen, welche Variablen unter Umständen in Abhängigkeit welcher anderer Variablen verändert werden können.
- Im einfachsten Falle kann dies auch zur Propagation von Konstanten genutzt werden.
  Beispiel: Nach `int a = 7; int b = a;` ist bekannt, dass b den Wert 7 hat.
- Die Datenflussanalyse kann für eine Variable recht umfassend durchgeführt werden, wenn sie lokal ist und ihre Adresse nie weitergegeben wurde.

```
void loopingsleep(int count) {
   for (int i = 0; i < count; ++i)
      ;
}
```

- Mit Hilfe der Datenflussanalyse lässt sich untersuchen, welche Teile einer Funktion Einfluss haben auf den **return**-Wert oder die außerhalb der Funktion sichtbaren Datenstrukturen.
- Anweisungen, die nichts von außen sichtbares verändern, können eliminiert werden.
- Auf diese Weise verschwindet die **for**-Schleife im obigen Beispiel.
- Der erwünschte Verzögerungseffekt lässt sich retten, indem in der Schleife unter Verwendung der Schleifenvariablen eine externe Funktion aufgerufen wird. (Das funktioniert, weil normalerweise keine globale Datenflussanalyse stattfindet.)

- Globale Variablen können ebenso in die Datenflussanalyse einbezogen werden.
- C und C++ gehen davon aus, dass globale Variablen sich nicht überraschend ändern, solange keine Alias-Problematik vorliegt und keine unbekannten Funktionen aufgerufen werden.
- Das ist problematisch, wenn Threads oder Signalbehandler unsynchronisiert auf globale Variablen zugreifen.

sigint.c

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

int signal_caught = 0;

void signal_handler(int signal) {
   signal_caught = signal;
}

int main() {
   if (signal(SIGINT, signal_handler) == SIG_ERR) {
      perror("unable to setup signal handler for SIGINT");
      exit(1);
   }
   printf("Try to send a SIGINT signal!\n");
   int counter = 0;
   while (!signal_caught) {
      for (int i = 0; i < counter; ++i);
      ++counter;
   }
   printf("Got signal %d after %d steps!\n", signal_caught, counter);
}
```

sigint.c

```
int counter = 0;
while (!signal_caught) {
   for (int i = 0; i < counter; ++i);
   ++counter;
}
```

- Hier sieht der Übersetzer nicht, dass sich die globale Variable *signal_caught* innerhalb eines Schleifendurchlaufs verändern könnte.
- Der optimierte Code testet deswegen die Variable *signal_caught* nur ein einziges Mal beim Schleifenantritt und danach nicht mehr. Entsprechend gibt es keinen Schleifenabbruch, wenn der Signalbehandler aktiv wird.

```
dairinis$ gcc -o sigint -std=c99 sigint.c
dairinis$ sigint
Try to send a SIGINT signal!
^CGot signal 2 after 24178 steps!
dairinis$ gcc -o sigint -O -std=c99 sigint.c
dairinis$ sigint
Try to send a SIGINT signal!
^C^Cdairinis$
```

```
volatile sig_atomic_t signal_caught = 0;
```

- Mit dem Schlüsselwort **volatile** können globale Variablen gekennzeichnet werden, die sich überraschend ändern können.
- Zusätzlich wurde hier noch korrekterweise der Datentyp *sig_atomic_t* anstelle von **int** verwendet, um die Atomizität eines Schreibzugriffs sicherzustellen.
- C und C++ garantieren, dass Zuweisungen an **volatile**-Objekte in der Ausführungsreihenfolge erfolgen. (Das ist bei anderen Objekten nicht zwangsläufig der Fall.)
- Beispiel: `a = 2; a = 3;`
  Hier würde normalerweise `a = 2` wegoptimiert werden. Wenn *a* jedoch eine **volatile**-Variable ist, wird ihr zuerst 2 zugewiesen und danach die 3.
- Wenn notwendige **volatile**-Auszeichnungen vergessen werden, können Programme bei eingeschaltetem Optimierer ein fehlerhaftes Verhalten aufweisen.

Optimierende Übersetzer bieten typischerweise Stufen an. Recht typisch ist dabei folgende Aufteilung des gcc:

Stufe	Option	Vorteile
0		schnelle Übersetzung, mehr Transparenz beim Debugging
1	-O1	lokale Peephole-Optimierungen
2	-O2	Minimierung des Umfangs des generierten Codes
3	-O3	Minimierung der Laufzeit mit ggf. umfangreicheren Code

Bei der (oder den) höchsten Optimierungsstufe(n) wird teilweise eine erhebliche Expansion des generierten Codes in Kauf genommen, um Laufzeitvorteile zu erreichen. Die wichtigsten Techniken:

- Loop unrolling
- Instruction scheduling
- Function inlining

Loop unrolling


