Objektorientierte Programmierung mit C++
SS 2018

Andreas F. Borchert

Universität Ulm

17. Juli 2018

Syllabus

Inhalte:

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

- Im August 2011 wurde der ISO-Standard für C++ veröffentlicht, ISO 14882-2012, der beachtliche Neuerungen einführt im Vergleich zu dem vorherigen Standard von 2003.
- Diese Änderungen sind so wesentlich, dass sich die Programmierung im Vergleich zum alten C++ grundlegend geändert hat.
- Im August 2014 wurde C++14 verabschiedet, im Dezember 2017 wurde mit C++17 die aktuelle Fassung des Standards veröffentlicht. Beide rundeten die Erweiterungen von C++11 ab.
- Der nächste Standard, voraussichtlich C++20, wird wohl die langersehnten *concepts* erhalten. Da diese von *gcc* bereits unterstützt werden, können wir darauf schon einen Blick werfen.
- Ein wesentliches Ziel der Vorlesung ist es zu zeigen, wie moderne Sprachtechniken effizient oder sogar effizienzsteigernd eingesetzt werden können.

- Für C++17 kommt nur GCC ab Version 7 in Frage oder Clang.
- Im Rahmen der Vorlesung werden wir primär mit GCC 7.3.0 arbeiten, die auf der Theon zur Verfügung steht.
- Auf den Maschinen im Poolraum in E.44 steht mit *g++-7.2* der GCC 7.2 zur Verfügung.
- Um den GCC 7.3.0 auf der Theon nutzen zu können, sollten Sie bei uns „ballinrobe“ in der Datei *~/.options* aufnehmen.

- 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, Lambda-Ausdrücke und Metaprogrammierung.
- Erwerb von Grundkenntnissen über die Implementierungen verschiedener Techniken, so dass das zu erwartende Laufzeitverhalten eingeschätzt werden kann

- Jede Woche gibt es zwei Vorlesungsstunden an jedem Dienstag von 14-16 Uhr im Raum E.03 in der Helmholtzstraße 22.
- Die Übungen finden an jedem Donnerstag von 16-18 Uhr im Raum E.44 in der Helmholtzstraße 18 statt.
- Webseite: `https://www.uni-ulm.de/mawi/mawi-numerik/lehre/sommersemester-2018/vorlesung-objektorientierte-programmierung-mit-c/`
- Alle Vorlesungsteilnehmer mögen sich bitte bei SLC für die Vorlesung registrieren.

Syllabus: Übungen

- Es gibt Übungen an jedem Donnerstag von 16-18 Uhr.
- Diese finden im Poolraum E.44 neben der Mathe-Bibliothek in der Helmholtzstraße 18 statt.
- Dies eröffnet die Möglichkeit, mit der Übungssitzung oder dem Übungsblatt sofort zu beginnen und Fragen dazu zu stellen oder Unterstützung bei aufkommenden Problemen zu erhalten.
- Die praktische Abwicklung der Übungen wird in den ersten Übungen am 19. April vorgestellt.
- 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 zu erwarten ist.

- Es gibt zwei schriftliche Prüfungen am Ende des Semesters in der Prüfungsperiode.
- 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.
- Die Prüfungstermine sind offen, d.h. Sie können auch den zweiten Prüfungstermin wahrnehmen, ohne an der ersten Prüfung teilgenommen zu haben.
- Es wird rechtzeitig vor der ersten Prüfung eine Probeklausur geben, damit Sie sich besser darauf vorbereiten können.

- Im nächsten Sommersemester 2018 werde ich die Vorlesung *Parallele Programmierung mit C++* anbieten.
- Diese Vorlesung geht ausführlich auf die verschiedenen Möglichkeiten und Techniken der Parallelisierung ein. Dies erfolgt sowohl aus der Perspektive der Architekturen als auch aus der Sicht der Programmierung.
- Es geht in der Vorlesung darum, die Grundlagen dafür zu erlenen, die es erlauben, geeignete Architekturen für parallelisierbare Problemstellungen auszuwählen und dazu passende Algorithmen zu entwickeln.
- Hierfür ist ebenfalls C++ im besonderen Maße geeignet. Seit C++11 sind insbesondere auch Threads Bestandteil des Standards.

Syllabus: Literatur

- 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):
  - August 2010: Letzte Arbeitsfassung vor C++11.
  - Oktober 2013: Diese Arbeitsfassung wurde zur Grundlage von C++14.
  - Juli 2016: aktuelle Arbeitsfassung für C++17, wird teilweise von GCC 6.x und vollständig von 7.x unterstützt.
- Die Standards können im Original als PDF von ISO oder den nationalen Standardorganisationen bezogen werden, sind jedoch dort leider sündhaft teuer.

- Bjarne Stroustrup, *The C++ Programming Language*, ISBN 0-321-56384-0 (vierte Auflage, die C++11 berücksichtigt)
- Bjarne Stroustrup, *Programming: Principles and Practice Using C++*, ISBN 978-0-321-99278-9 (zweite Auflage, berücksichtigt C++14)
- 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
- 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
- Scott Meyers, *Effective Modern C++*, ISBN 978-1-491-90399-5

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)

- 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 20, Zimmer 1.23.
- 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.
  - Charakteristisch ist, dass das gesamte Typsystem dynamisch ist und somit keine statische Typsicherheit existiert: *message not understood*

- 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.
- Wann findet die Lokalisierung statt? Nur zur Laufzeit oder auch teilweise zur Übersetzzeit?

- 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: Potential für eine 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 den Ausbau der zur Übersetzzeit ausgewerteten **constexpr**-Funktionen (eingeführt in C++11, erheblich erweitert für C++14) hat sich die Metaprogrammierung vereinfacht.
- Auch durch spezielle Bibliotheken für die Metaprogrammierung wird die Anwendung 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.

- 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.

- Mit C++11 erfolgte eine sehr umfangreiche Revision, die u.a. folgende Features einführte:
  - Rvalue-Referenzen zusammen mit *move constructors*,
  - **constexpr** zur Berechnung von Ausdrücken und Funktionen zur Übersetzzeit,
  - automatische Ableitung eines Typen (Typinferenz),
  - beliebig viele Parameter für Templates,
  - Lambda-Ausdrücke,
  - Threads und
  - *smart pointers*.
- Diese Erweiterungen wurden in C++14 und C++17 weiter ausgebaut und abgerundet. Der aktuelle Standard wurde im Dezember 2017 veröffentlicht und wird kurz C++17 genannt.

- In C++ sind Übersetzungseinheiten Dateien mit Programmtext, die dem C++-Übersetzer unmittelbar auf der Kommandozeile zum Übersetzen angegeben werden. Als Dateiendung wird hier gerne „.cpp“ benutzt – es sind aber auch viele andere üblich wie etwa „.C“ oder „.cc“.
- Mit Hilfe der **#include**-Direktive des Präprozessors können noch sehr viel mehr Programmtexte indirekt hinzukommen.
- Eine Übersetzungseinheit wird normalerweise direkt in Maschinencode für eine ausgewählte Plattform übersetzt (normalerweise die lokale, ein *cross compiler* kann auch für andere Plattformen übersetzen). Diese Resultate werden auch Objekte genannt (hat nichts mit den OO-Konzepten zu tun).
- Mit Hilfe des *ld* (*linkage editor*) können mehrere Objekte zusammen mit den Bibliotheken zu einem ausführbaren Programm zusammengefügt werden.

⟨translation-unit⟩ ⟶ [⟨declaration-seq⟩]

⟨declaration-seq⟩ ⟶ ⟨declaration⟩

⟶ ⟨declaration-seq⟩ ⟨declaration⟩

⟨declaration⟩ ⟶ ⟨block-declaration⟩

⟶ ⟨nodeclspec-function-declaration⟩

⟶ ⟨function-definition⟩

⟶ ⟨template-declaration⟩

⟶ ⟨deduction-guide⟩

⟶ ⟨explicit-instantiation⟩

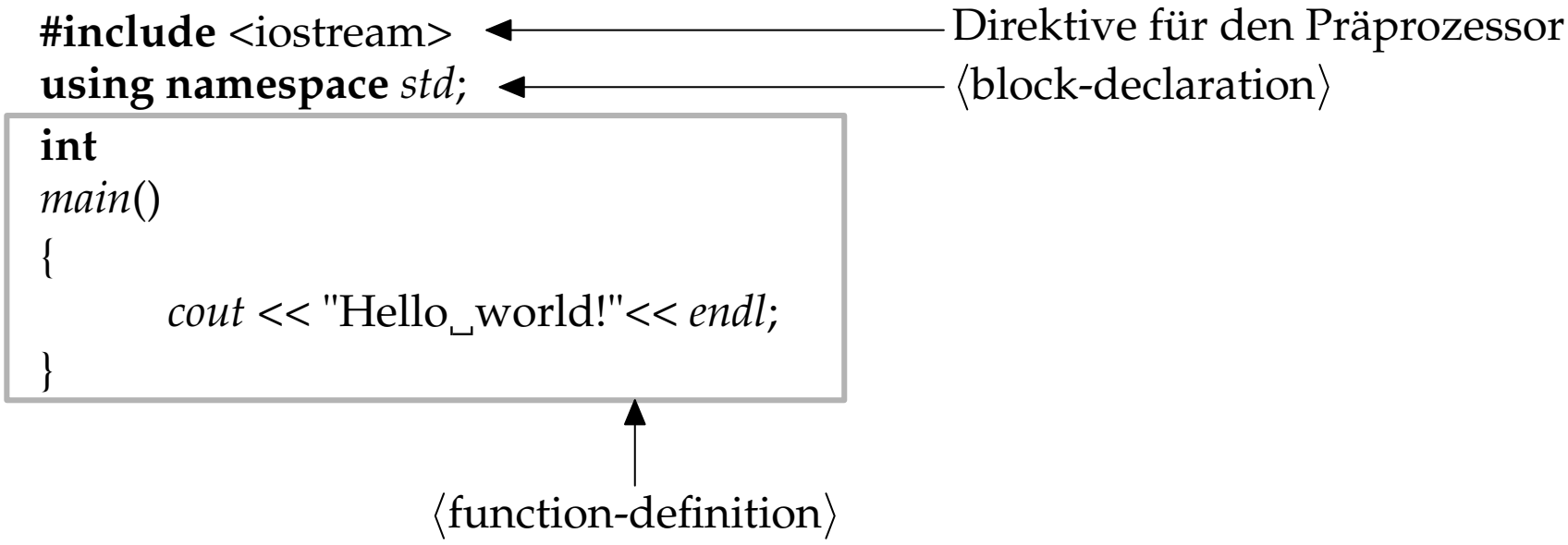
⟶ ⟨explicit-specialization⟩

⟶ ⟨linkage-specification⟩

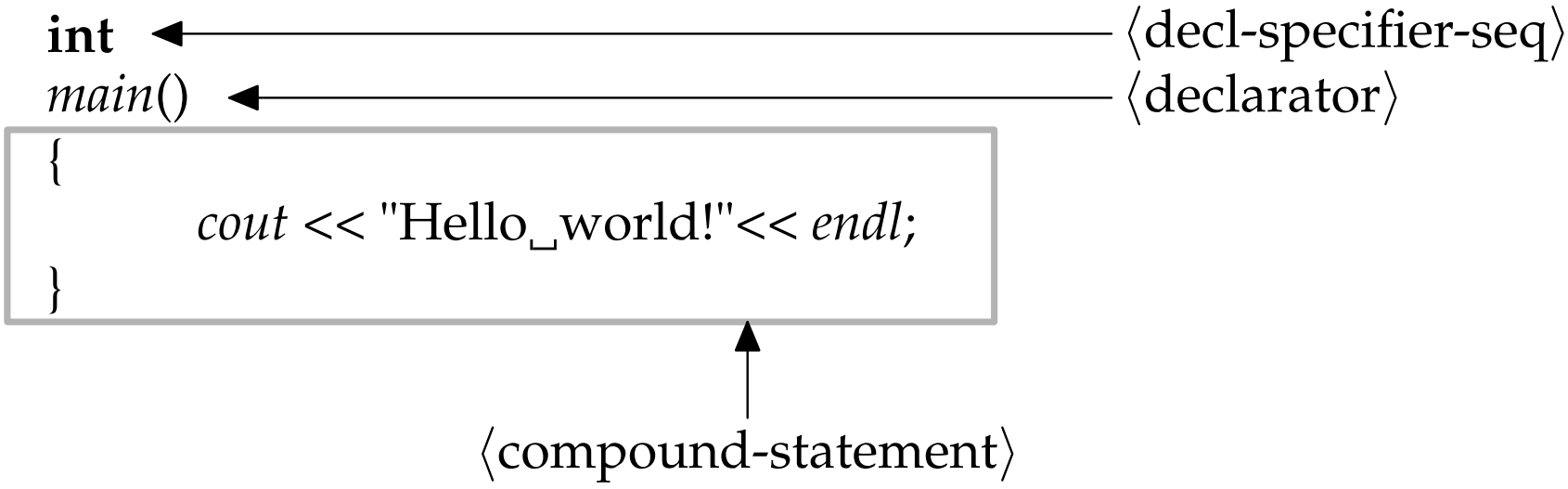
⟶ ⟨namespace-definition⟩

⟶ ⟨empty-declaration⟩

⟶ ⟨attribute-declaration⟩



- Präprozessor-Anweisungen betten sich nicht in die C++-Syntax ein. Durch den Präprozessor werden sie durch Text ersetzt, der der C++-Syntax entsprechend sollte. Bei **#include**-Direktiven ist dies der Inhalt der gegebenen Datei (hier *iostream*, die standardmäßig zur Verfügung steht).
- Mit **using namespace** *std* lässt sich alles aus dem *std*-Namensraum ohne Qualifikation verwenden, also etwa *cout* anstelle von *std::cout*.



⟨function-definition⟩ ⟶ [⟨attribute-specifier-seq⟩]
[⟨decl-specifier-seq⟩]
⟨declarator⟩ [⟨virt-specifier-seq⟩]
⟨function-body⟩

⟨function-body⟩ ⟶ [⟨ctor-initializer⟩]
⟨compound-statement⟩
⟶ ⟨function-try-block⟩
⟶ „=“ **default** „;“
⟶ „=“ **delete** „;“

- Normalerweise ist nur die erste Variante interessant.
- Der ⟨function-try-block⟩ erlaubt eine saubere Lösung der Ausnahmenbehandlung bei Konstruktoren mit Sub-Konstruktoren, die möglicherweise Ausnahmenbehandlungen auslösen.
- Die letzteren beiden Varianten betreffen nur einige standardmäßig unterstützte Methoden – wir kommen darauf noch zurück.
- *ctor* steht für *constructor* – entsprechend kann ein ⟨ctor-initializer⟩ nur bei Konstruktoren vorkommen.

$$
\begin{aligned}
\langle \text{block-declaration} \rangle \quad &\longrightarrow \quad \langle \text{simple-declaration} \rangle \\
&\longrightarrow \quad \langle \text{asm-definition} \rangle \\
&\longrightarrow \quad \langle \text{namespace-alias-definition} \rangle \\
&\longrightarrow \quad \langle \text{using-declaration} \rangle \\
&\longrightarrow \quad \langle \text{using-directive} \rangle \\
&\longrightarrow \quad \langle \text{static\_assert-declaration} \rangle \\
&\longrightarrow \quad \langle \text{alias-declaration} \rangle \\
&\longrightarrow \quad \langle \text{opaque-enum-declaration} \rangle
\end{aligned}
$$

- Blockdeklarationen können in C++ sowohl auf globaler Ebene als auch innerhalb eines Blocks (d.h. inmitten regulären Programmtexts) erfolgen.

⟨simple-declaration⟩ ⟶ ⟨decl-specifier-seq⟩
[⟨init-declarator-list⟩] „;“
⟶ ⟨attribute-specifier-seq⟩
⟨decl-specifier-seq⟩
⟨init-declarator-list⟩ „;“
⟶ [⟨attribute-specifier-seq⟩] ⟨decl-specifier-seq⟩
[⟨ref-qualifier⟩]
„[“ ⟨identifier-list⟩ „]“ ⟨initializer⟩ „;“

- Die Mehrzahl der Deklarationen in C++ fällt unter die Rubrik der ⟨simple-declaration⟩. Dazu gehören u.a. Variablen- und Klassendeklarationen.
- Die wichtigsten Teile einer Deklaration sind die ⟨decl-specifier⟩, die den Grundtyp spezifizieren und der ⟨declarator⟩, der einen Namen mit einer möglicherweise abgeleiteten Variante des Grundtyps assoziiert.

⟨decl-specifier-seq⟩ ⟶ ⟨decl-specifier⟩
[⟨attribute-specifier-seq⟩]
⟶ ⟨decl-specifier⟩
⟨decl-specifier-seq⟩
⟨decl-specifier⟩ ⟶ ⟨storage-class-specifier⟩
⟶ ⟨defining-type-specifier⟩
⟶ ⟨function-specifier⟩
⟶ **friend**
⟶ **typedef**
⟶ **constexpr**
⟶ **inline**

$$
\begin{aligned}
\langle \text{defining-type-specifier} \rangle &\longrightarrow \langle \text{type-specifier} \rangle \\
&\longrightarrow \langle \text{class-specifier} \rangle \\
&\longrightarrow \langle \text{enum-specifier} \rangle \\
\langle \text{type-specifier} \rangle &\longrightarrow \langle \text{simple-type-specifier} \rangle \\
&\longrightarrow \langle \text{elaborated-type-specifier} \rangle \\
&\longrightarrow \langle \text{typename-specifier} \rangle \\
&\longrightarrow \langle \text{cv-specifier} \rangle
\end{aligned}
$$

⟨simple-type-specifier⟩ ⟶ [⟨nested-name-specifier⟩] ⟨type-name⟩
⟶ ⟨nested-name-specifier⟩ **template** ⟨simple-template-id⟩
⟶ [⟨nested-name-specifier⟩] ⟨template-name⟩
⟶ **char** | **char16_t** | **char32_t** | **wchar_t**
⟶ **bool** | **short** | **int** | **long**
⟶ **signed** | **unsigned**
⟶ **float** | **double**
⟶ **void** | **auto**
⟶ ⟨decltype-specifier⟩

- ⟨simple-type-specifier⟩ schließt alle elementaren Datentypen ein. **auto** zwingt den Übersetzer, den Datentyp selbst zu deduzieren.

$$
\begin{array}{rcl}
\langle \text{type-name} \rangle & \longrightarrow & \langle \text{class-name} \rangle \\
& \longrightarrow & \langle \text{enum-name} \rangle \\
& \longrightarrow & \langle \text{typedef-name} \rangle \\
& \longrightarrow & \langle \text{simple-template-id} \rangle \\
\langle \text{decltype-specifier} \rangle & \longrightarrow & \textbf{decltype} \text{ „(“ } \langle \text{expression} \rangle \text{ „)“} \\
& \longrightarrow & \textbf{decltype} \text{ „(“ } \textbf{auto} \text{ „)“}
\end{array}
$$

- Mit **decltype** kann ein Datentyp von einem Ausdruck abgeleitet werden.

⟨class-name⟩ ⟶ ⟨identifier⟩
⟶ ⟨simple-template-id⟩
⟨class-specifier⟩ ⟶ ⟨class-head⟩
„{“ [⟨member-specification⟩] „}“
⟨class-head⟩ ⟶ ⟨class-key⟩ [⟨attribute-specifier-seq⟩]
⟨class-head-name⟩ [⟨class-virt-specifier⟩]
[⟨base-clause⟩]
⟶ ⟨class-key⟩ [⟨attribute-specifier-seq⟩]
[⟨base-clause⟩]
⟨class-key⟩ ⟶ **class**
⟶ **struct**
⟶ **union**

- Bei **class** sind alle Felder und Methoden per Voreinstellung **private**, bei **struct** sind sie (in Kompatibilität zu C) per Voreinstellung **public**. Bei einer **union** werden sämtliche Datenfelder übereinander gelegt.

Deklarationen innerhalb einer Klasse

⟨member-specification⟩ ⟶ ⟨member-declaration⟩
[⟨member-specification⟩]
⟶ ⟨access-specifier⟩ „:“
[⟨member-specification⟩]

⟨member-declaration⟩ ⟶ [⟨attribute-specifier-seq⟩]
[⟨decl-specifier-seq⟩]
[⟨member-declarator-list⟩] „;“
⟶ ⟨function-definition⟩
⟶ ⟨using-declaration⟩
⟶ ⟨static_assert-declaration⟩
⟶ ⟨template-declaration⟩
⟶ ⟨deduction-guide⟩
⟶ ⟨alias-declaration⟩
⟶ ⟨empty-declaration⟩

Greeting.hpp

```
#ifndef GREETING_H
#define GREETING_H

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

#endif
```

- Klassendeklarationen (mitsamt allen öffentlichen und auch privaten Datenfeldern und Methoden) sind in Dateien, die mit ».hpp«, ».hh« oder ».h« enden, unterzubringen. Hierbei steht ».h« allgemein für eine Header-Datei bzw. ».hh« oder ».hpp« für eine 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. (Vorsicht ist hier u.a. bei dem *HFS+*-Dateisystem von Apple geboten.)
- 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 Operanden einen *ostream* und als rechten Operanden 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"

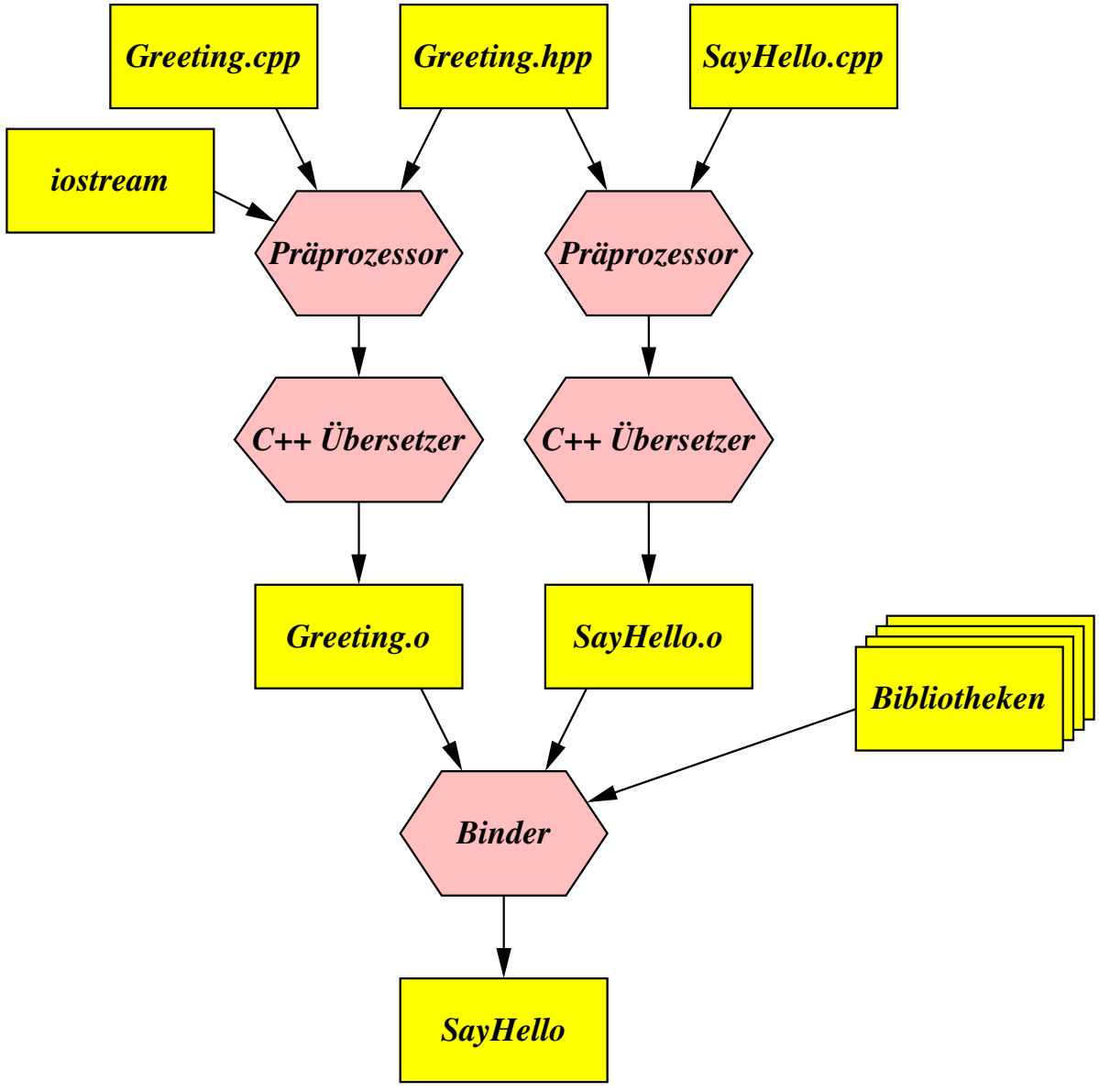
int main() {
   Greeting greeting;
   greeting.hello();
   greeting.hello();
   return 0;
} // main()
```

- 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.

```
theon$ ls
Greeting.cpp  Greeting.hpp  SayHello.cpp
theon$ wget --quiet \
> http://www.mathematik.uni-ulm.de/numerik/cpp/ss18/Makefile
theon$ make depend
gcc-makedepend  Greeting.cpp SayHello.cpp
theon$ make
g++ -Wall -g -std=gnu++17   -c -o Greeting.o Greeting.cpp
g++ -Wall -g -std=gnu++17   -c -o SayHello.o SayHello.cpp
g++ -o SayHello  Greeting.o SayHello.o
theon$ ./SayHello
Hello, fans of C++!
Hello, fans of C++!
theon$ make realclean
rm -f Greeting.o SayHello.o
rm -f SayHello
theon$ ls
Greeting.cpp  Greeting.hpp  Makefile  SayHello.cpp
theon$
```

- *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.

```
theon$ wget --quiet \
> http://www.mathematik.uni-ulm.de/numerik/cpp/ss18/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.

```
theon$ 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 von Github beziehen. Es ist in Perl geschrieben und sollte mit jeder üblichen Perl-Installation zurechtkommen.

Vollständige Implementierung im Header

Greeting.hpp

```
#ifndef GREETING_H
#define GREETING_H

#include <iostream>

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

#endif
```

- Es ist auch möglich, die Methoden innerhalb des Headers direkt zu implementieren.
- Das verlangsamt die Übersetzungszeiten und es ist nicht sichergestellt, dass der Code für die Methodenimplementierungen nur einmal existiert, wenn diese mehrfach per **#include** einkopiert und somit übersetzt werden.

Greeting.hpp

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

- Es ist auch möglich, dem Übersetzer nahezulegen, auf den Methodenaufruf zu verzichten und stattdessen diesen mit der Implementierung der Methode zu ersetzen.
- Ob dies sinnvoll ist, hängt u.a. auch davon ab, wie umfangreich die Methode ist.
- Das ist nicht möglich mit Methoden, deren zugehörige Implementierung zur Laufzeit gesucht wird (dynamischer Polymorphismus).

Lebensort und -dauer eines Objekts

SayHello.cpp

```
#include "Greeting.hpp"

Greeting greeting1;

int main() {
   greeting1.hello();

   Greeting greeting2;
   greeting2.hello();

   Greeting* greeting3 = new Greeting();
   greeting3->hello();
   delete greeting3;
} // main()
```

- Global erzeugte Objekte wie *greeting1* werden vor dem Aufruf von *main* erzeugt und erst nach dem Verlassen von *main* abgebaut.
- Lokale Variablen wie *greeting2* werden jedesmal erzeugt, wenn der umgebende Block erzeugt wird und beim Verlassen des Blocks automatisch abgebaut.
- Mit **new** kann ein Objekt dynamisch auf dem Heap erzeugt werden. Dieses existiert, bis es explizit mit **delete** wieder abgebaut wird.

Klassen in C++ verhalten sich völlig anders als solche in Java und vielen anderen objekt-orientierten Programmiersprachen:

- Wir haben keine implizite Zeigersemantik.
- Objekte einer Klasse können (wenn nichts anderes bestimmt ist) kopiert und als Wert einer Funktion oder Methode übergeben werden (*call by value*).
- Objekte können nicht nur auf dem Heap leben.
- Objekte werden immer in wohldefinierter Weise abgebaut.

vector2d.cpp

```
#include <iostream>

class Vector2D {
   public:
      double x, y;
};

void print_point(Vector2D v) {
   std::cout << "(" << v.x << ", " << v.y << ")";
}

Vector2D add_vectors(Vector2D v1, Vector2D v2) {
   Vector2D result;
   result.x = v1.x + v2.y; result.y = v1.y + v2.y;
   return result;
}

int main() {
   Vector2D a; a.x = 1; a.y = 2; Vector2D b; b.x = 10; b.y = 20;
   Vector2D c = add_vectors(a, b);
   print_point(c); std::cout << std::endl;
}
```

vector2d.cpp

```
Vector2D add_vectors(Vector2D v1, Vector2D v2) {
   Vector2D result;
   result.x = v1.x + v2.y; result.y = v1.y + v2.y;
   return result;
}
```

- Wenn ein Objekt per Parameter übergeben wird, dann wird per Voreinstellung jede einzelne Variablenkomponente (hier x und y) kopiert. Die Parameter *v1* und *v2* leben dann lokal auf dem Stack.
- Objekte können auch zurückgegeben werden. In diesem Fall wird *result* komponentenweise bei der Rückgabe in c kopiert.

vector2d-scale.cpp

```
void scale_vector(Vector2D v, double factor) {
   v.x *= factor; v.y *= factor;
}

int main() {
   Vector2D a; a.x = 1; a.y = 2;
   scale_vector(a, 10);
   print_point(a); std::cout << std::endl;
}
```

- Diese Variante von *scale_vector* ist sinnlos, da nur die lokale Kopie *v* verändert wird, jedoch nicht der Vektor *a* in *main*.

Es gibt hierzu zwei Möglichkeiten:

- Unter expliziter Verwendung von Zeigern (so wie es auch in C üblich war).
- Unter Verwendung von Referenzen.

Dann lässt sich das auch unnötiges Kopieren vermeiden, das bei größeren Objekten (man denke an sehr große Matrizen) recht teuer werden kann.

Parameterübergabe mit einem Zeiger

vector2d-scale2.cpp

```
void scale_vector(Vector2D* vp, double factor) {
   vp->x *= factor; vp->y *= factor;
}

int main() {
   Vector2D a; a.x = 1; a.y = 2;
   scale_vector(&a, 10);
   print_point(a); std::cout << std::endl;
}
```

- Der Adress-Operator „&“ liefert die Adresse eines Objekts – hier mit dem Datentyp *Vector2D∗*.
- Die Funktion *scale_vector* erwartet hier einen Zeiger und ist entsprechend gezwungen, diesen immer zu dereferenzieren.

vector2d-scale3.cpp

```
void scale_vector(Vector2D& vp, double factor) {
   vp.x *= factor; vp.y *= factor;
}

int main() {
   Vector2D a; a.x = 1; a.y = 2;
   scale_vector(a, 10);
   print_point(a); std::cout << std::endl;
}
```

- Referenzen sind implizite Zeiger.
- Entsprechend fällt das explizite Dereferenzieren und die Verwendung des Adress-Operators bei der Übergabe weg.
- Die Umsetzung ist äquivalent zur vorherigen Variante mit expliziten Zeigern.

vector2d-scale3.cpp

```
void print_point(const Vector2D& v) {
   std::cout << "(" << v.x << ", " << v.y << ")";
}
```

- Die Verwendung von Referenzen ist auch dann sinnvoll, wenn das Objekt nicht zu verändern ist, da dann der Kopieraufwand entfällt.
- In diesem Fall ist es sinnvoll, **const** hinzuzufügen. Das sichert zu, dass die so per Referenz übergebene Variable nicht verändert wird.

counter.hpp

```
class Counter {
   public:
      // constructors
      Counter() : counter{0} {
      }
      Counter(int counter) : counter{counter} {
      }
      // accessors
      int get() const {
         return counter;
      }
      // mutators
      int increment() {
         assert(counter < INT_MAX);
         return ++counter;
      }
      int decrement() {
         assert(counter > INT_MIN);
         return --counter;
      }
   private:
      int counter;
};
```

counter.hpp

```
private:
   int counter;
```

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

counter.hpp

```
  Counter() : counter{0} {
  }
  Counter(int counter) : counter{counter} {
  }
```

- Eine Klasse kann beliebig viele Konstruktoren anbieten, solange sie sich in ihrer Signatur unterscheiden.
- Wenn kein Konstruktor angegeben ist, generiert der Übersetzer automatisch einen parameterlosen Konstruktor, der, sofern möglich, sämtliche Datenfelder analog ohne Parameter konstruiert.
- Es ist zulässig, die Parameter der Konstruktoren genauso zu nennen wie die entsprechenden Objektvariablen.

⟨ctor-initializer⟩ ⟶ „:“ ⟨mem-initializer-list⟩

⟨mem-initializer-list⟩ ⟶ ⟨mem-initializer⟩ [„,...“]

⟶ ⟨mem-initializer-list⟩ „,“ ⟨mem-initializer⟩ [„,...“]

⟨mem-initializer⟩ ⟶ ⟨mem-initializer-id⟩ „(“ [⟨expression-list⟩] „)“

⟶ ⟨mem-initializer-id⟩ ⟨braced-init-list⟩

⟨mem-initializer-id⟩ ⟶ ⟨class-or-decltype⟩

⟶ ⟨identifier⟩

⟨braced-init-list⟩ ⟶ „{“ ⟨initializer-list⟩ [„,“] „}“

⟶ „{“ „}“

- Seit C++11 wird die {...}-Notation (⟨braced-init-list⟩) bevorzugt, da sie einen unschönen grammatikalischen Konflikt vermeidet und mehr Möglichkeiten erlaubt.

- Es ist in C++ sehr wichtig, zwischen einer Initialisierung mit Hilfe eines ⟨ctor-initializer⟩ und einer regulären Zuweisung innerhalb des ⟨compound-statement⟩ des Konstruktors zu unterscheiden.
- Bevor das ⟨compound-statement⟩ des Konstruktors ausgeführt wird, müssen alle Teilobjekte konstruiert sein. Wenn die Initialisierung innerhalb des ⟨ctor-initializer⟩ fehlt, dann wird jeweils der parameterlose Konstruktor verwendet.
- Wenn der parameterlose Konstruktor für eines der Teilobjekte nicht zur Verfügung stehen sollte, dann geht es überhaupt nicht ohne die Konstruktion innerhalb des ⟨ctor-initializer⟩.
- Eine implizite automatische Konstruktion in Kombination mit einer späteren Zuweisung kann zu ineffizienteren Code führen. Daher wird grundsätzlich empfohlen, soweit wie möglich alle Teilobjekte innerhalb des ⟨ctor-initializer⟩ zu initialisieren.
- Die Reihenfolge im ⟨ctor-initializer⟩ sollte der der Deklaration der Teilobjekte entsprechen. In letzterer Reihenfolge werden sie ausgeführt.

counter.hpp

```
Counter() : counter{0} {
}
Counter(int counter) : counter{counter} {
}
```

- Es ist zulässig, die Parameter der Konstruktoren genauso zu nennen wie die entsprechenden Objektvariablen.
- Bei der ⟨mem-initializer-id⟩ wird nur unter den zu initialisierenden Namen der Objektvariablen gesucht bzw. dem Namen der eigenen Klasse oder den Namen der Basisklassen.
- In der ⟨expression-list⟩ bzw. der ⟨initializer-list⟩ werden zuerst die Parameternamen durchsucht, bevor die Namen der Objektvariablen in Betracht gezogen werden.

intinit.cpp

```
int main() {
   cout << "Testing..." << endl;
   int i; // undefined
   cout << "i = " << i << endl;
   int j{17}; // well defined
   cout << "j = " << j << endl;
   int k{}; // well defined: 0
   cout << "k = " << k << endl;
}
```

- Die elementare Datentypen bieten ebenfalls parameterlose Konstruktoren an und einen Konstruktor mit einem Parameter des entsprechenden Typs.
- Wenn jedoch kein Konstruktor explizit aufgerufen wird, erfolgt keine Initialisierung.

```
clonmel$ intinit
Testing...
i = 4927
j = 17
k = 0
clonmel$
```

counter.hpp

```
int get() const {
   return counter;
}
```

- Zugriffsmethoden, die den abstrakten Zustand des Objekts nicht verändern dürfen, werden mit **const** ausgezeichnet.
- Nur diese Methoden dürfen aufgerufen werden, wenn das Objekt in einem Kontext nur lesenderweise zur Verfügung steht.
- Mit **mutable** deklarierte Variablen dürfen auch von **const**-Methoden verändert werden. Dies ist sinnvoll etwa zur Vermeidung sich sonst wiederholender Berechnungen und sollte nicht zu außen sichtbaren Veränderungen führen. (Abstrakter vs. konkreter Zustand eines Objekts.)

```
Counter(const Counter& orig) : counter{orig.counter} {
}
```

- Wenn einer der Konstruktoren genau einen Parameter mit einem Referenztyp der eigenen Klasse hat, dann handelt es sich dabei um einen expliziten Kopierkonstruktor.
- Normalerweise wird dieser mit **const** versehen.
- Der Kopierkonstruktor wird dann ggf. implizit verwendet bei der Parameterübergabe, bei **return** und einer Zuweisung.
- Wenn der Kopierkonstruktor nicht explizit deklariert und nicht ausdrücklich unterbunden wird, dann erzeugt der Übersetzer einen, wobei jedes Teilobjekt entsprechend kopierkonstruiert wird. Das funktioniert nur, wenn dies für alle Teilobjekte geht.

- Klassen, die selbst Ressourcen verwalten wie etwa dynamisch angelegte Speicherbereiche, benötigen sehr viel Sorgfalt in C++, damit mit der impliziten Verwendung von Konstruktoren und Methoden keine Probleme entstehen.
- Es ist dabei insbesondere sicherzustellen, dass dynamische Datenstrukturen nur ein einziges Mal freigegeben werden. D.h. das unbemerkte Kopieren von Zeigern ist ein Problem.
- Das folgende Beispiel illustriert dies an einer sehr einfachen Klasse, die ein **int**-Array verwaltet.

array.hpp

```
class Array {
   public:
      Array() : nof_elements(0), ip(nullptr) {}
      Array(unsigned int nof_elements) : nof_elements{nof_elements},
            ip{new int[nof_elements] {}} {
      }
      Array(const Array& other) : nof_elements{other.nof_elements},
            ip{new int[nof_elements]} {
         for (unsigned int i = 0; i < nof_elements; ++i) {
            ip[i] = other.ip[i];
         }
      }
      Array(Array&& other) : nof_elements{other.nof_elements},
            ip{other.ip} {
         other.nof_elements = 0; other.ip = nullptr;
      }
      ~Array() { delete[] ip; }
      Array& operator=(const Array& other) = delete;
      Array& operator=(Array&& other) = delete;
      unsigned int size() const { return nof_elements; }
      int& operator()(unsigned int i) {
         assert(i < nof_elements); return ip[i];
      }
      const int& operator()(unsigned int i) const {
         assert(i < nof_elements); return ip[i];
      }
   private:
      unsigned int nof_elements; int* ip;
};
```

array.hpp

```
Array(unsigned int nof_elements) : nof_elements{nof_elements},
      ip{new int[nof_elements] {}} {
}
```

- Mit dem **new**-Operator kann Speicher dynamisch belegt werden. Neben einem Typnamen kann auch in Array-Notation eine Dimensionierung mit angegeben werden.
- Hier wird ein **int**-Array mit *nof_elements* Elementen angelegt.
- Der **new**-Operator lässt hier auch sogleich die Initialisierung der neuen Speicherfläche hinzu. Da {} hier angegeben ist, wird das Array mit Nullen initialisiert. (Ohne diesen expliziten Hinweis würden die **int** uninitialisiert bleiben.)

array.hpp

```
~Array() {
   delete[] ip;
}
```

- Mit dem Operator **delete**[] kann ein Array wieder freigegeben werden.
- Wenn der Zeiger ein **nullptr** sein sollte, stört das nicht.
- Anders als in Java wird diese Funktion garantiert aufgerufen, wenn die Lebenszeit des Objekts beendet ist.

Der Begriff *Resource Acquisition Is Initialization* (RAII) geht auf Bjarne Stroustrup und Andrew Koenig zurück. Folgende Prinzipien sind damit verknüpft:

- Ressourcen werden mit Objekten fest verknüpft.
- Bei der Initialisierung des Objekts wird die Ressource akquiriert.
- Beim Dekonstruieren des Objekts wird die Ressource freigegeben.

Diese Technik vermeidet Fehler und stellt insbesondere sicher, dass auch im Falle einer Ausnahmenbehandlung (*exception handling*) alles sauber abgebaut und damit freigegeben wird.

Da bei C++ Objekte kopiert und nicht ohne weiteres nur Zeiger einander zugewiesen werden, ist bei Klassen, die mit Ressourcen in Verbindung stehen, Vorsicht geboten:

Wenn immer eine Klasse eine Ressource verwaltet, sind folgende spezielle Methoden immer explizit zu definieren bzw. zu deaktivieren, um die unerwünschte implizite Definition zu unterbinden:

- Kopier-Konstruktor
- Zuweisungs-Operator
- Dekonstruktor

Kopierkonstruktur für ein Array

array.hpp

```
Array(const Array& other) : nof_elements{other.nof_elements},
      ip{new int[nof_elements]} {
   for (unsigned int i = 0; i < nof_elements; ++i) {
      ip[i] = other.ip[i];
   }
}
```

- Der Kopierkonstruktor hat die Aufgabe, die gesamte Datenstruktur zu duplizieren.
- Hierfür ist das Kopieren nur des Zeigers unzureichend. Denn dann würde die Klon-Semantik verlorengehen und wir hätten das Problem, dass das Array beim Abbau mehrfach freigegeben würde. Die vom Übersetzer zur Verfügung stehende voreingestellte Implementierung dieses Konstruktors wäre somit fatal.
- Hier wird wiederum dynamisch Speicher von der gegebenen Größe angelegt und dann mit Hilfe der **for**-Schleife die Elemente einzeln kopiert. (Wie das effizienter geht, kommt später.)

array.hpp

```
Array(Array&& other) : nof_elements{other.nof_elements},
      ip{other.ip} {
   other.nof_elements = 0;
   other.ip = nullptr;
}
```

- Der Verschiebekonstruktor (*move constructor*) wird dann verwendet, wenn das Quellobjekt unmittelbar nach dem Aufruf abgebaut wird. Das ist insbesondere bei temporären Objekten der Fall.
- In diesem Fall können wir die dynamische Datenstruktur einfach übernehmen und müssen dann nur sicherstellen, dass beim Abbau des Quellobjekts keine versehentliche Freigabe der Datenstruktur erfolgt.

array.hpp

```
Array& operator=(const Array& other) = delete;
Array& operator=(Array&& other) = delete;
```

- Der Übersetzer unterstützt auch implizit Zuweisungen. Bei Objekten mit Zeigern ist hier ebenfalls die voreingestellte Implementierung unzureichend.
- Hier wird gezeigt, wie die voreingestellte Implementierung mit Hilfe von = **delete** unterdrückt werden kann, ohne sie durch eine eigene Implementierung zu ersetzen.

array.hpp

```
int& operator()(unsigned int i) {
   assert(i < nof_elements); return ip[i];
}
const int& operator()(unsigned int i) const {
   assert(i < nof_elements); return ip[i];
}
```

- Statt traditioneller *set*- und *get*-Methoden kann auch der direkte Zugriff auf ein ansonsten privates Element gegeben werden, indem eine Referenz zurückgegeben wird.
- Das Resultat kann dann sowohl als *lvalue* (links von einer Zuweisung) als auch als *rvalue* (rechts der Zuweisung) verwendet werden.
- Methoden können auch nach einem Operator benannt werden mit Hilfe des Schlüsselworts **operator**. (Hier ist es der Funktionsoperator ().)