```
for (int i = 0; i < 100; ++i) {
   a[i] = i;
}
```

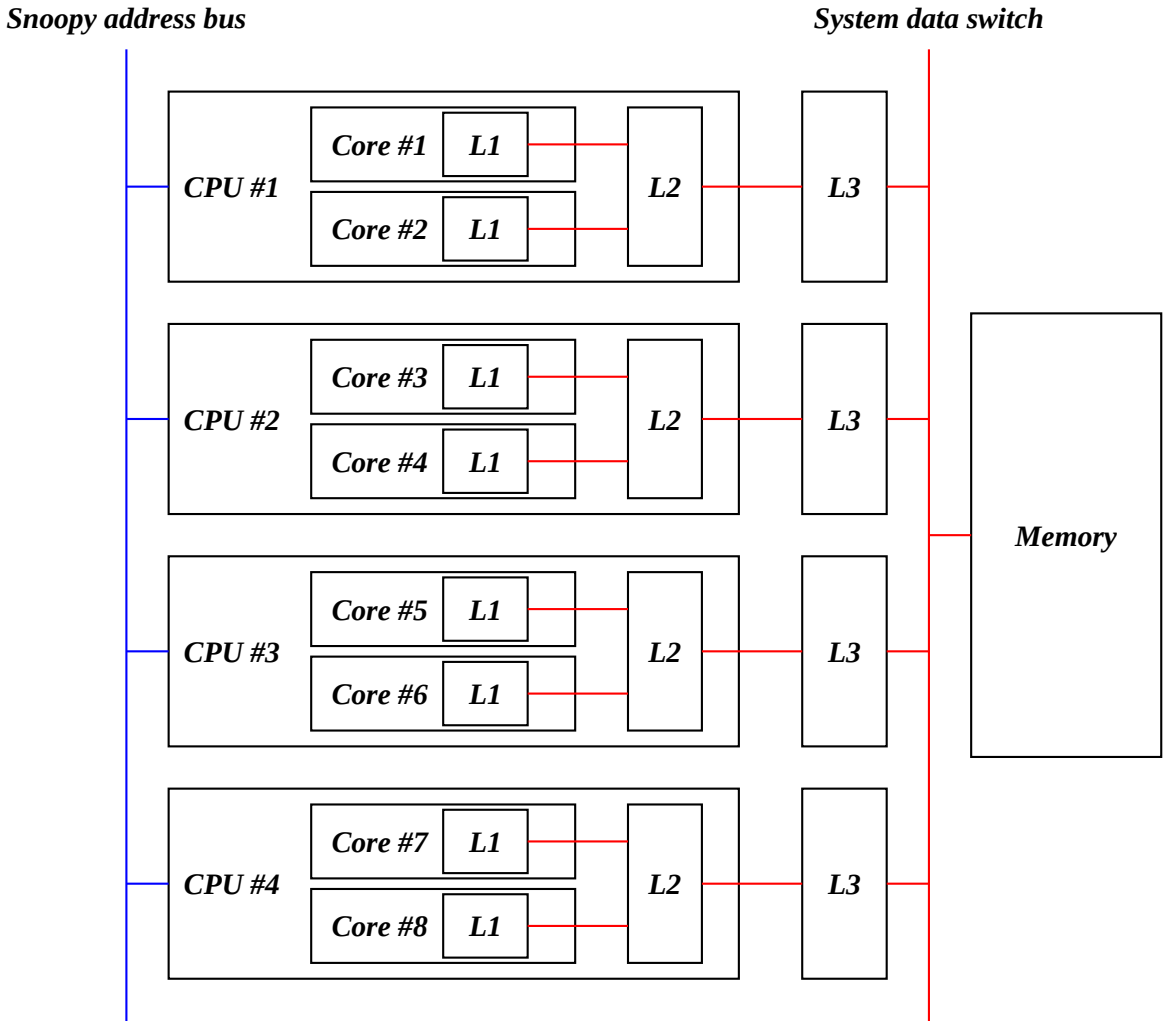
- Diese Technik reduziert deutlich die Zahl der bedingten Sprünge, indem mehrere Schleifendurchläufe in einem Zug erledigt werden.
- Das geht nur, wenn die einzelnen Schleifendurchläufe unabhängig voneinander erfolgen können, d.h. kein Schleifendurchlauf von den Ergebnissen der früheren Durchgänge abhängt.
- Dies wird mit Hilfe der Datenflussanalyse überprüft, wobei sich der Übersetzer auf die Fälle beschränkt, bei denen er sich sicher sein kann. D.h. nicht jede für diese Technik geeignete Schleife wird auch tatsächlich entsprechend optimiert.