```
Array a{10};
a(1) = 77;
a(2) = a(1) + 10;
```

Copy and Swap

array.hpp

```
friend void swap(Array& a1, Array& a2) {
   std::swap(a1.nof_elements, a2.nof_elements);
   std::swap(a1.ip, a2.ip);
}
Array(const Array& other) :
      nof_elements{other.nof_elements},
      ip{new int[nof_elements]} {
   for (unsigned int i = 0; i < nof_elements; ++i) {
      ip[i] = other.ip[i];
   }
}
Array(Array&& other) : Array() {
   swap(*this, other);
}
Array& operator=(Array other) {
   swap(*this, other);
   return *this;
}
```

- Wenn die Zuweisung unterstützt werden soll, dann dient ein *swap*-Operator der Vereinfachung.

- 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 2.5.1 ist die aktuelle Fassung von Dezember 2017.
- Zu dem Standard gehören mehrere Dokumente, wovon für uns insbesondere die *OMG UML Superstructure* interessant ist: Im Abschnitt 9 werden Klassendiagramme beschrieben, im Abschnitt 17.8 Sequenzdiagramme und im Abschnitt 18 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.

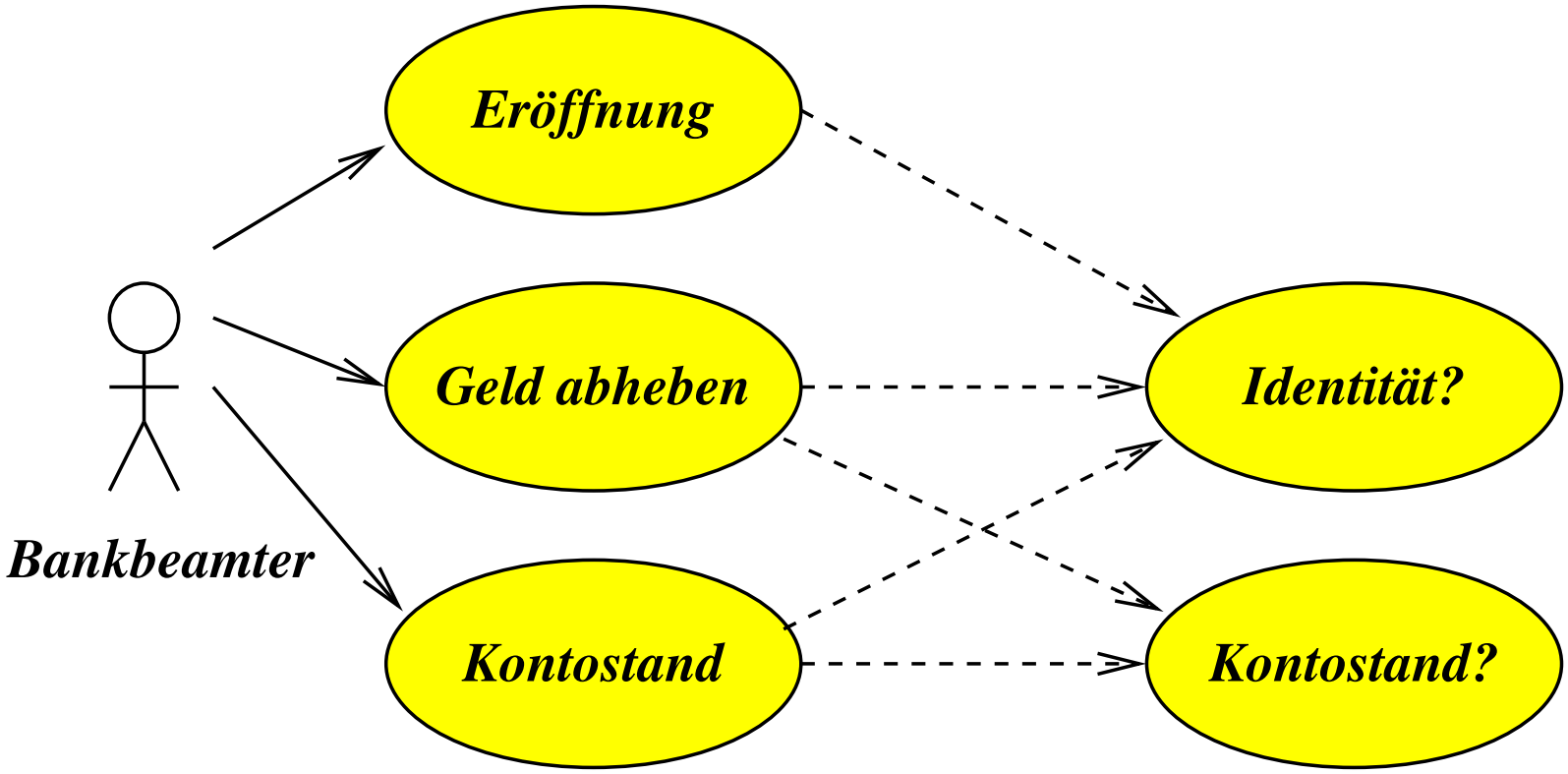
- *Use Cases* dokumentieren während der Analyse die typischen Prozeduren aus der Sicht der aktiven Teilnehmer (Akteure) für ausgewählte Fälle.
- Akteure sind aktive Teilnehmer, die Prozesse in Gang setzen oder Prozesse am Laufen halten.

- 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.

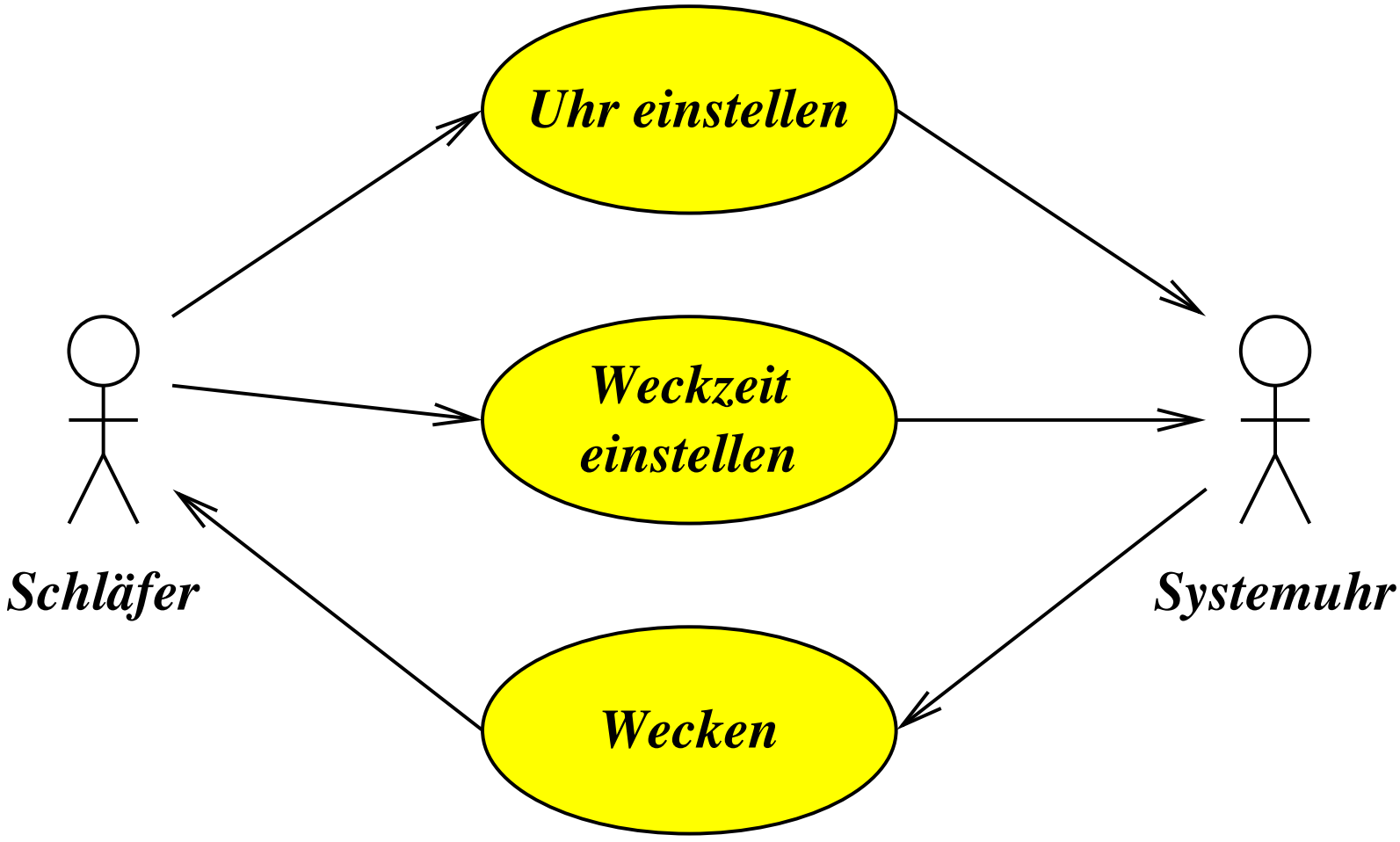
Aus welchen für die Nutzer sichtbaren Schritten bestehen einzelne typische Abläufe bei dem Umgang mit Bankkunden?

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

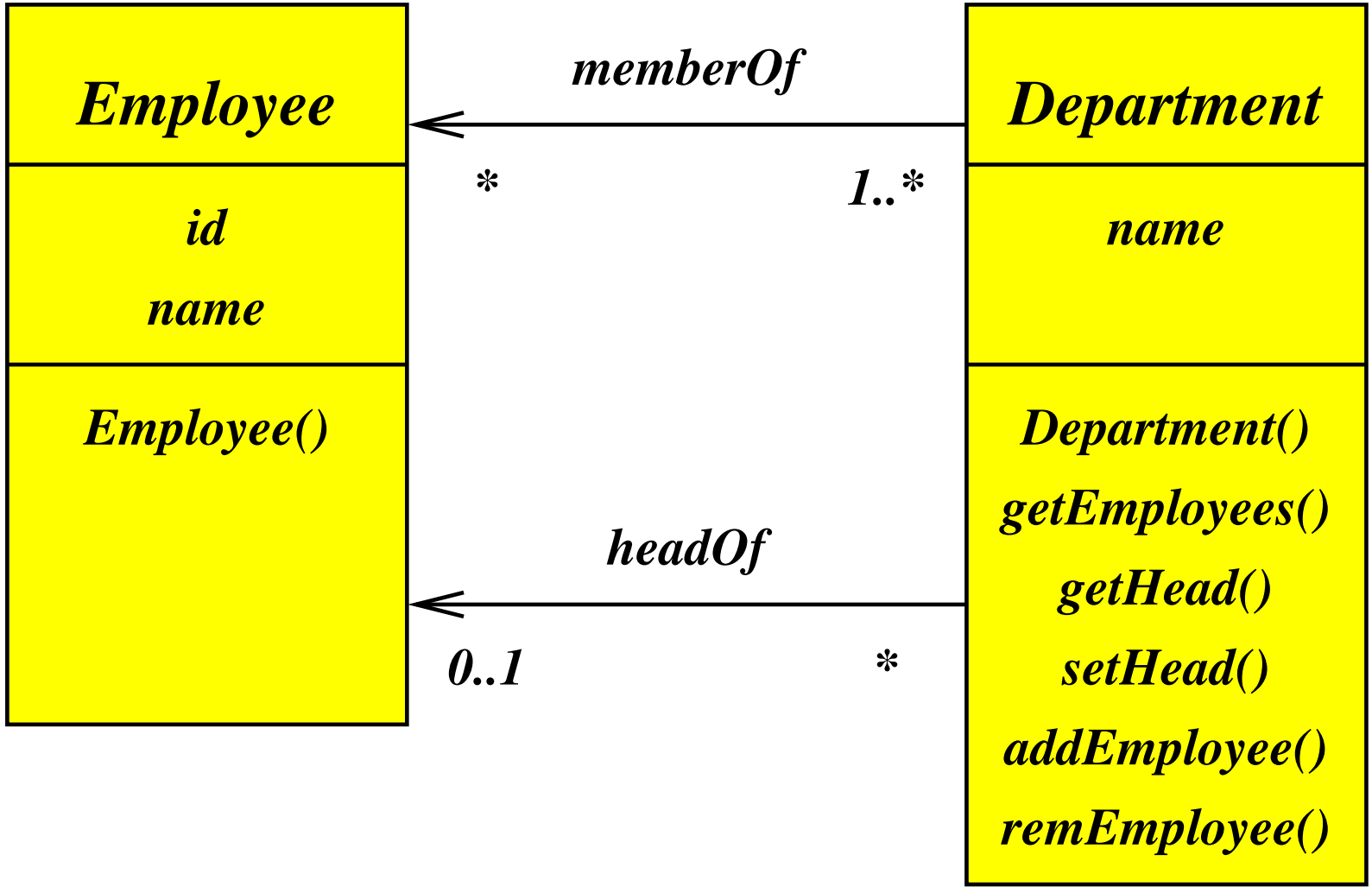
- Hier wurden nur die Aktivitäten aufgeführt, die der Schalterbeamte im Umgang mit dem System ausübt.
- Der Akteur ist hier der Schalterbeamte, weil er in diesen Fällen mit dem System arbeitet. Der Kunde wird nur dann zum Akteur, wenn er beispielsweise am Bankautomaten steht oder über das Internet auf 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

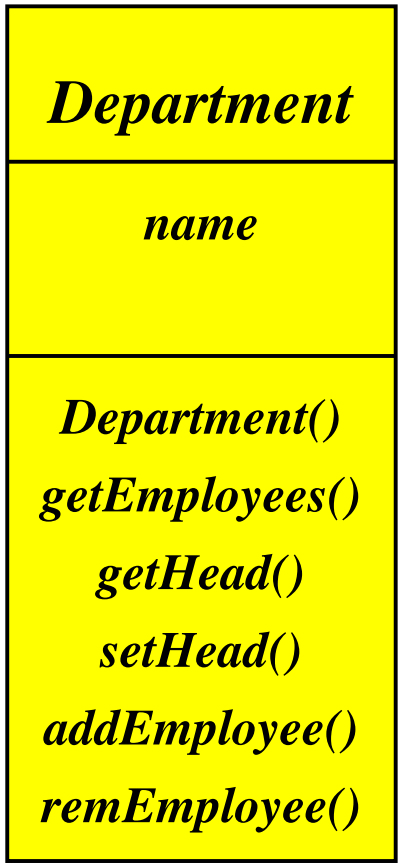
- Eine glatte Linie mit einem Pfeil verbindet einen Akteur mit einem Use-Case. Das bedeutet, dass die mit dem Use-Case verbundene Prozedur von diesem Akteur angestoßen bzw. durchgeführt wird.
- Gestrichelte Linien repräsentieren Beziehungen zwischen mehreren Prozeduren. Damit können Gemeinsamkeiten hervorgehoben 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

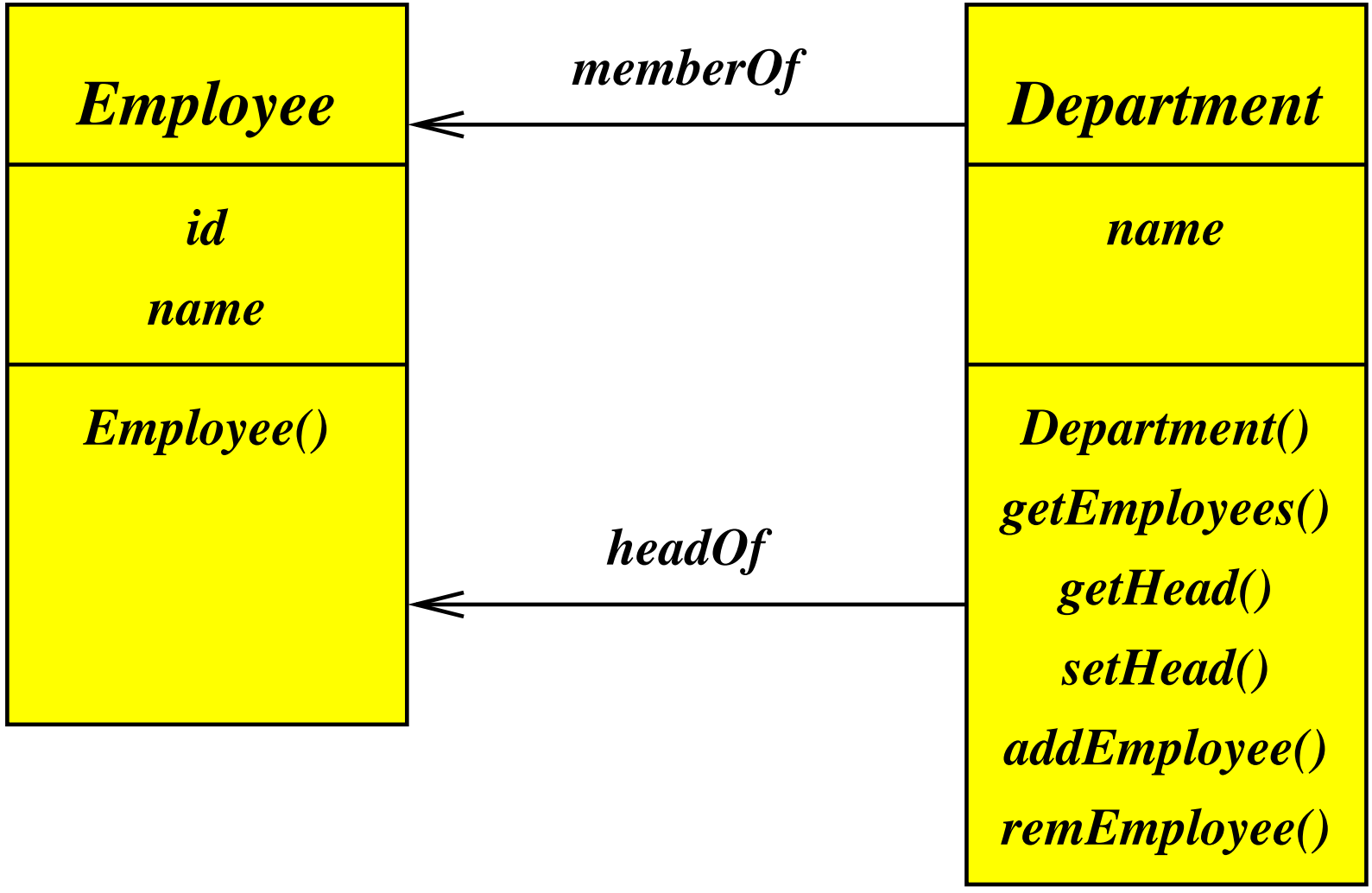
- Es können auch Pfeile von Prozeduren zu Akteuren gehen, wenn sie eine Benachrichtigung repräsentieren, die sofort wahrgenommen wird.
- Ein Wecker hat intern einen Akteur — die Systemuhr. Sie aktualisiert laufend die Zeit und muß natürlich eine Neu-Einstellung der Zeit sofort erfahren.
- Das Auslösen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

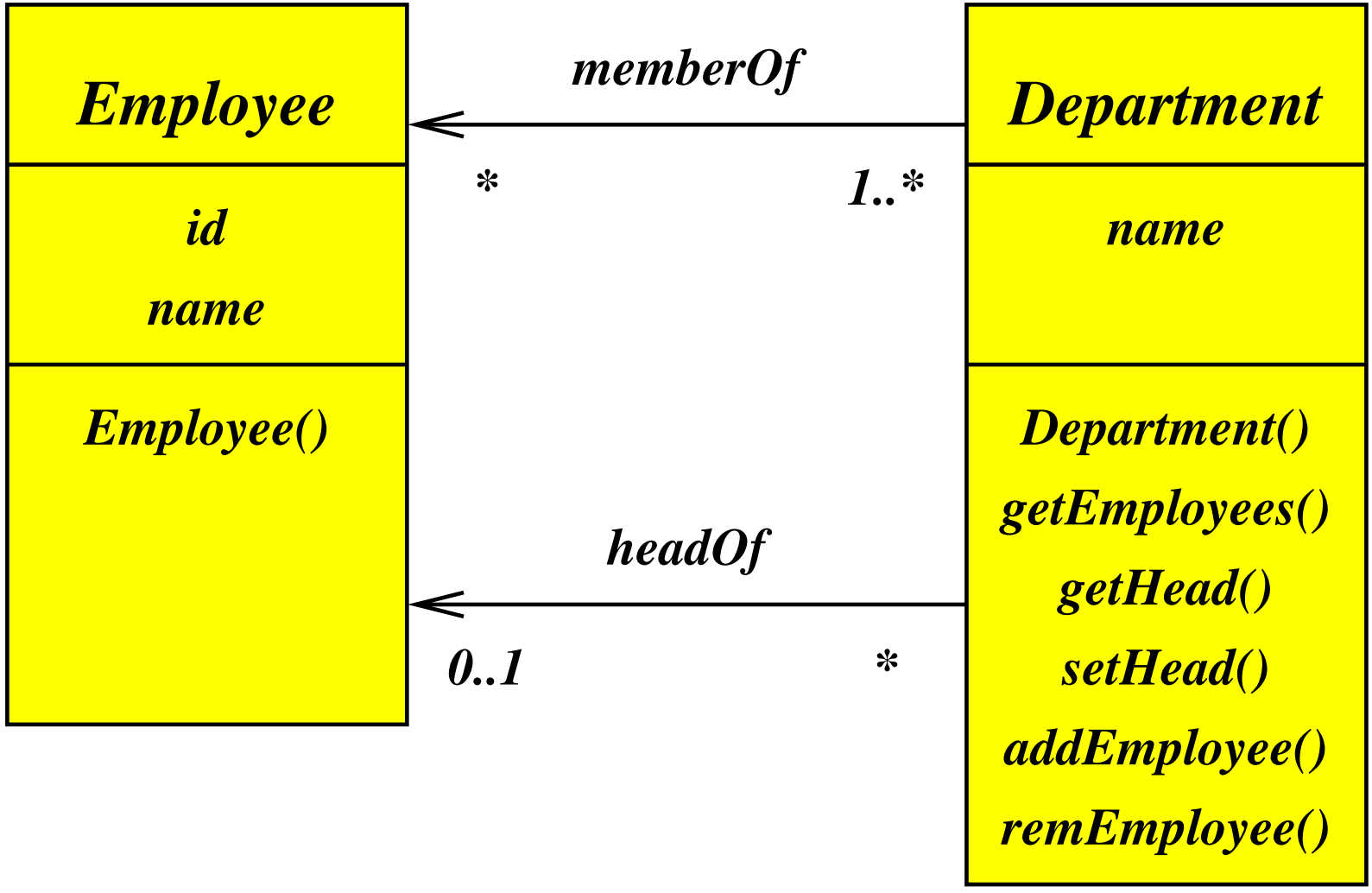


- Die Rechtecke für eine Klasse spezifizieren den Namen der Klasse und die öffentlichen Felder und Methoden. Die erste Methode sollte (sofern vorhanden) der Konstruktor sein.
- Diese Sektionen werden durch horizontale Striche getrennt.
- Bei einem Übersichtsdiagramm ist es auch üblich, nur den Klassennamen 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 sichtbar, wie eine Navigation durch eine Datenstruktur möglich ist. Es lässt sich somit die Frage beantworten, ob beginnend von einem Objekt einer Klasse eine Traverse über weitere Objekte möglich ist 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.

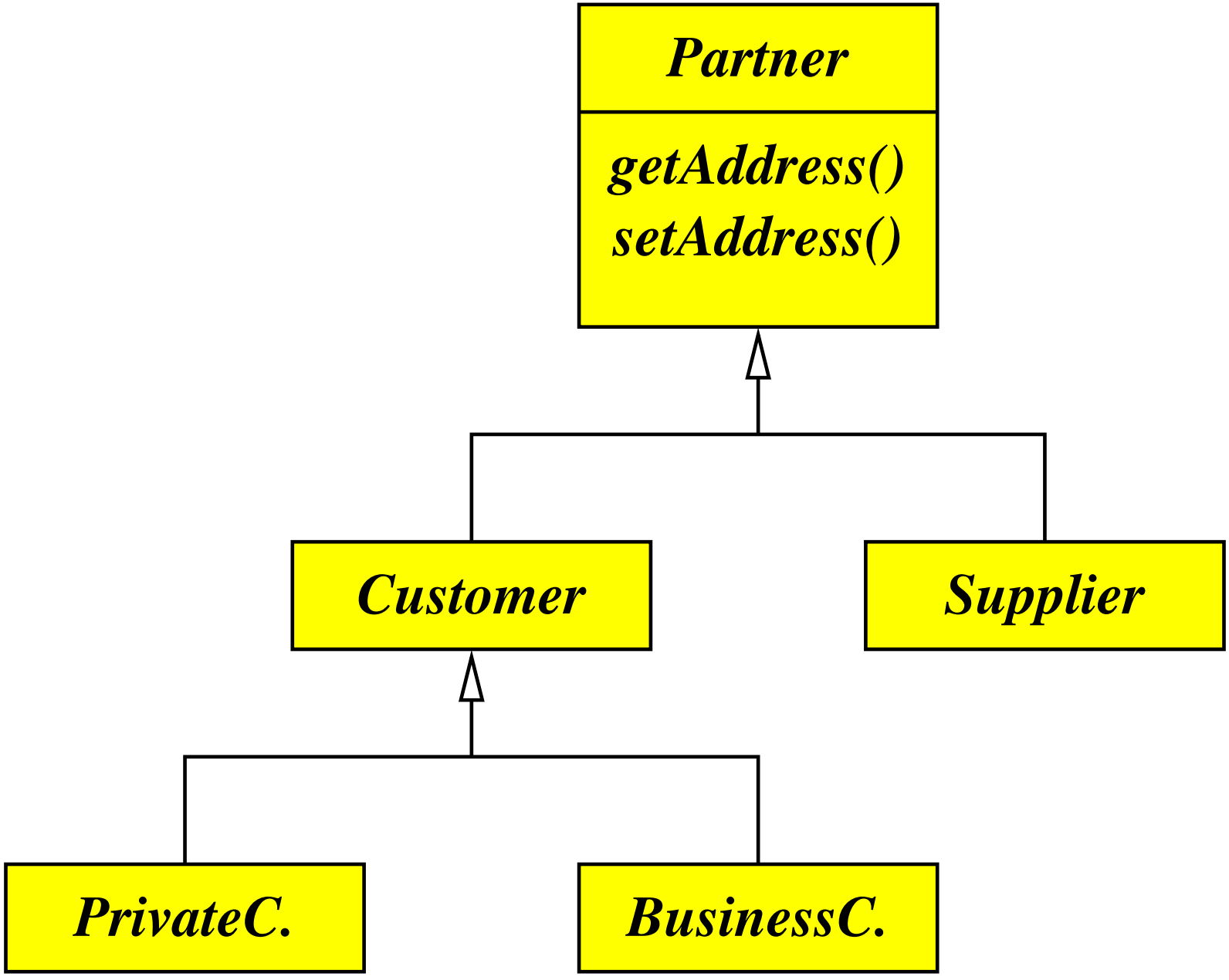

Employee
id
name
Employee()
memberOf
*
1..*
Department
name
Department()
getEmployees()
getHead()
setHead()
addEmployee()
remEmployee()
headOf
0..1
*

- Komplexitätsgrade 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.

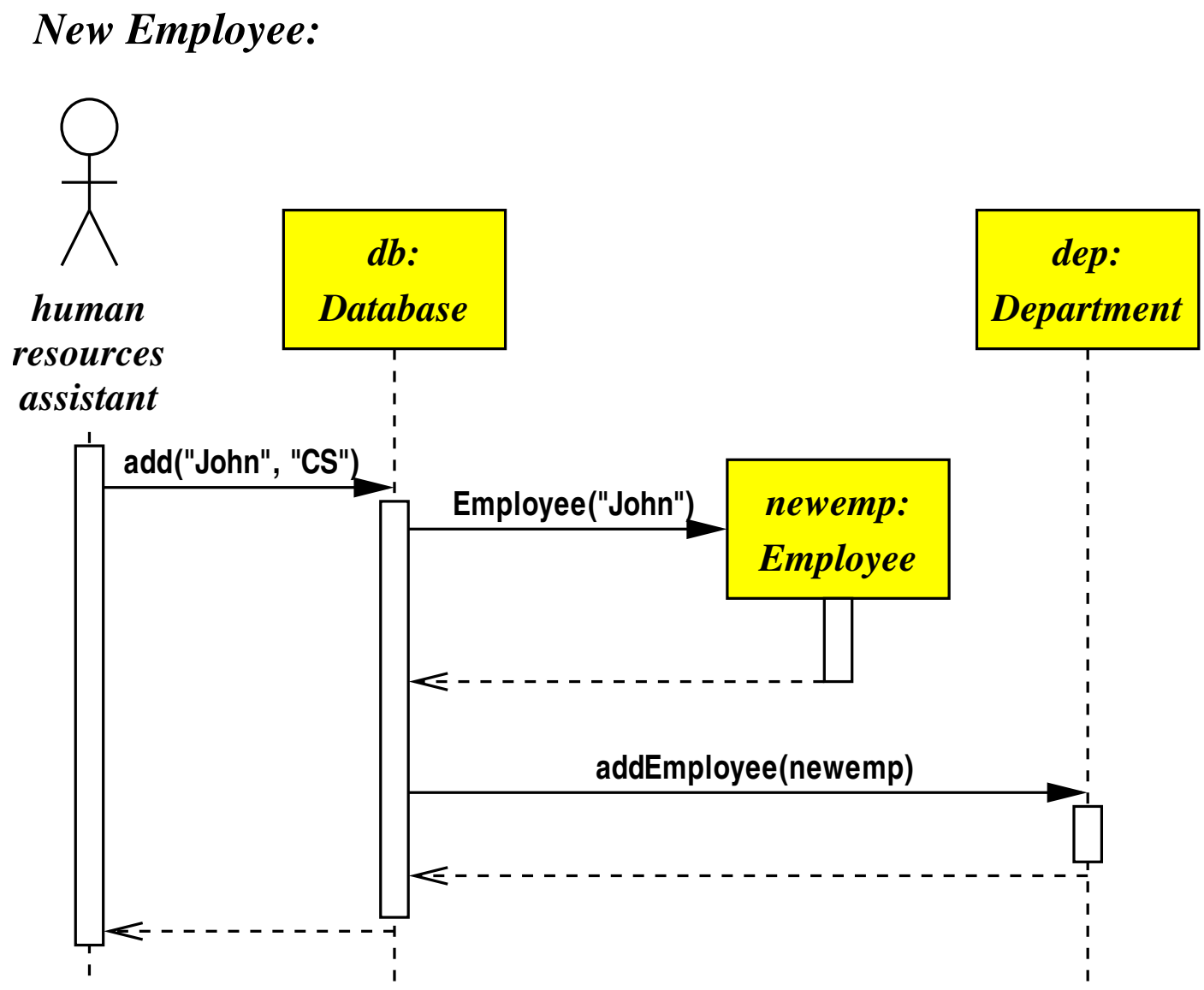
Bei der Implementierung ist der Komplexitätsgrad am Pfeilende relevant:

- 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 **nullptr** 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 **nullptr** ist oder auf einen *Employee* zeigt.
- Für die Beziehung *memberOf* wird hingegen bei der Klasse *Department* eine Liste benötigt.

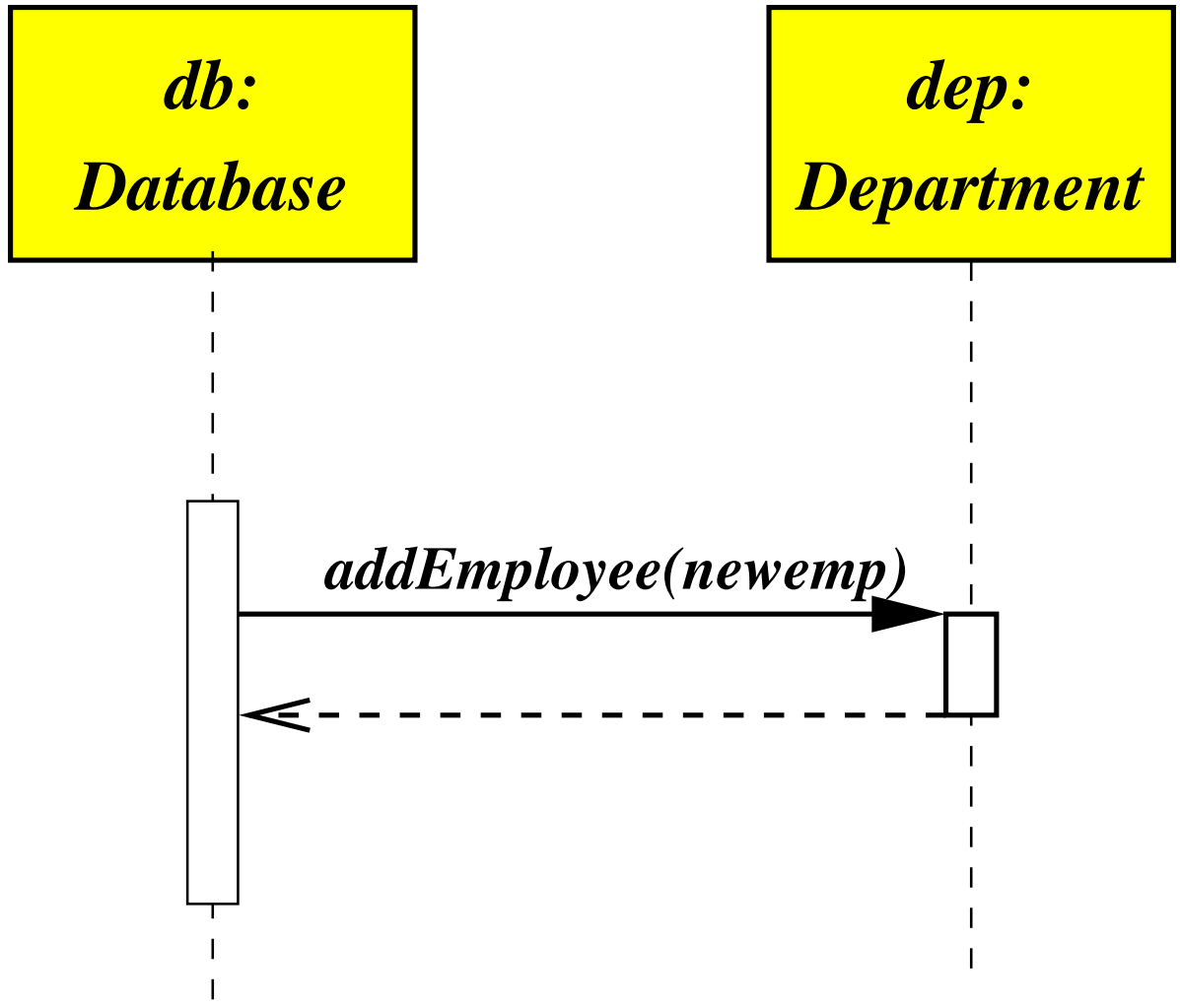
- Auch der Komplexitätsgrad am Anfang des Pfeiles ist relevant, da er angibt, wieviel Verweise insgesamt von Objekten der einen Klasse auf ein einzelnes Objekt der anderen Klasse auftreten können.
- Im Beispiel muß jeder Angestellte in mindestens einer Abteilung aufgeführt sein. Er darf aber auch in 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.

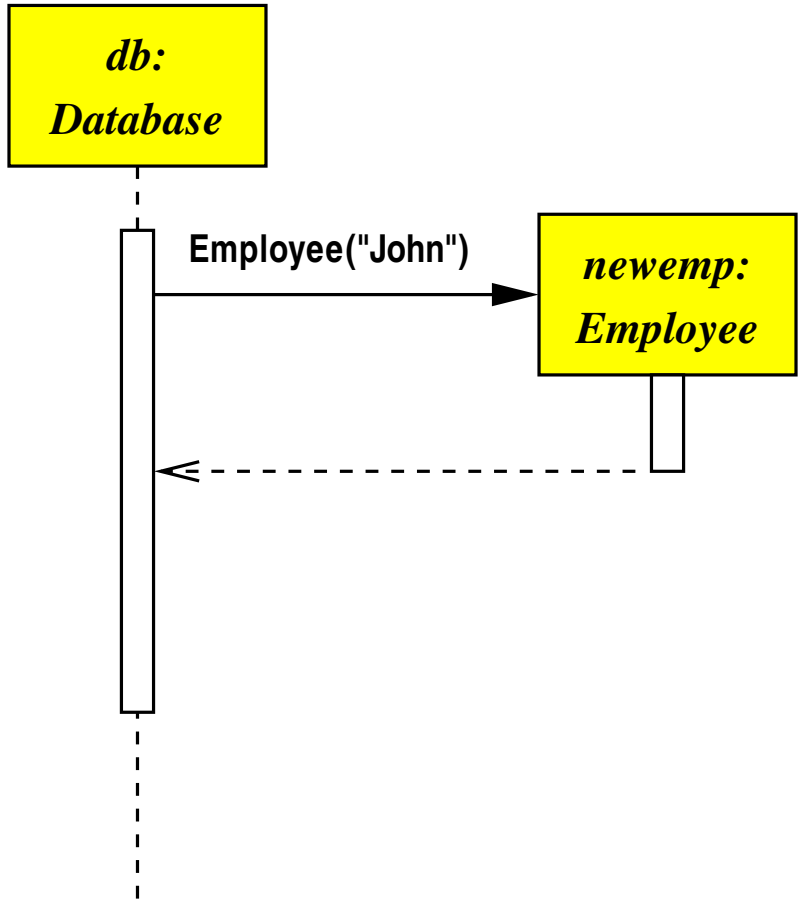

New Employee:
human resources assistant
db: Database
dep: Department
add("John", "CS")
Employee("John")
newemp: Employee
addEmployee(newemp)

- 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 ein langgezogenes ein Rechteck dargestellt.
- Der Methodenaufruf selbst wird durch einen Pfeil dargestellt, der mit dem Aufruf beschriftet wird.
- Die Rückkehr kann entweder weggelassen werden oder sollte durch eine gestrichelte Linie markiert werden.

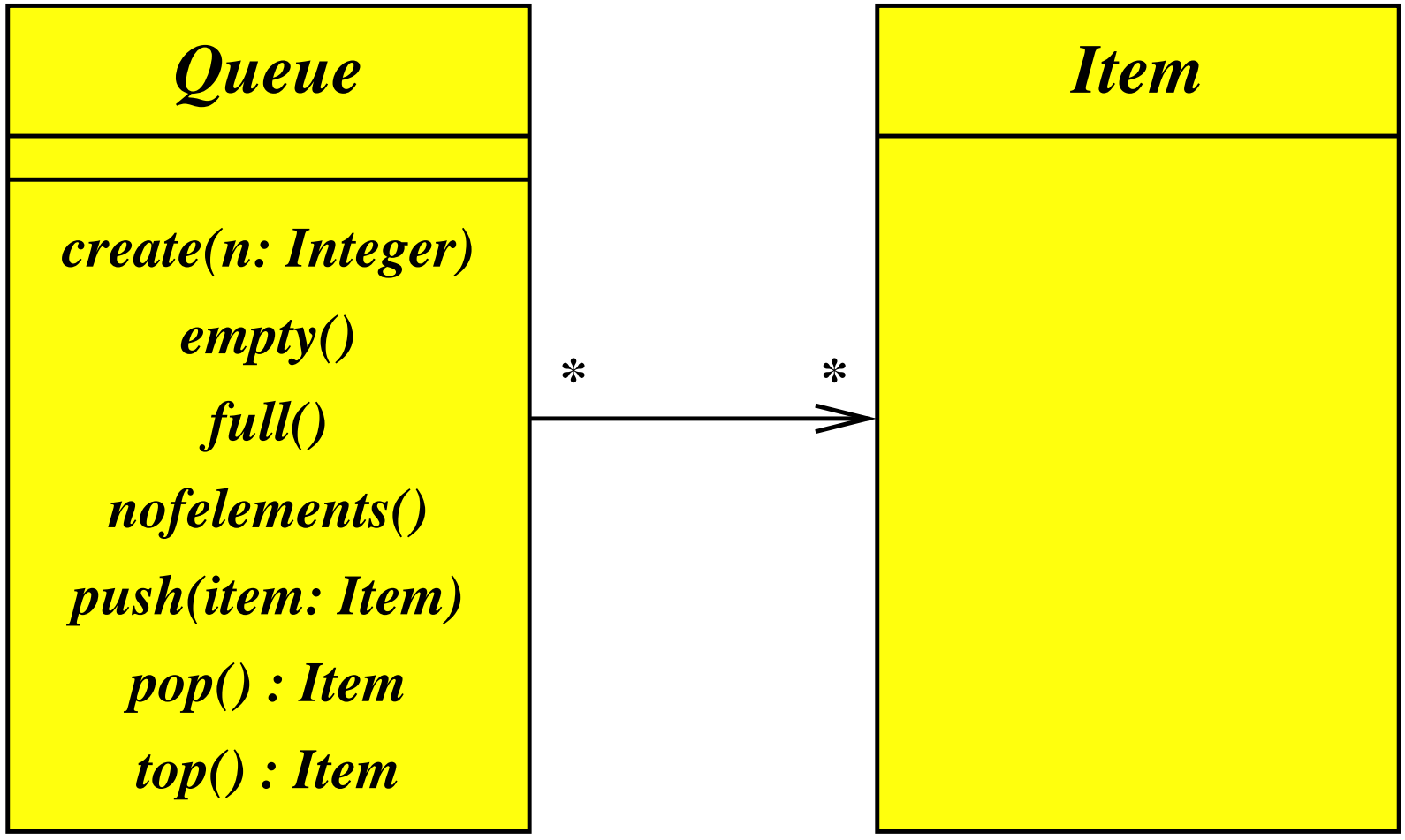
Konstruktoren in einem Sequenz-Diagramm



- 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 der Schnittstellengestaltung 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

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:

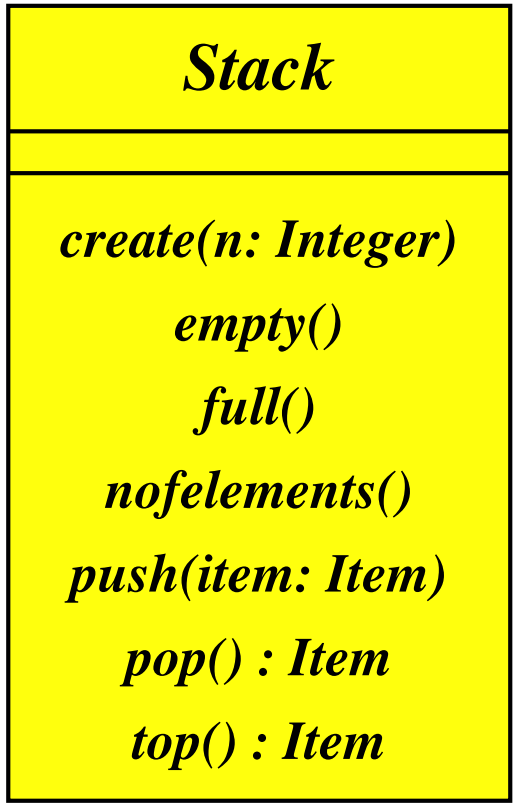
- *nofelements*()== 0 && *empty*()|| *nofelements*()> 0 && !*empty*()
- *empty*()&& !*full*()|| *full*()&& !*empty*()|| !*full*()&& !*empty*()
- *nofelements*()>= *n* || !*full*()

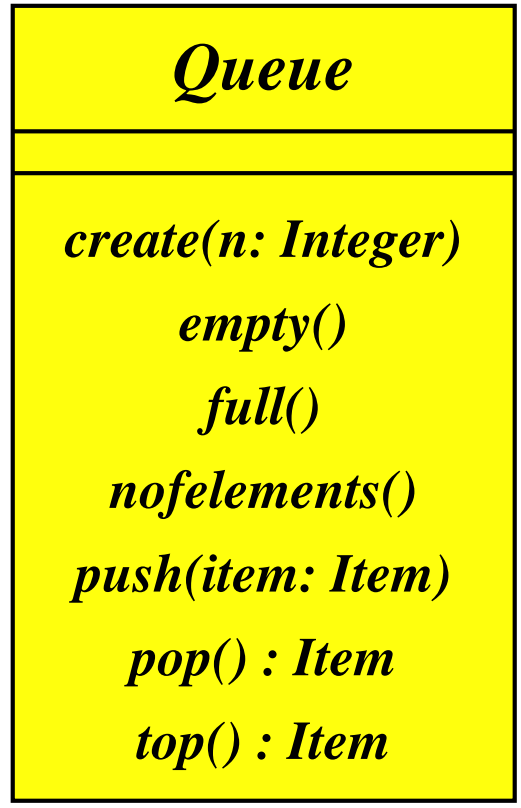
```
void Queue::push(const Item& item) {
   // precondition
   assert(!full());
   // prepare to check postcondition
   int before = nofelements();