```
for (int i = 0; i < 100; i += 4) {
   a[i] = i;
   a[i+1] = i + 1;
   a[i+2] = i + 2;
   a[i+3] = i + 3;
}
```

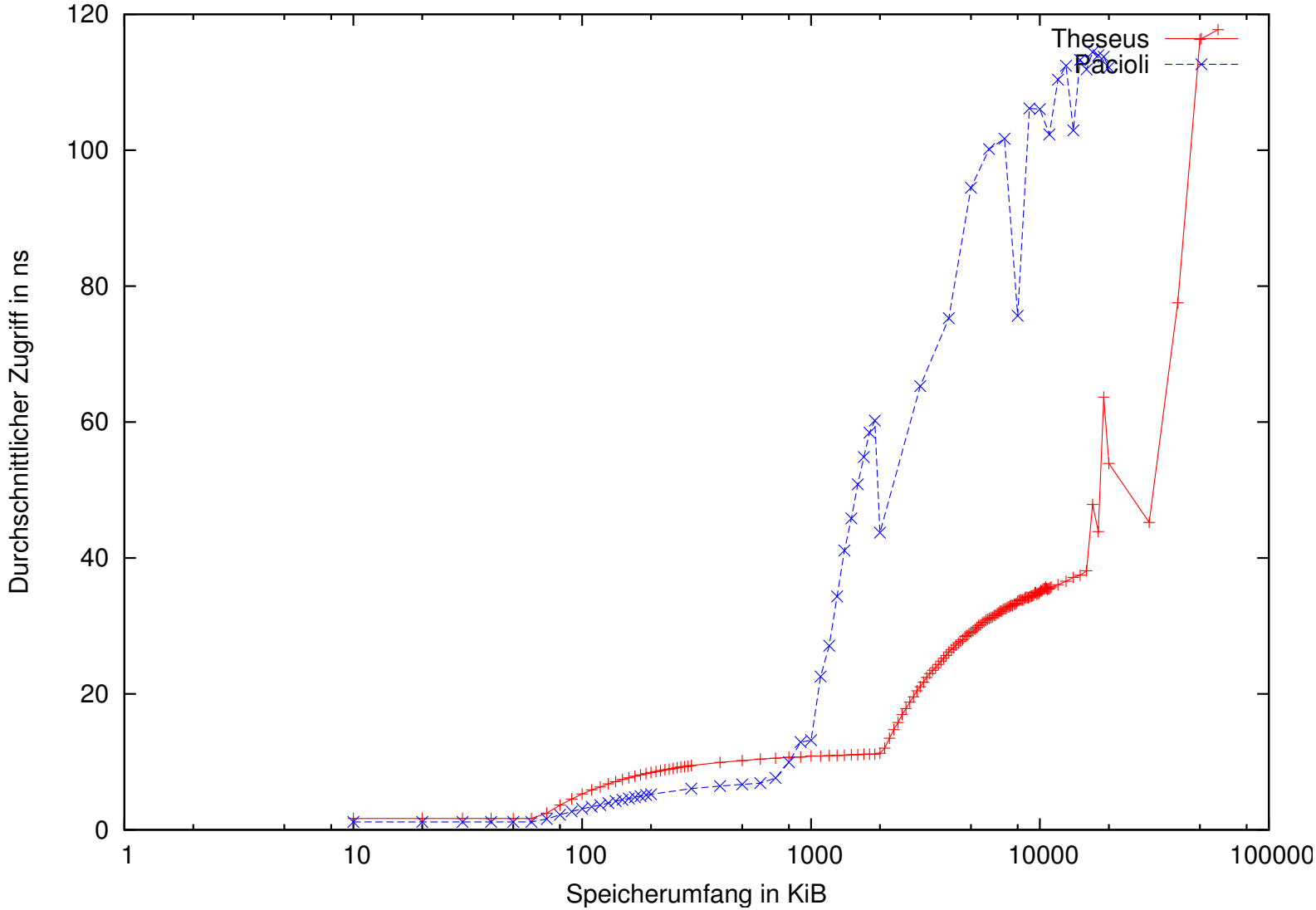
- Zugriffe einer CPU auf den primären Hauptspeicher sind vergleichsweise langsam. Obwohl Hauptspeicher generell schneller wurde, behielten die CPUs ihren Geschwindigkeitsvorsprung.
- Grundsätzlich ist Speicher direkt auf einer CPU deutlich schneller. Jedoch lässt sich Speicher auf einem CPU-Chip aus Komplexitäts-, Produktions- und Kostengründen nicht beliebig ausbauen.
- Deswegen arbeiten moderne Architekturen mit einer Kette hintereinander geschalteter Speicher. Zur Einschätzung der Größenordnung sind hier die Angaben für die Theseus, die mit Prozessoren des Typs UltraSPARC IV+ ausgestattet ist:

Cache	Kapazität	Taktzyklen
Register		1
L1-Cache	64 KiB	2-3
L2-Cache	2 MiB	um 10
L3-Cache	32 MiB	um 60
Hauptspeicher	32 GiB	um 250

Cache-Struktur der Theseus



Gemessene Speicherzugriffszeiten



- Theseus (rot): L1 (64 KiB), L2 (2 MiB), L3 (32 MiB)
- Pacioli (blau): L1 (64 KiB), L2 (1 MiB)

- Ein Cache ist in sogenannten *cache lines* organisiert, d.h. eine *cache line* ist die Einheit, die vom Hauptspeicher geladen oder zurückgeschrieben wird.