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

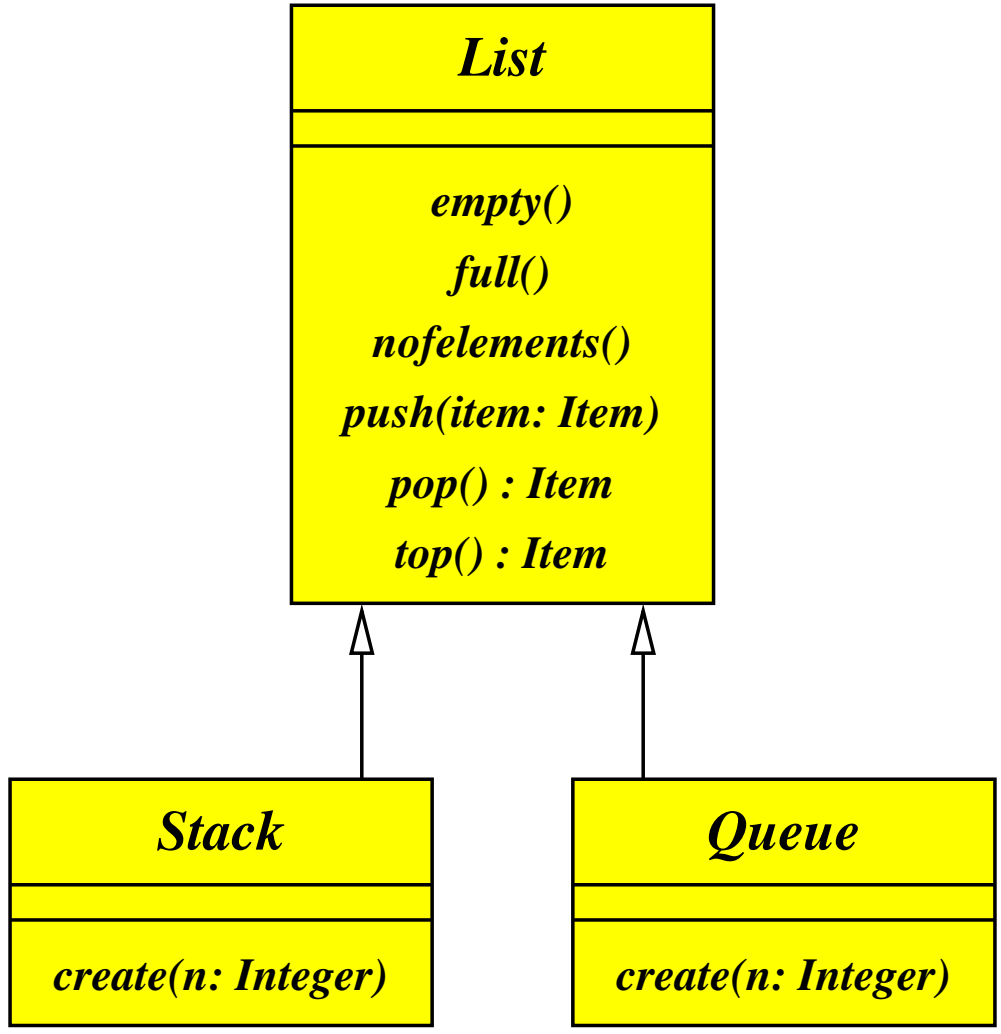
   // checking postcondition
   int after = nofelements();
   assert(!empty() && after == before + 1);
}
```

- 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.

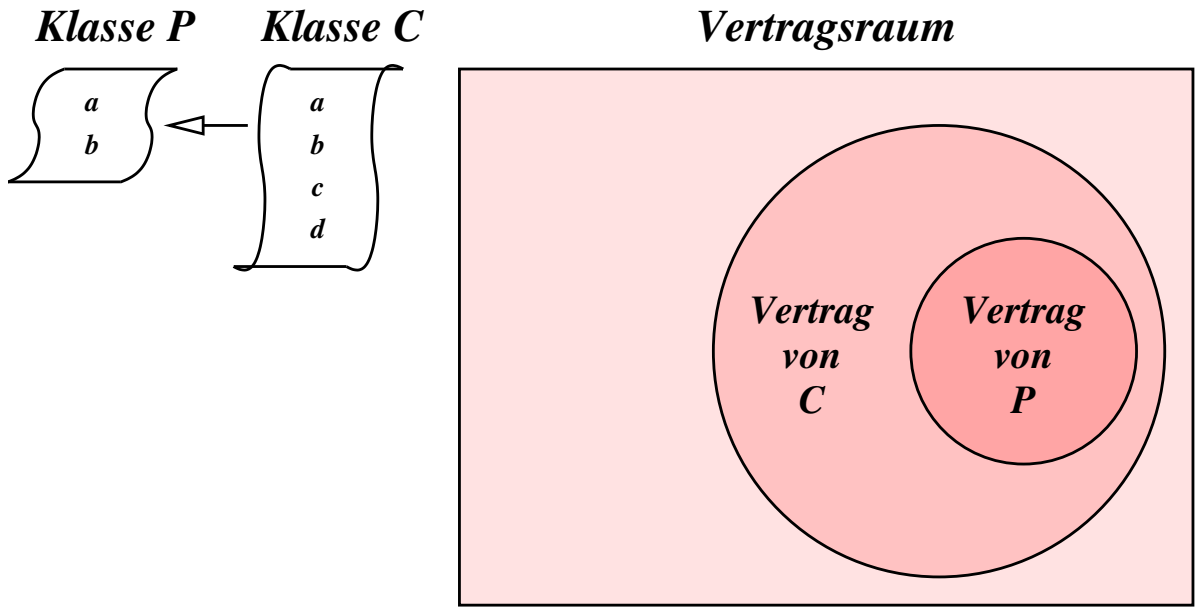

List
empty()
full()
nofelements()
push(item: Item)
pop() : Item
top() : Item
Stack
create(n: Integer)
Queue
create(n: Integer)

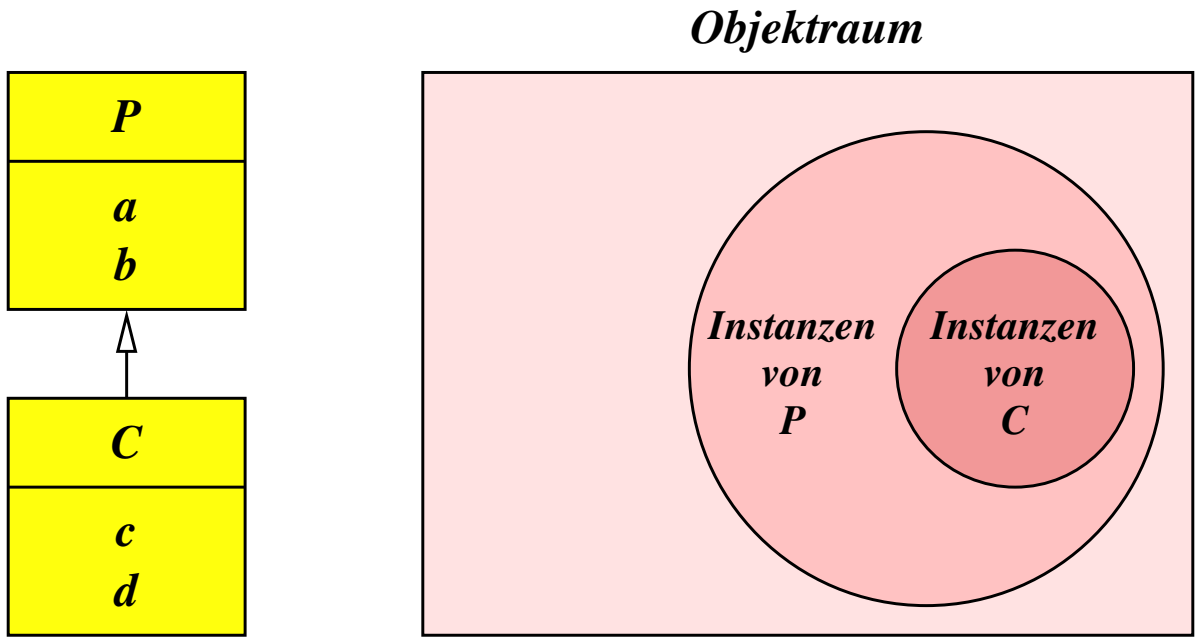
- Einige Klienten benötigen nur allgemeine Listen, die alles akzeptieren, was ihnen gegeben wird. Diese Klienten interessieren sich nicht für die Reihenfolge, in der die Listenelemente später entnommen werden.
- Der Vertrag für *List* spezifiziert, dass *pop()* bei Einhalten der Vorbedingung einer nicht-leeren 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





- 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.

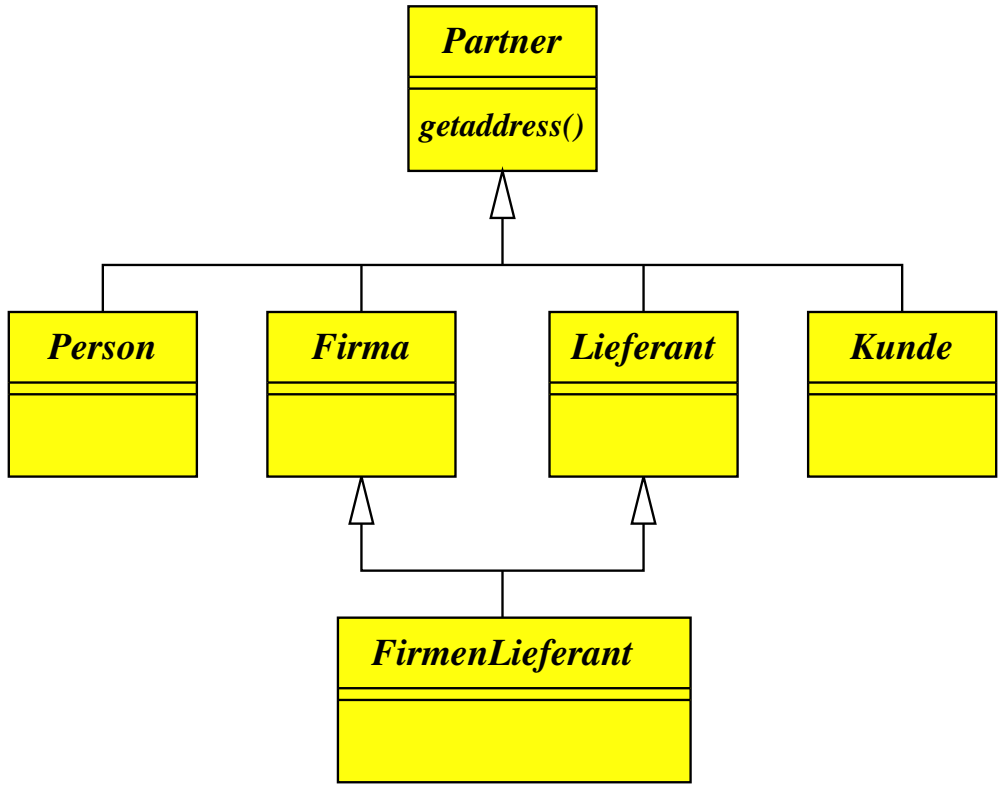
Vererbung sollte genutzt werden, wenn

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

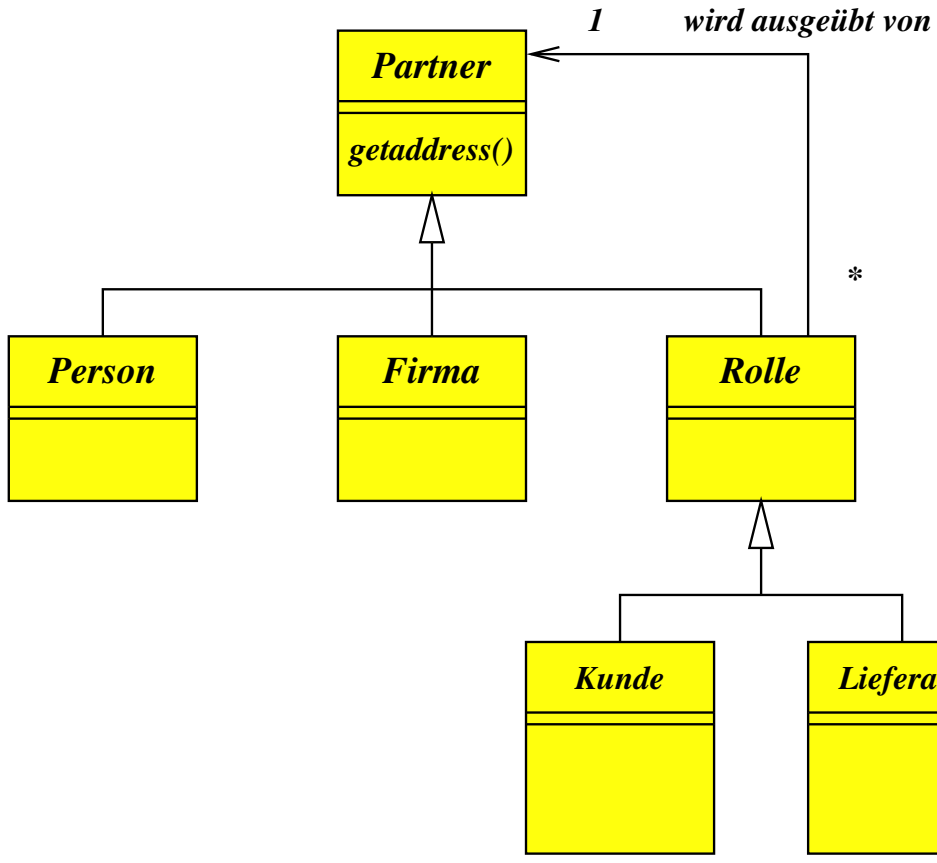
Vererbung sollte **nicht** genutzt werden, um

- 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.



- 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.

- Am Beispiel der *Array*-Klasse wurde bereits das RAII-Prinzip demonstriert.
- Dies lässt sich auch auf rekursive Datenstrukturen übertragen.
- Das folgende Beispiel zeigt dies für einen einfachen, sortierten Binärbaum.
- Der Einfachheit halber bleibt in dem Beispiel die gesamte rekursive Datenstruktur privat, weil das die Verantwortung des Aufräumens erleichtert.
- Ansonsten werden *smart pointers* benötigt, die noch später in der Vorlesung vorgestellt werden.

ListOfFriends.hpp

```
class ListOfFriends {
   public:
      // constructor
      ListOfFriends();
      ListOfFriends(const ListOfFriends& other);
      ListOfFriends(ListOfFriends&& other);
      ~ListOfFriends();
      friend void swap(ListOfFriends& l1, ListOfFriends& l2);

      // assignment
      ListOfFriends& operator=(ListOfFriends other);

      // printing
      void print();

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

   private:
      struct Node* root;
      void addto(Node*& p, Node* newNode);
      void visit(const Node* 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* node);
   ~Node();
}; // struct Node

Node::Node(const Friend& newFriend) :
      left{nullptr}, right{nullptr}, 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* node) :
      left{nullptr}, right{nullptr}, 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.

⟨new-expression⟩ ⟶ [„::“] **new** [⟨new-placement⟩]
⟨new-type-id⟩ [⟨new-initializer⟩]
⟶ [„::“] **new** [⟨new-placement⟩]
„(“ ⟨type-id⟩ „)“ [⟨new-initializer⟩]
⟨new-placement⟩ ⟶ „(“ ⟨expression-list⟩ „)“
⟨new-type-id⟩ ⟶ ⟨type-specifier-seq⟩
⟶ ⟨new-declarator⟩
⟨new-declarator⟩ ⟶ ⟨ptr-operator⟩ [⟨new-declarator⟩]
⟶ ⟨noptr-new-declarator⟩
⟨noptr-new-declarator⟩ ⟶ „[“ ⟨expression⟩ „]“
[⟨attribute-specifier-seq⟩]
⟶ ⟨noptr-new-declarator⟩
„[“ ⟨constant-expression⟩ „]“
[⟨attribute-specifier-seq⟩]
⟨new-initializer⟩ ⟶ „(“ [⟨expression-list⟩] „)“
⟶ ⟨braced-init-list⟩

ListOfFriends.cpp

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

- Hier werden die Felder *left* und *right* zunächst in der Initialisierungssequenz auf **nullptr** 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 **nullptr** ist.
- Zu beachten ist hier, dass der Konstruktor sich selbst rekursiv für die Unterbäume *left* und *right* von *node* aufruft, sofern diese nicht **nullptr** sind.
- Auf diese Weise erhalten wir hier eine tiefe Kopie (*deep copy*), die den gesamten Baum beginnend bei *node* dupliziert.

ListOfFriends.cpp

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

- Wie beim Konstruieren muss hier die Destruktion bei *Node* rekursiv arbeiten.
- **delete** unternimmt nichts, wenn der angegebene Zeiger den Wert **nullptr** hat. Auf diese Weise wird die Rekursion beendet.
- Diese Lösung geht davon aus, dass ein Unterbaum niemals mehrfach referenziert wird.
- Nur durch die Einschränkung der Sichtbarkeit kann dies auch garantiert werden.

⟨delete-expression⟩ $\longrightarrow$ [„::“] **delete** ⟨cast-expression⟩

$\longrightarrow$ [„::“] **delete** „[“ „]“ ⟨cast-expression⟩

ListOfFriends.cpp

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

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

- Der Konstruktor ohne Parameter (*default constructor*) ist trivial: Wir setzen nur *root* auf **nullptr**.
- 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 **nullptr** ist.

ListOfFriends.cpp

```
void swap(ListOfFriends& l1, ListOfFriends& l2) {
   std::swap(l1.root, l2.root);
}

ListOfFriends::ListOfFriends(ListOfFriends&& other) : ListOfFriends() {
   swap(*this, other);
} // ListOfFriends::ListOfFriends
```

- Der Übernahmekonstruktor (*move constructor*) wird ähnlich wie beim kopierenden Konstruktor implizit aufgerufen.
- Entsprechend der *copy-and-swap*-Vorgehensweise wird dieser auf *swap* zurückgeführt.
- Dies stellt sicher, dass das Quellobjekt in einem Zustand hinterlassen wird, das einen Abbau durch den Dekonstruktor zulässt.

ListOfFriends.cpp

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

- Analog delegiert der Destruktor für *ListOfFriends* die Arbeit an den Destruktor für *Node*.

ListOfFriends.cpp

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

- Da der voreingestellte Zuweisungs-Operator nur den Wurzelzeiger kopieren würde, muss einer explizit definiert werden.
- Der Parameter wird hier per *call-by-value* übermittelt. Je nach Kontext kommt hier der normale Kopierkonstruktor oder der Verschiebekonstruktor zum Einsatz. Auf der lokalen Kopie ist dann die *swap*-Operation zulässig.

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* p) {
   if (p) {
      visit(p->left);
      std::cout << p->f.get_name() << ": " <<
         p->f.get_info() << std::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 **nullptr** 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. Bei der Parameterübergabe wird der Parameter kopierkonstruiert von *list1*, wonach per *swap* die Zeiger ausgetauscht werden.

TestFriends.cpp

```
ListOfFriends gen_friends() {
   ListOfFriends list;
   list.add(Friend{"Ralf", "lives in Neu-Ulm"});
   list.add(Friend{"Lisa", "loves her bike"});
   return list;
}

int main() {
   // ...
   ListOfFriends list4;
   list4 = gen_friends();
   // ...
}
```

- Hier wird zunächst der Konstruktor von *ListOfFriends* ohne Parameter aufgerufen (*default constructor*).
- Der Rückgabewert der Funktion *gen_friends* ist ein temporäres Objekt. Wenn dies an *list4* zugewiesen wird, wird der Parameter mit dem Verschiebekonstruktor erzeugt und dann per *swap* die Zeiger ausgetauscht. Die Datenstruktur wird nirgends dupliziert.
- 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 const 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 const 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 const 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 final: public Function {
   public:
      const std::string& get_name() const override;
      double execute(double x) const override;
}; // 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.
- Wenn eine Klasse oder Methode als **final** deklariert wird, dann darf sie nicht mehr abgeleitet bzw. überdefiniert werden.
- Die Auszeichnung einer Methode mit **override** weist darauf hin, dass eine virtuelle Methode überdefiniert wird.
- 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"

static std::string name {"sin"};

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

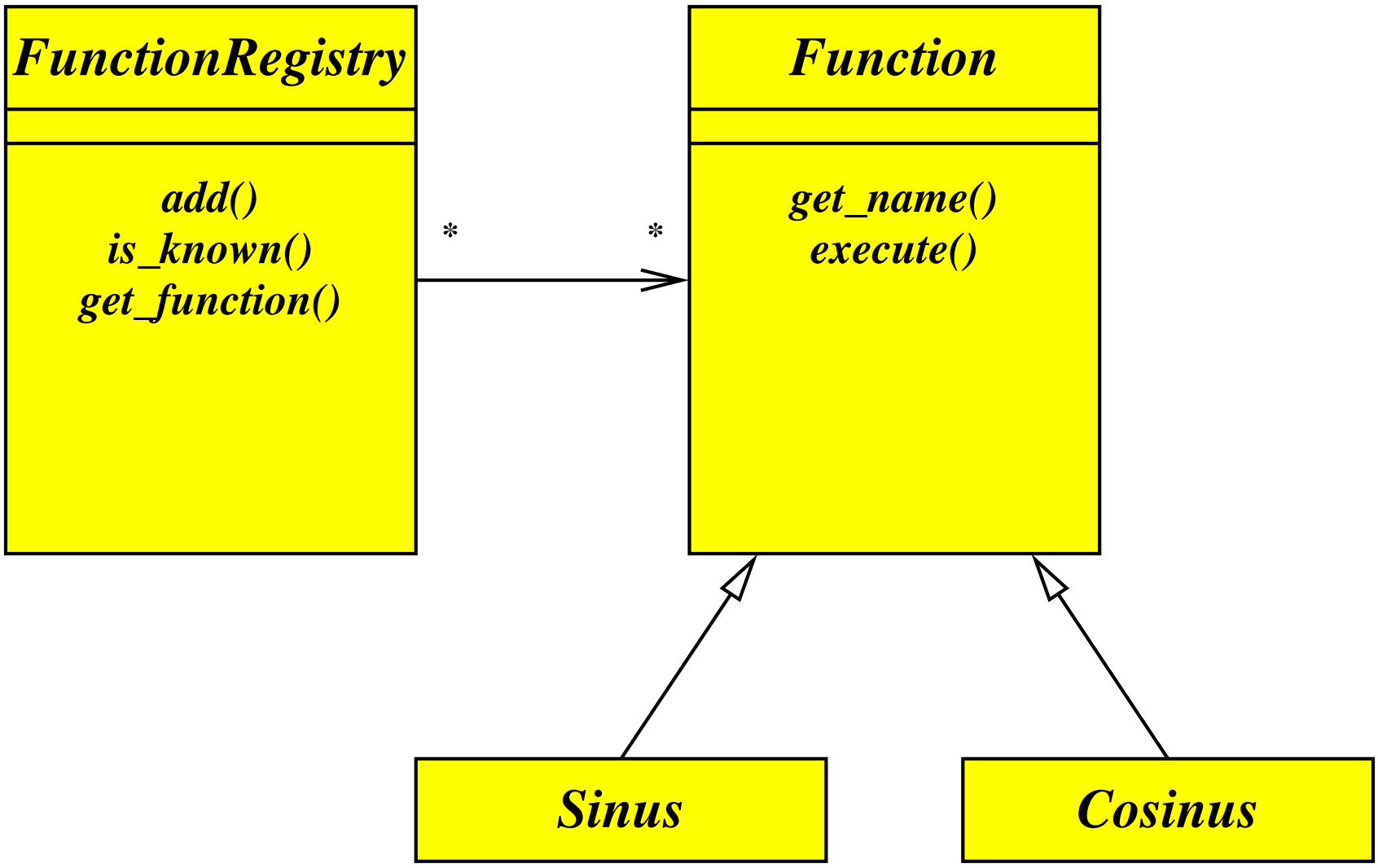
   while (cout << f->get_name() << ": " &&
          cin >> x) {
      cout << f->execute(x) << endl;
   }
   return 0;
} // main
```

- Wenn eine Methode mit dem Schlüsselwort **virtual** versehen ist, dann erfolgt die Bestimmung der zugeordneten Methodenimplementierung *erst zur Laufzeit* in Abhängigkeit des dynamischen Typs, 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 hier der dynamische Typ *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(const std::string& fname) const;
      Function* get_function(const 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.

FunctionRegistry.cpp

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

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

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

Function* FunctionRegistry::get_function(const 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(const 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 unmittelbar das gewünschte Objekt, sondern einen Iterator, der darauf zeigt.
- 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

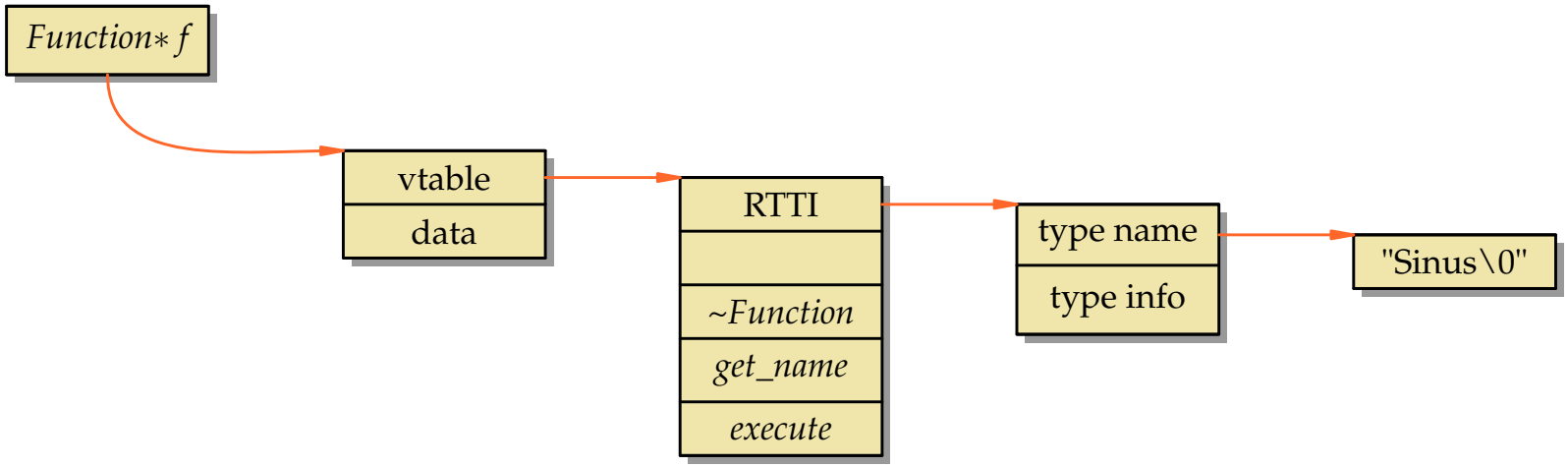
```
#include <iostream>
#include "Sinus.hpp"
#include "Cosinus.hpp"
#include "FunctionRegistry.hpp"

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? 174



- 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. 140 ns.

TestFunctions.s

```
        call    _ZN16FunctionRegistry12get_functionERKNSt7__cxx1112basic_string
        movq    (%rax), %rdx
        movsd   -104(%rbp), %xmm0
        movq    %rax, %rdi
        call    *24(%rdx)
```

- Dies ist der optimierte Assembler-Text für die 64-Bit-x86-Architecktur, der für $f->execute(x)$ nach dem Aufruf von $f = registry.get\_function(fname)$ generiert wird.
- Das Resultat von *get_function* liegt nach dem Aufruf in *%rax*. Dieser Zeiger wird dann dereferenziert und der dann zur Verfügung stehende *vtable*-Zeiger in *%rdx* abgelegt. Das ist der erste Speicherzugriff.
- Beim Aufruf wird der *vtable*-Zeiger dereferenziert (zweiter Speicherzugriff), um den Zeiger auf die aufzurufende Methode zu holen und diesen aufzurufen.

- Dynamischer Polymorphismus verschenkt u.U. Optimierungspotential.
- Wenn eine Klassen-Implementierung bei der Übersetzung des Klienten zugänglich ist (z.B. weil sie sich in der entsprechenden Header-Datei befindet), ergibt sich für den Übersetzer die Möglichkeit, den Methodenaufruf wegzuoptimieren und den ⟨function-body⟩ der Methode direkt an der Stelle des Aufrufs hineinzugenerieren.
- Mit dem Schlüsselwort **inline** kann auch direkt darum gebeten werden.
- All dies entfällt bei dynamischen Polymorphismus, weil erst zur Laufzeit die zugehörige Methode ermittelt wird.

- 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.
- Wenn sich zur Übersetzzeit bereits ermitteln lässt, welche Methoden aufzurufen sind, dann lässt sich das in C++ auf Basis des statischen Polymorphismus besser umsetzen.

```
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 C++14) 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.

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 Parametern abhängen. Überwiegend handelt es sich dabei um Typparameter.
- Sie können nur in instantiierter Form verwendet werden, wenn alle Parameter gegeben und ggf. deklariert sind.
- Unter bestimmten Umständen ist auch eine implizite Festlegung eines Typparameter möglich, wenn sich dieser aus dem Kontext ergibt.
- Generische Module wurden zuerst von CLU und Ada unterstützt (nicht in Kombination mit OO-Techniken) und später in Eiffel, einer statisch getypten OO-Sprache.
- Generische Klassen wurden zunächst primär für Container-Klassen verwendet wie etwa in der STL, der Einsatz von Templates wurde aber zunehmend ausgeweitet, insbesondere nach der Einführung von C++11.

- 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 Parametern) Code zu generieren ist.
- Anders als bei Makros
  - müssen sich generische Module sich an die üblichen Regeln halten (korrekte Syntax, Sichtbarkeit, Typverträglichkeiten),
  - können Syntaxfehler schon vor einer Instantiierung festgestellt werden und es
  - lässt sich die Code-Duplikation im Falle zweier Instanzen mit identischen Parametern vermeiden.

```
class List {
   // ...
   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.

Java hat nach dem Vorbild bereits damals existierender anderer objekt-orientierter Programmiersprachen[1] eine Klasse *Object* eingeführt, von der implizit alle anderen Klassen abgeleitet sind. Das hat Vor- und Nachteile:

- Container können ohne die Techniken generischer Klassen geschrieben werden, indem *Object* als Basisklasse für die enthaltenen Objekte verwendet wird.
- Die Klasse *Object* enthält zahlreiche Methoden wie beispielsweise *toString*, die von den abgeleiteten Klassen überdefiniert werden können.
- Wenn Elemente dem Container entnommen werden, ist immer eine dynamische Typkonvertierung notwendig.
- Elementare Typen wie beispielsweise **int** sind keine Erweiterungen von *Object* und benötigen daher Wrapper-Klassen wie beispielsweise *Integer*, um ganzzahlige Werte einzupacken.

- 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.
- Typparameter 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.

Deklaration eines Templates

$$
\begin{array}{rcl}
\langle\text{template-declaration}\rangle & \longrightarrow & \textbf{template} \\
 & & \text{„<“ } \langle\text{template-parameter-list}\rangle \text{ „>“} \\
 & & \langle\text{declaration}\rangle \\
\langle\text{template-parameter-list}\rangle & \longrightarrow & \langle\text{template-parameter}\rangle \\
 & \longrightarrow & \langle\text{template-parameter-list}\rangle \text{ „,“} \\
 & & \langle\text{template-parameter}\rangle \\
\langle\text{template-parameter}\rangle & \longrightarrow & \langle\text{type-parameter}\rangle \\
 & \longrightarrow & \langle\text{parameter-declaration}\rangle
\end{array}
$$

- Eine reguläre ⟨parameter-declaration⟩ ist nur zulässig für ganzzahlige Datentypen wie etwa **int**, Aufzählungstypen, Zeiger und Referenzen.

⟨type-parameter⟩ ⟶ ⟨type-parameter-key⟩ [„...“] [⟨identifier⟩]
⟶ ⟨type-parameter-key⟩ [⟨identifier⟩]
„=“ ⟨type-id⟩
⟶ **template**
„<“ ⟨template-parameter-list⟩ „>“
⟨type-parameter-key⟩ [„...“] [⟨identifier⟩]
⟶ **template**
„<“ ⟨template-parameter-list⟩ „>“
⟨type-parameter-key⟩ [⟨identifier⟩]
„=“ ⟨id-expression⟩

⟨type-parameter-key⟩ ⟶ **class**
⟶ **typename**

- Das ist die Syntax von C++17. Bei älteren Versionen ist bei Template-Template-Parametern bei ⟨type-parameter-key⟩ immer **class** anzugeben.

sample-lines.cpp

```
#include <cstdlib>
#include <iostream>
#include <string>
#include "reservoir-sampler.hpp"

int main(int argc, char** argv) {
   ReservoirSampler<std::string> r(10);
   std::string line;
   while (std::getline(std::cin, line)) {
      r.add(std::move(line));
   }
   for (std::size_t index = 0; index < r.get_size(); ++index) {
      std::cout << r(index) << std::endl;
   }
}
```

- Diese Anwendung wählt aus den Zeilen der Standardeingabe bis zu 10 Zeilen zufällig aus und gibt sie anschließend aus.
- *ReservoirSampler* ist eine Container-Klasse, die sich bis zu n Objekten merkt entsprechend dem Reservoir-Sampling-Algorithmus (hier $n = 10$).

sample-lines.cpp

```
ReservoirSampler<std::string> r(nof_lines);
std::string line;
while (std::getline(std::cin, line)) {
   r.add(std::move(line));
}
for (std::size_t index = 0; index < r.get_size(); ++index) {
   std::cout << r(index) << std::endl;
}
```

- Mit *ReservoirSampler<std::string>* wird die Template-Klasse *ReservoirSampler* mit *std::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 der auszuwählenden Einträge bestimmt.
- Der ()-Operator wurde hier überladen, um einen Zugriff auf die ausgewählten Elemente zu erlauben.