- Jede der *cache lines* umfasst – je nach Architektur – 32 - 128 Bytes. Auf der Theseus sind es beispielsweise 64 Bytes.
- Jede der *cache lines* kann unabhängig voneinander gefüllt werden und einem Abschnitt im Hauptspeicher entsprechen.
- Das bedeutet, dass bei einem Zugriff auf `a[i]` mit recht hoher Wahrscheinlichkeit auch `a[i+1]` zur Verfügung steht.

- Diese Technik bemüht sich darum, die Instruktionen (soweit dies entsprechend der Datenflussanalyse möglich ist) so anzuordnen, dass in der Prozessor-Pipeline keine Stockungen auftreten.
- Das lässt sich nur in Abhängigkeit des konkret verwendeten Prozessors optimieren, da nicht selten verschiedene Prozessoren der gleichen Architektur mit unterschiedlichen Pipelines arbeiten.
- Ein recht großer Gewinn wird erzielt, wenn ein vom Speicher geladener Wert erst sehr viel später genutzt wird.
- Beispiel: `x = a[i] + 5; y = b[i] + 3;`
  Hier ist es sinnvoll, zuerst die Ladebefehle für `a[i]` und `b[i]` zu generieren und erst danach die beiden Additionen durchzuführen und am Ende die beiden Zuweisungen.

axpy.c