Eine Container-Klasse als Template

reservoir-sampler.hpp

```
#ifndef RESERVOIR_SAMPLER_HPP
#define RESERVOIR_SAMPLER_HPP

#include <algorithm>
#include <cassert>
#include <cstdlib>
#include <random>
#include <utility>

template<typename T>
class ReservoirSampler {
   public:
      /* ... */
   private:
      std::mt19937 engine;
      std::size_t size;
      std::size_t taken;
      T* data;
};

#endif
```

reservoir-sampler.hpp

```
std::mt19937 engine;
std::size_t size;
std::size_t taken;
T* data;
```

- Um Elemente zufällig aussuchen zu können, verwenden wir ein *std*::*mt19937*-Objekt als Generator für Pseudo-Zufallszahlen.
- *size* gibt die Zahl der gewünschten Elemente an.
- *taken* gibt die Zahl der mit *add* hinzugefügten Elemente an.
- *data* zeigt auf ein Array mit *size* Elementen.
- Bei *data* sind nur $min(size, taken)$ Elemente tatsächlich existent. Wir werden hier noch sehen, wie wir in C++ mit der Situation umgehen können, nur die tatsächlich existierenden Elemente zu konstruieren. Um das zu erreichen, wird das Belegen und Freigeben von Speicher von dem Konstruieren und Abbauen von Objekten völlig getrennt.
- Die Methode *get_size* liefert die Zahl der verfügbaren Elemente, also $min(size, taken)$.

reservoir-sampler.hpp

```
template<typename T>
class ReservoirSampler {
   public:
      ReservoirSampler(std::size_t size) :
         engine(std::random_device()()),
         size(size), taken(0),
         /* allocate raw memory without constructing anything */
         data(static_cast<T*>(operator new[](sizeof(T) * size))) {
      }
      /* ... */

   private:
      /* ... */
};
```

- Das ist der „normale“ Konstruktor, mit dem ein Reservoir einer gegebenen Größe angelegt werden kann.
- Um das Konstruieren noch nicht vorhandener Elemente des Typs T zu vermeiden, legen wir nur den Speicher an, ohne eines der Objekte zu konstruieren. Dies geht mit der expliziten Verwendung von **operator new**[], wobei dann die Größe in Bytes anzugeben ist.

reservoir-sampler.hpp

```
ReservoirSampler(const ReservoirSampler& other) :
      engine(std::random_device()()),
      size(other.size), taken(other.taken),
      /* allocate raw memory without constructing anything */
      data(static_cast<T*>(operator new[](sizeof(T) * size))) {
   for (std::size_t index = 0; index < other.get_size(); ++index) {
      /* copy-construct already constructed elements of other */
      new (data + index) T(other.data[index]);
   }
}
```

- Der Kopierkonstruktor konstruiert nur die n Elemente, die bei *other* bereits existieren.
- Die Methode *get_size* liefert uns n, d.h. die Zahl existierender Elemente.
- Das Konstruieren auf bereits vorhandenem Speicher geht mit **new**, wenn vor dem Datentyp in Klammern die Adresse angegeben wird (*placement*).

reservoir-sampler.hpp

```
~ReservoirSampler() {
   /* as we allocated raw memory we need to deconstruct
      all elements ourselves */
   for (std::size_t index = 0; index < get_size(); ++index) {
      data[index].~T();
   }
   /* release raw memory */
   operator delete[](data);
}
```

- Da das Belegen von Speicher und das Konstruieren getrennt erfolgte, müssen wir beim Abbau auch beides getrennt vornehmen.
- Der *destructor* wird hier für die existierenden Elemente explizit aufgerufen.
- Mit **operator delete**[](*data*) wird dann nur der Speicher freigegeben.

reservoir-sampler.hpp

```
friend void swap(ReservoirSampler& rs1, ReservoirSampler& rs2) {
   /* there is no need to swap the engines */
   std::swap(rs1.size, rs2.size);
   std::swap(rs1.taken, rs2.taken);
   std::swap(rs1.data, rs2.data);
}
```

- Die *swap*-Funktion wird wie gewohnt implementiert – wir verzichten hier aus pragmatischen Gründen auf das Vertauschen der Pseudo-Zufallsgeneratoren.

reservoir-sampler.hpp

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

- Entsprechend dem *copy and swap idiom* lassen sich der *move constructor* und der Zuweisungsoperator wie gewohnt leicht implementieren.

reservoir-sampler.hpp

```
void add(T value) {
   if (taken < size) {
      /* move-construct new element in reservoir */
      new (data + taken) T(std::move(value));
   } else {
      std::size_t select = std::uniform_int_distribution<std::size_t>
         (0, taken)(engine);
      if (select < size) {
         using std::swap;
         /* use argument-dependent lookup (ADL) */
         swap(value, data[select]);
      }
   }
   ++taken;
}
```

- Wenn *taken* $<$ *size*, dann ist ein weiteres Element zu konstruieren. Das erfolgt hier wieder mit **new** unter Verwendung eines *placement* auf *data* + *taken*.
- Da die lokale Variable *value* danach nicht mehr benötigt wird, können wir hier *std*::*move*(*value*) verwenden, so dass ggf. der *move constructor* zum Einsatz kommt.

reservoir-sampler.hpp

```
if (select < size) {
   using std::swap;
   /* use argument-dependent lookup (ADL) */
   swap(value, data[select]);
}
```

- Wenn ein bereits existierendes Element auszutauschen ist, erledigen wir das mit *swap*.
- Wir benutzen hier *swap* ohne Qualifikation, damit ggf. die *swap*-Funktion gefunden wird, die im Namensraum von T liegt.
- Mit **using** *std*::*swap* wird sichergestellt, dass notfalls *std*::*swap* verwendet wird, wenn keine passendere *swap*-Funktion vorliegt.
- Da die Suche nach dem passenden *swap* den Datentyp der Argumente berücksichtigt, wird dies als *argument-dependent lookup* (ADL) bezeichnet.

reservoir-sampler.hpp

```
std::size_t get_taken() const {
   return taken;
}
std::size_t get_size() const {
   return std::min(size, taken);
}
const T& operator()(std::size_t index) const {
   assert(index < get_size());
   return data[index];
}
```

- Die verbleibenden Funktionen sidn trivial zu implementieren.
- Die Minimumfunktion *std*::*min* wird von **#include** <algorithm> geliefert und akzeptiert beliebig viele Argumente.

- 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 hat.

- Folgende Abhängigkeiten sind zu erfüllen für den Typ-Parameter T der Template-Klasse *ReservoirSample*:
  - Es wird wird ein Kopierkonstruktor für T benötigt. Dieser ist zwingend für den Kopierkonstruktor von *ReservoirSample* notwendig. In den anderen Fällen kann auch alternativ ein *move constructor* bzw. *swap* zum Zuge kommen, falls vorhanden.
  - Destruktor: Für den Abbau der Objekte (entweder beim Austauschen oder beim Abbau des gesamten Reservoirs) wird dieser benötigt.

template-failure.cpp

```
#include "reservoir-sampler.hpp"

struct Test {
   int i;
   Test(int i) : i(i) {}
   Test(const Test& other) = delete;
};

int main() {
   ReservoirSampler<Test> r(5);
   Test val{1};
   r.add(val);
}
```

- Hier wurde der Kopierkonstruktur explizit unterbunden, womit implizit auch der *move constructor* wegfällt.
- Damit wird eine der Template-Abhängigkeiten von *ReservoirSample* nicht erfüllt.

```
theon$ g++ -o template-failure template-failure.cpp 2>&1 | head -7
template-failure.cpp: In function 'int main()':
template-failure.cpp:12:13: error: use of deleted function 'Test::Test(const Test&)'
    r.add(val);
             ^
template-failure.cpp:6:4: note: declared here
    Test(const Test& other) = delete;
    ^~~~
theon$
```

- Die Fehlermeldungen können bei nicht erfüllten Template-Abhängigkeiten ungemein umfangreich werden.
- Es lohnt sich hier aber ein Blick auf die ersten Meldungen, die das Problem recht treffend beschreiben.

- Bei der Übersetzung von Templates gibt es ein schwerwiegendes Problem:
  - Dort, wo die Methoden einer Template-Klasse implementiert sind, ist nicht bekannt, welche Instanzen benötigt werden.
  - Dort, wo das Template instantiiert wird sind die Methodenimplementierungen der Template-Klasse unbekannt, wenn diese nicht in der entsprechenden Header-Datei stehen.
- 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.
- 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 ReservoirSampling<std::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 ReservoirSampling<std::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 hierfür.
- Diese Technik wird daher typischerweise nur in isolierten Fällen benutzt.

- Polymorphismus bedeutet, dass die jeweilige Methode bzw. Funktion in Abhängigkeit der Parametertypen (u.a. auch nur von einem einzigen Parametertyp) ausgewählt wird.
- Die Auswahl kann statisch (also zur Übersetzzeit) oder dynamisch (zur Laufzeit) erfolgen.
- Dynamischer Polymorphismus wird grundsätzlich von allen objekt-orientierten Programmiersprachen unterstützt.
- In einigen Fällen lässt sich die Auswahl der Methode oder Funktion auch im Rahmen der Optimierung zur Übersetzzeit treffen.
- Bei statischem Polymorphismus wird der Übersetzer gezwungen, die Auswahl zur Übersetzzeit durchzuführen.

- Für C++ werden seit einiger Zeit auch Optimierer angeboten, die beim Zusammenbau des Programms aktiv werden und Optimierungen über die einzelnen Übersetzungseinheiten hinweg vornehmen können.
- Bei neueren GCC-Versionen wird dies durch die Option *-flto* möglich (LTO = *link time optimization*). Die Option muss zur Übersetz- und Zusammenbauzeit angegeben werden.
- Zur Übersetzzeit wird dann die internen Datenstrukturen mit in der Ausgabe abgelegt, die dann zur Zusammenbauzeit ausgewertet werden kann.
- Dies eröffnet die Möglichkeit, dynamischen Polymorphismus durch statischen Polymorphismus zu ersetzen, wenn feststeht, dass bei dem Aufruf einer virtuellen Methode an einer Stelle immer die gleiche Implementierung aufgerufen wird.
- Seit dem GCC 5 konnten mit dieser Optimierung 50% der virtuellen Methodenaufrufe bei Firefox durch statische Aufrufe ersetzt werden.

- Neben der Frage, ob die Entscheidung zur Übersetzzeit oder Laufzeit fällt, 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 oder Eiffel)	virtuelle Methoden in C++
unbeschränkt	Templates in C++	(z.B. in Smalltalk oder Perl)

- Mit den *concepts* gibt es Bestrebungen, auch in C++ die Möglichkeit von Beschränkungen einzuführen. In C++17 wurden *concepts* noch nicht integriert, für C++20 sind sie aber geplant.
- Der *g++* unterstützt eine experimentelle Fassung der *concepts*, die über die Option „-fconcepts“ aktiviert werden kann.

Statischer vs. dynamischer Polymorphismus in C++ 222

Vorteile dynamischen Polymorphismus in C++:

- Unterstützung heterogener Datenstrukturen, etwa einer Liste von Widgets oder graphischer Objekte.
- Dynamisches Nachladen unbekannter Implementierungen ist möglich.
- 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) { std::srand(seedval); }
      long next() { return std::rand(); }
};

class Rand48 {
   public:
      /* note srand48 & lrand48 are part of the
         POSIX standard but neither of the C or
         C++ standard and thereby not in std:: */
      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*.

Beispiel für statischen Polymorphismus


```
template<typename Rand>
unsigned int test_sequence(Rand& rg) {
   constexpr unsigned int N = 64;
   unsigned int hits[N][N][N] = {{{0}}};
   rg.seed(std::random_device()());
   unsigned int r1 = rg.next() / N % N;
   unsigned int r2 = rg.next() / N % N;
   unsigned int max = 0;
   for (unsigned int i = 0; i < N*N*N*N; ++i) {
      unsigned int r3 = rg.next() / N % N;
      unsigned 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;
   std::cout << "result of StdRand: "
      << test_sequence(stdrand) << std::endl;
   std::cout << "result of Rand48: "
      << test_sequence(rand48) << std::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;
}
```

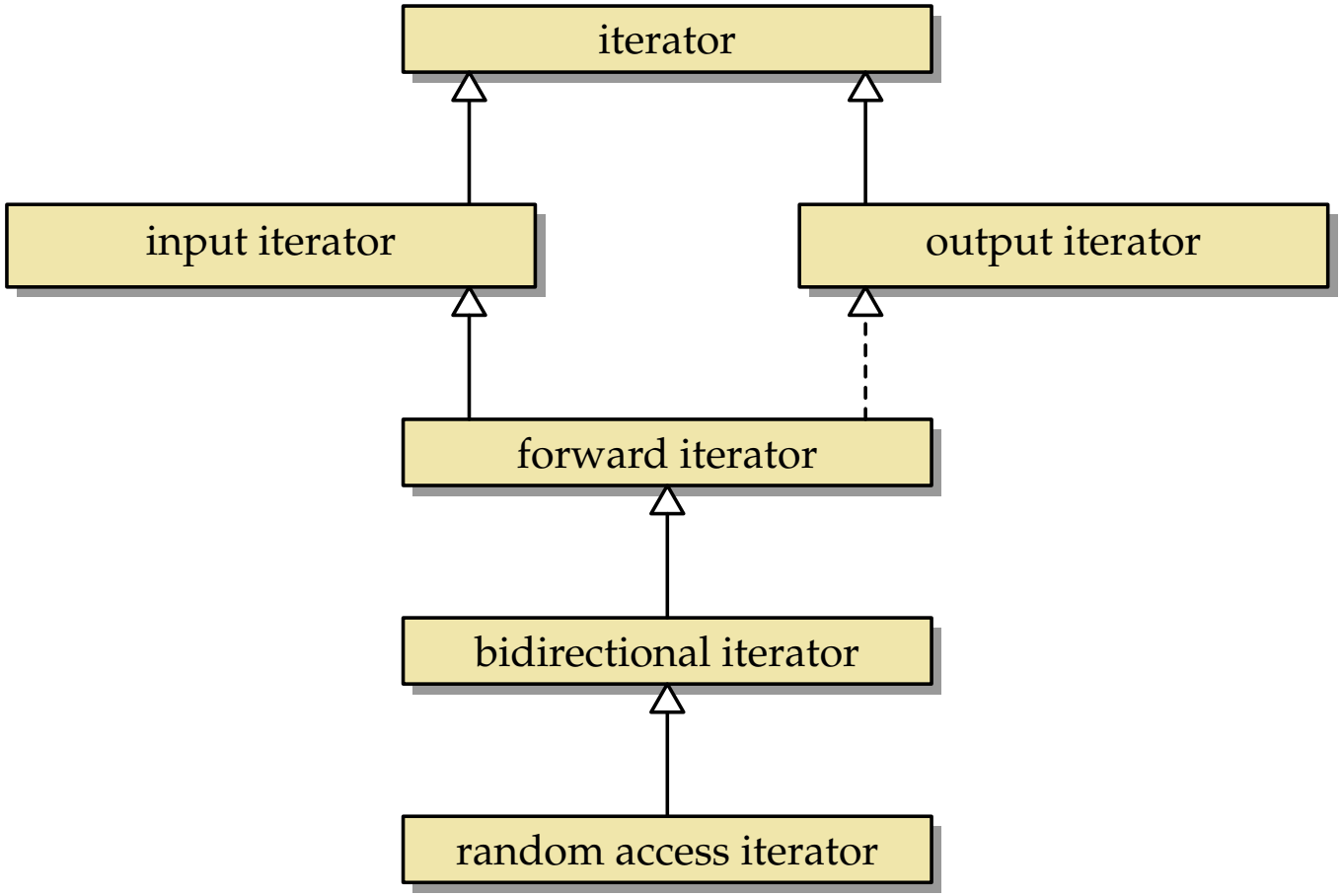
- Speziell konstruierte Typen können separat behandelt werden, so dass sich etwa Zeiger von anderen Typen unterscheiden lassen.

```
template<typename T>
constexpr std::size_t dim(const T& vec) {
   return sizeof(vec)/sizeof(*vec);
}
```

- Funktionen, die mit *constexpr* deklariert sind, werden zur Überzeitzeit ausgeführt, wenn sie mit zur Übersetzzeit bekannten Werten arbeiten.
- Hier wird die Dimensionierung eines Arrays abgefragt.
- Obige Variante lässt sich auf beliebige Typen anwenden, scheitert aber, wenn *vec* sich nicht dereferenzieren lässt. Es werden auch Zeigertypen zugelassen, bei denen dann eine 1 zurückgeliefert wird.
- Diese Fassung funktioniert nur für die Arrays, für die es gedacht ist:

```
template<typename T, std::size_t N>
constexpr std::size_t dim(const T (&vec)[N]) {
   return N;
}
```

- Iteratoren in C++ können als Verallgemeinerung von Zeigern gesehen werden, die einen universellen Zugriff auf Datenstrukturen erlauben.
- Durch die syntaktisch gleichartige Verwendung von Iteratoren und Zeigern können mit Hilfe des statischen Polymorphismus auch reguläre Zeiger als Iteratoren verwendet werden.
- Für die einzelnen Operationen eines Iterators gibt es einheitliche semantische Spezifikationen, die auch die jeweilige Komplexität angeben. Diese entsprechen denen der klassischen Zeiger-Operatoren.
- Dies sollte eingehalten werden, damit die auf Iteratoren arbeitenden Algorithmen semantisch korrekt sind und die erwartete Komplexität haben.
- Die auf Iteratoren basierenden Algorithmen sind immer generisch, d.h. es wird typischerweise mit entsprechenden impliziten Template-Parametern gearbeitet.



- Der C++-Standard spezifiziert eine (auf statischem Polymorphismus) beruhende semantische Hierarchie der Iteratoren-Klassen.

Alle Iteratoren erlauben das Dereferenzieren und das Weitersetzen:

Operator	Rückgabe-Typ	Beschreibung
$*it$	*Element&*	Zugriff auf ein Element; nur zulässig, wenn *it* dereferenzierbar ist
$++it$	*Iterator&*	Iterator vorwärts weitersetzen

Iteratoren unterstützen Kopierkonstruktoren, Zuweisungen und *std*::*swap*. Wieweit ein Iterator weitergesetzt werden darf bzw. ob jeweils eine Dereferenzierung zulässig ist, lässt sich diesen Operationen nicht entnehmen.

Diese Iteratoren erlauben es, eine Sequenz zu konsumieren, d.h. sukzessive auf die einzelnen Elemente zuzugreifen:

Operator	Rückgabe-Typ	Beschreibung
it1 != *it2*	**bool**	Vergleich zweier Iteratoren
∗*it*	*Element*	Lesezugriff auf ein Element
it−>*member*	Typ von *member*	Lesezugriff auf ein Datenfeld
++*it*	*Iterator*&	Iterator zuerst vorwärts weitersetzen
it++	*Iterator*&	Iterator danach vorwärts weitersetzen
∗*it*++	*Element*	dereferenziert den Iterator und liefert diesen Wert; der Iterator wird danach weitergesetzt

avg.cpp

```
#include <iostream>
#include <iterator>

int main() {
   std::istream_iterator<double> it(std::cin);
   std::istream_iterator<double> end;
   unsigned int count = 0; double sum = 0;
   while (it != end) {
      sum += *it++; ++count;
   }
   std::cout << (sum / count) << std::endl;
}
```

- *std::istream_iterator* ist ein Input-Iterator, der den >>-Operator verwendet, um die einzelnen Werte des entsprechenden Typs von der Eingabe einzulesen.
- Charakteristisch ist hier der konsumierende Charakter, d.h. es ist nur ein einziger Durchlauf möglich.

avg.hpp

```
#ifndef AVG_HPP
#define AVG_HPP
#include <cstdlib>
#include <type_traits>

template<typename ForwardIterator>
auto avg(ForwardIterator it1, ForwardIterator it2)
      -> decltype(*it1 / sizeof(int)) {
   using T = decltype(*it1 + *it1);
   if (it1 == it2) {
      return T{};
   }
   T sum{}; std::size_t count = 0;
   while (it1 != it2) {
      sum += *it1++; ++count;
   }
   return sum / count;
}

#endif
```

- Die Ermittlung des Durchschnitts lässt sich auch als Template-Funktion formulieren.

avg.hpp

```
template<typename ForwardIterator>
auto avg(ForwardIterator it1, ForwardIterator it2)
      -> decltype(*it1 / sizeof(int)) {
   using T = decltype(*it1 + *it1);
   if (it1 == it2) {
      return T{};
   }
   T sum{}; std::size_t count = 0;
   while (it1 != it2) {
      sum += *it1++; ++count;
   }
   return sum / count;
}
```

- Mit **decltype** kann der Typ eines Ausdrucks wie ein Typname verwendet werden.
- Es ist darauf zu achten, dass *decltype*(∗*it1*) für einen *std*::*istream_iterator*<**double**> hier **const double**& liefern würde. So etwas könnte bei Bedarf mit Hilfe von *std*::*decay* auf **double** reduziert werden.

avg-test.cpp

```
#include <iostream>
#include <iterator>
#include "avg.hpp"

int main() {
   std::istream_iterator<double> it(std::cin);
   std::istream_iterator<double> end;
   std::cout << avg(it, end) << std::endl;
}
```

- Nun lässt sich *avg* auf eine beliebige mit Forward-Iteratoren spezifizierte Menge an Daten anwenden, bei denen die Template-Abhängigkeiten erfüllt sind.