```
// y = y + alpha * x
void axpy(int n, double alpha, const double* x, double* y) {
   for (int i = 0; i < n; ++i) {
      y[i] += alpha * x[i];
   }
}
```

- Dies ist eine kleine Blattfunktion, die eine Vektoraddition umsetzt. Die Länge der beiden Vektoren ist durch n gegeben, x und y zeigen auf die beiden Vektoren.
- Aufrufkonvention:

Variable	Register
n	%o0
alpha	%o1 und %o2
x	%o3
y	%o4

Resultat von gcc -O3

axpy.s

```
        add     %sp, -120, %sp
        cmp     %o0, 0
        st      %o1, [%sp+96]
        st      %o2, [%sp+100]
        ble     .LL5
        ldd     [%sp+96], %f12
        mov     0, %g2
        mov     0, %g1
.LL4:
        ldd     [%g1+%o3], %f10
        ldd     [%g1+%o4], %f8
        add     %g2, 1, %g2
        fmuld   %f12, %f10, %f10
        cmp     %o0, %g2
        faddd   %f8, %f10, %f8
        std     %f8, [%g1+%o4]
        bne     .LL4
        add     %g1, 8, %g1
.LL5:
        jmp     %o7+8
        sub     %sp, -120, %sp
```

- Ein *loop unrolling* fand hier nicht statt, wohl aber ein *instruction scheduling*.

- Der C-Compiler von Sun generiert für die gleiche Funktion 241 Instruktionen (im Vergleich zu den 19 Instruktionen beim gcc).
- Der innere Schleifenkern mit 81 Instruktionen behandelt 8 Iterationen gleichzeitig. Das orientiert sich exakt an der Größe der *cache lines* der Architektur: $8 * \textbf{sizeof}(\textbf{double}) == 64$.
- Mit Hilfe der prefetch-Instruktion wird dabei jeweils noch zusätzlich dem Cache der Hinweis gegeben, die jeweils nächsten 8 Werte bei x und y zu laden.
- Der Code ist deswegen so umfangreich, weil
  - die Randfälle berücksichtigt werden müssen, wenn n nicht durch 8 teilbar ist und
  - die Vorbereitung recht umfangreich ist, da der Schleifenkern von zahlreichen bereits geladenen Registern ausgeht.

- Der gcc kann mit der Option „-funroll-loops“ ebenfalls dazu überredet werden, Schleifen zu expandieren.
- Bei diesem Beispiel werden dann ebenfalls 8 Iterationen gleichzeitig behandelt.
- Der innere Schleifenkern besteht beim gcc nur aus 51 Instruktionen – ein *prefetch* entfällt und das Laden aus dem Speicher wird nicht an den Schleifenanfang vorgezogen. Entsprechend wird hier das Optimierungspotential noch nicht ausgereizt.

- Hierbei wird auf den Aufruf einer Funktion verzichtet. Stattdessen wird der Inhalt der Funktion genau dort expandiert, wo sie aufgerufen wird.
- Das läuft so ähnlich ab wie bei der Verwendung von Makros, nur gelten weiterhin die bekannten Regeln.
- Das kann jedoch nur gelingen, wenn der Programmtext der aufzurufenden Funktion bekannt ist.
- Das klappt bei Funktionen, die in der gleichen Übersetzungseinheit enthalten sind und in C++ bei Templates, bei denen der benötigte Programmtext in der entsprechenden Headerdatei zur Verfügung steht.
- Letzteres wird in C++ intensiv (etwa bei der STL oder der *iostreams*-Bibliothek) genutzt, was teilweise erhebliche Übersetzungszeiten mit sich bringt.