Diese Iteratoren erlauben es, den Objekten einer Sequenz sukzessive neue Werte zuzuweisen oder eine Sequenz neu zu erzeugen:

Operator	Rückgabe-Typ	Beschreibung
$*it = element$	–	Zuweisung; *it* ist danach nicht notwendigerweise dereferenzierbar
$++it$	*Iterator&*	Iterator vorwärts weitersetzen
$it++$	*Iterator&*	Iterator vorwärts weitersetzen
$*it++ = element$	–	Zuweisung mit anschließendem Weitersetzen des Iterators

Zu beachten ist hier, dass Mehrfachzuweisungen auf $*it$ nicht notwendigerweise zulässig sind. Die Dereferenzierung ist nur auf der linken Seite einer Zuweisung zulässig.

inserter.cpp

```
#include <iostream>
#include <list>
#include <iterator>
#include <algorithm>
int main() {
   std::list<double> values;
   std::istream_iterator<double> input(std::cin);
   std::istream_iterator<double> input_end;
   std::back_insert_iterator<std::list<double>> output(values);
   std::copy(input, input_end, output);
   for (auto value: values) { std::cout << value << std::endl; }
}
```

- Einfüge-Iteratoren sind Output-Iteratoren, die alle ihnen übergebenen Werte in einen Container einfügen.
- Zur Verfügung stehen *std::back_insert_iterator*, *std::front_insert_iterator* und *std::insert_iterator*.
- *std*::*copy* ist ein im Standard vorgegebener Algorithmus, der zwei einen Bereich definierende Input-Iteratoren und einen Output-Iterator als Parameter erhält.

Forward-Iteratoren unterstützen alle Operationen eines Input-Iterators. Im Vergleich zu diesen ist es zulässig, die Sequenz mehrfach zu durchlaufen. Entsprechend gilt:

- Aus $it1 == it2$ folgt $++it1 == ++it2$.
- Wenn $it1 == it2$ gilt, dann sind entweder beide Iteratoren dereferenzierbar oder keiner der beiden.
- Wenn die Iteratoren $it1$ und $it2$ dereferenzierbar sind, dann gilt $it1 == it2$ genau dann, wenn $*it1$ und $*it2$ das gleiche Objekt adressieren.

Forward-Iteratoren können auch die Operationen eines Output-Iterators unterstützen. Dann sind sie schreibbar und erlauben Mehrfachzuweisungen, ansonsten erlauben sie nur Lesezugriffe (*constant iterator*).

Beispiel für einen Forward-Iterator

forward.cpp

```
#include <iostream>
#include <forward_list>
#include <iterator>
#include <algorithm>
int main() {
   std::forward_list<double> values;
   std::istream_iterator<double> input(std::cin);
   std::istream_iterator<double> input_end;
   std::copy(input, input_end, front_inserter(values));
   // move all values < 0 to the front:
   auto middle = std::partition(values.begin(), values.end(),
      [](double val) { return val < 0; });
   std::ostream_iterator<double> out(std::cout, " ");
   std::cout << "negative values: " << std::endl;
   std::copy(values.begin(), middle, out); std::cout << std::endl;
   std::cout << "non-negative values: " << std::endl;
   std::copy(middle, values.end(), out); std::cout << std::endl;
}
```

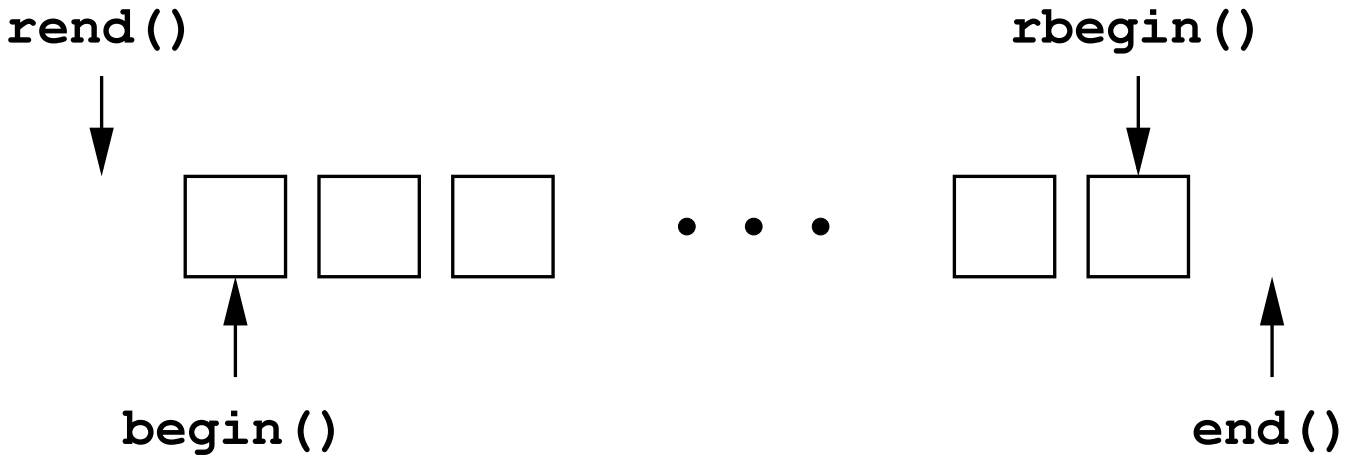
- *std*::*forward_list* ist eine einfach verkettete lineare Liste.

Bidirektionale Iteratoren sind Forward-Iteratoren, bei denen der Iterator in beide Richtungen versetzt werden kann:

Operator	Rückgabe-Typ	Beschreibung
$--it$	*Iterator&*	Iterator zuerst rückwärts weitersetzen
$it--$	*Iterator&*	Iterator danach rückwärts weitersetzen
$*it-- = element$	*Element&*	Zuweisung mit anschließendem Zurücksetzen des Iterators

Random-Access-Iteratoren sind bidirektionale Iteratoren, die die Operationen der Zeigerarithmetik unterstützen:

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 $it1 < it2$



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

- 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 bei einem *iterator* oder *const_iterator* den Container nicht verlassen, nicht einmal um einen einzelnen Schritt.

Durch die polymorphen Iteratoren wurde beginnend mit C++11 eine entsprechende auf Iteratoren basierende polymorphe Form der **for**-Anweisung möglich. Dabei wird

for (⟨for-range-declaration⟩ : ⟨for-range-initializer⟩)
⟨statement⟩

durch den Übersetzer zu folgendem Programmtext expandiert:

```
{
    auto&& __range = ⟨for-range-initializer⟩;
    auto __begin = begin-expr;
    auto __end = end-expr;
    for (; __begin != __end; ++__begin) {
        ⟨for-range-declaration⟩ = *__begin;
        ⟨statement⟩
    }
}
```

(Der expandierte Text wurde Abschnitt 9.5.4 im ISO-Standard 14882:2017 entnommen.)

```
{
    auto&& __range = ⟨for-range-initializer⟩;
    auto __begin = begin-expr;
    auto __end = end-expr;
    for (; __begin != __end; ++__begin) {
        ⟨for-range-declaration⟩ = *__begin;
        ⟨statement⟩
    }
}
```

begin-expr und *end-expr* werden wie folgt bestimmt:

- Bei Arrays wird Zeigerarithmetik verwendet, d.h. *__range* bzw. *__range* + *__bound*, wobei *__bound* die Länge des Arrays ist.
- Wenn *__range* einer Klasse angehört mit den Methoden *begin* und *end*, werden diese verwendet.
- Zuletzt wird nach passenden Funktionen *begin*(*__range*) und *end*(*__range*) gesucht.

sample-lines.cpp

```
class Line: public std::string {
   friend std::istream& operator>>(std::istream& in, Line& line) {
      return std::getline(in, line);
   }
};

int main() {
   std::istream_iterator<Line> it(std::cin);
   std::istream_iterator<Line> it_end;
   ReservoirSampler<Line> r(10, it, it_end);
   for (auto& line: r) {
      std::cout << line << std::endl;
   }
}
```

- Wenn die *ReservoirSampler*-Klasse Iteratoren unterstützt, lässt sich die Anwendung deutlich vereinfachen.
- Um mit stream-basierten Input-Operatoren die Eingabe zeilenweise zu bearbeiten, benötigen wir hier eine abgeleitete Klasse *Line*, für die wir den Eingabe-Operator für *std*::*string* überdefinieren können.

reservoir-sampler.hpp

```
template <typename Iterator>
void add(Iterator it1, Iterator it2) {
   while (it1 != it2) {
      add(*it1++);
   }
}
```

- Mit einer weiteren *add*-Methode kann mit Hilfe von Iteratoren eine Menge von hinzuzufügenden Objekten spezifiziert werden.
- Diese Methode kann auch sogleich von einem entsprechenden Konstruktor verwendet werden:

reservoir-sampler.hpp

```
template<typename Iterator>
ReservoirSampler(std::size_t size, Iterator it1, Iterator it2) :
      ReservoirSampler(size) {
   add(it1, it2);
}
```

reservoir-sampler.hpp

```
using Iterator = const T*;
Iterator begin() const {
   return data;
}
Iterator end() const {
   return data + get_size();
}
```

- Dank der Methoden *begin* und *end* wird das Durchlaufen eines Reservoirs mit einer entsprechenden **for**-Anweisung möglich.
- Wenn die Objekte zusammenhängend als Array im Speicher liegen, dann können auch problemlos Zeiger als Iteratoren verwendet werden.

Idee: Ist es möglich,

```
for (int i = 1; i < 10; ++i) {
    // ...
}
```

zu

```
for (auto i: range(1, 10)) {
    // ...
}
```

zu vereinfachen? Prinzipiell ja: Wir benötigen eine entsprechende Klasse mit einem zugehörigen Iterator-Typ, der die Operatoren $*$ und $++$ unterstützt.

range.hpp

```
template <typename T>
class IntegralRange {
   public:
      IntegralRange(T begin_val, T end_val) :
         begin_val(begin_val), end_val(end_val) {
      }

      class Iterator {
         // ...
      };

      Iterator begin() const {
         return Iterator(begin_val);
      }
      Iterator end() const {
         return Iterator(end_val);
      }

   private:
      T begin_val;
      T end_val;
};
```

range.hpp

```
class Iterator {
   public:
      Iterator(T val) : val(val) {
      }
      T operator*() {
         return val;
      }
      Iterator& operator++() {
         ++val; return *this;
      }
      Iterator& operator++(int) {
         Iterator it = *this;
         ++val; return it;
      }
      bool operator==(const Iterator& other) const {
         return val == other.val;
      }
      bool operator!=(const Iterator& other) const {
         return val != other.val;
      }

   private:
      T val;
};
```

Das müsste nun in eine entsprechende Template-Funktion verpackt werden, die den Typparameter automatisch bestimmt:

```
template<typename T>
IntegralRange<T> range(T begin_val, T end_val) {
    return IntegralRange<T>(begin_val, end_val);
}
```

Problem: Diese Template-Funktion akzeptiert zunächst prinzipiell beliebige T, geht dann aber schief, wenn T sich nicht wie ein ganzzahliger Datentyp verhält. Ist es möglich, hier eine Einschränkung vorzunehmen?

- Wenn der C++-Übersetzer alle in Frage kommenden Kandidaten einer Template-Funktion in Betracht zieht, werden zunächst die Template-Parameter nur in die Template-Parameter-Deklaration und die Funktionsdeklaration (Parameter und Return-Typ) eingesetzt und danach überprüft, ob das Resultat semantisch zulässig ist.
- Wenn die Antwort nein ist, dann ist das kein Fehler. Stattdessen wird nur ganz einfach der Kandidat aus dem Pool der Kandidaten entfernt.
- Das Prinzip nennt sich SFINAE: *Substitution Failure Is Not An Error*.

Angenommen, wir wollen bei *range* nur **int** und **unsigned int** zulassen. Dann könnten wir das so organisieren:

```
template <typename T> struct is_integer {};
template <> struct is_integer<int> {
    using type = IntegralRange<int>;
}
template <> struct is_integer<unsigned int> {
    using type = IntegralRange<unsigned int>;
}

template<typename T>
typename is_integer<T>::type
range(T begin_val, T end_val) {
    return IntegralRange<T>(begin_val, end_val);
}
```

Hier wird nun *is_integer*<*T*>::*type* benutzt, das nur für die Fälle **int** und **unsigned int** definiert ist. Für alle anderen Typen gibt es keinen *type*, was zu einem Ausschluss per SFINAE führt.

Es lohnt sich, die Bedingungen (wie hier *is_integer*) von dem gewünschten Typ zu trennen (war hier *IntegralRange*<*T*>). Die Standard-Bibliothek bietet hierfür *std*::*enable_if*, das so definiert sein könnte:

```
template<bool B, class T = void>
struct enable_if {};

template<class T>
struct enable_if<true, T> { using type = T; };
```

Der erste Parameter ist die **bool**-Bedingung, der zweite Parameter spezifiziert den gewünschten Typ.

Als nächstes wird eine Hilfsklasse benötigt, die auf der Template-Ebene einen konstanten Wert eines integralen Typs repräsentiert. Die C++-Bibliothek bietet hierfür *std*::*integral_constant* an, das wie folgt implementiert sein könnte:

```
template<class T, T v>
struct integral_constant {
    static constexpr T value = v;
    using value_type = T;
    using type = integral_constant;
    constexpr operator value_type() const noexcept {
        return value;
    }
    constexpr value_type operator()() const noexcept {
        return value;
    }
};
```

Darauf basierend lassen sich Hilfsklassen für **bool**-Werte definieren:

```
template <bool B>
using bool_constant = integral_constant<bool, B>;

using true_type = bool_constant<true>;
using false_type = bool_constant<false>;
```

Nun lässt sich die Aufzählung der zugelassenen Typen unabhängig vom Resultat-Typ umsetzen:

```
template <typename T> struct is_integer :
    public std::false_type{};
template <> struct is_integer<int> :
    public std::true_type{};
template <> struct is_integer<unsigned int> :
    public std::true_type{};

template<typename T>
typename std::enable_if<
    is_integer<T>::value, // boolean-valued condition
    IntegralRange<T> // wanted type, if correct
>::type // is the wanted type, if correct
range(T begin_val, T end_val) {
    return IntegralRange<T>(begin_val, end_val);
}
```

In <*type_traits*> finden sich aber bereits eine Vielzahl einzelner Tests, u.a. auch *std*::*is_integral*, der alle integralen Typen umfasst:

```
template<typename T>
typename std::enable_if<
    std::is_integral<T>::value,
    IntegralRange<T>
>::type
range(T begin_val, T end_val) {
    return IntegralRange<T>(begin_val, end_val);
}
```

Wenn wir eine weitere Variante für Zeiger und Iteratoren zulassen wollen, könnten wir im einfachsten Falle die ganzzahligen Typen ausschließen:

```
template<typename T>
typename std::enable_if<
    !std::is_integral<T>::value,
    IntegralRange<T>
>::type
range(T begin_it, T end_it) {
    return IteratorRange<T>(begin_it, end_it);
}
```

Es wäre aber besser, die auf *IntegralRange* basierende Variante auf Typen zu beschränken, die wie Zeiger oder Iteratoren aussehen, d.h. die die unären Operatoren $*$ und $++$ unterstützen.
Wir benötigen hierzu etwas Handwerkszeug, um per SFINAE die Unterstützung von Operatoren und Methoden zu testen:

- Mit *std*::*declval* können wir aus einem Typ ein Objekt des Typs machen, ohne zu wissen, wie der Konstruktor aussieht.
- Mit **decltype** kann der Typ eines Ausdrucks wie ein Typname verwendet werden.
- Mit *std*::*remove_reference* werden wir && bzw. & los.

std::*declval* ist eine Template-Funktion aus <*utility*>, die nur deklariert, jedoch nie definiert wird:

```
template<class T>
typename std::add_rvalue_reference<T>::type declval();
```

- Wir dürfen *declval* somit nur in Konstruktionen einsetzen, die vollständig zur Übersetzzeit ausgewertet werden und bei denen nur der Typ relevant ist, jedoch nicht die (nicht vorhandene) Definition der Funktion.
- Mit *add_rvalue_reference* bleiben Referenztypen so wie sie sind, nur Typen ohne Referenz (also weder & noch &&) werden mit && versehen.
- Diese Ergänzung ist notwendig, um beispielsweise die Existenz von Methoden testen zu können, die nur für *rvalue*-Objekte zugänglich sind: **struct** *foo* { **void** *bar*()&& { /*... */} };

Angenommen, wir haben einen Template-Parameter für einen Zeigertyp oder Iterator und wollen den Typ nach der Dereferenzierung haben:

```
template<typename T>
struct dereferenced {
    using type = decltype(*std::declval<T>());
};
```

- Innerhalb von **decltype** wird nur der Typ eines Ausdrucks bestimmt, dieser jedoch nicht bewertet. Deswegen ist *std*::*declval*<*T*>() zulässig und liefert ein virtuelles Objekt des Typs *T*, das wir dann dereferenzieren können, um davon mit **decltype** den Typ zu bestimmen.
- *dereferenced*<**int***>::*type* und *dereferenced*<*std*::*vector*<**int**>::*iterator*> entsprechen nun jeweils **int**.

Nun lässt sich das so kombinieren, dass wir abprüfen, ob die Operatoren $*$ und $++$ unterstützt werden:

```
template<typename T>
typename std::remove_reference<
    decltype(
        *std::declval<T&>(), // * supported?
        ++std::declval<T&>(), // ++ supported?
        /* wanted type, comes with &&: */
        std::declval<IteratorRange<T>>()
    )
>::type // now with && removed
range(T begin_it, T end_it) {
    return IteratorRange<T>(begin_it, end_it);
}
```

Da diese Lösungen wenig elegant aussehen und auch nicht entgegenkommend sind bei Fehlermeldungen, wurde bereits seit langer Zeit in C++ über ein geeigneteres Sprachmittel nachgedacht, den sogenannten *concepts*:

- Bislang sind die *concepts* noch nicht in den Standard (zuletzt C++17) aufgenommen worden. Die Aufnahme in C++20 ist aber geplant.
- Es gibt eine technische Spezifikation für *concepts* (ISO/IEC TS 19217:2015), die kurz als *concepts TS* bezeichnet werden. Diese Variante wird vom g++ mit der Option „-fconcepts“ unterstützt. Ein dem nahekommender Draft steht unter N4549 zur Verfügung.
- Für C++20 wurden Überarbeitungen vorgeschlagen (P0587R0), die zu der in P0734R0 dokumentierten Arbeitsfassung für C++20 geführt haben.
- Die C++20-Fassung ist bislang nirgends umgesetzt. Die folgenden Beispiele lassen sich mit „-fconcepts“ übersetzen.

Die wichtigste Ergänzung sind die ⟨requires-expression⟩s, die es erlauben, Anforderungen zu spezifizieren:

⟨primary-expression⟩ ⟶ ⟨requires-expression⟩

⟨requires-expression⟩ ⟶ **requires**
[⟨requirement-parameter-list⟩]
⟨requirement-body⟩

⟨requirement-parameter-list⟩ ⟶ „(“
[⟨parameter-declaration-clause⟩] „)“

⟨requirement-body⟩ ⟶ „{“ ⟨requirement-seq⟩ „}“

⟨requirement-seq⟩ ⟶ ⟨requirement⟩

⟶ ⟨requirement-seq⟩ ⟨requirement⟩

⟨requirement⟩ ⟶ ⟨simple-requirement⟩

⟶ ⟨type-requirement⟩

⟶ ⟨compound-requirement⟩

⟶ ⟨nested-requirement⟩

Einzelne Anforderungen innerhalb eines ⟨requirement-body⟩ können wie folgt aussehen:

⟨simple-requirement⟩ ⟶ ⟨expression⟩ „;“

⟨type-requirement⟩ ⟶ **typename** [⟨nested-name-specifier⟩] ⟨type-name⟩ „;“

⟨compound-requirement⟩ ⟶ „{“ ⟨expression⟩ „}“ [**noexcept**] [⟨trailing-return-type⟩] „;“

⟨nested-requirement⟩ ⟶ ⟨requires-clause⟩ „;“

Zu den Deklarationen werden die Konzepte hinzugefügt, die anschließend in Typspezifikationen referenziert werden können:

$$
\begin{aligned}
\langle\text{decl-specifier}\rangle &\longrightarrow \textbf{concept} \\
\langle\text{simple-type-specifier}\rangle &\longrightarrow \langle\text{constrained-type-specifier}\rangle \\
\langle\text{constrained-type-specifier}\rangle &\longrightarrow \langle\text{qualified-concept-name}\rangle
\end{aligned}
$$

forward-iterator.hpp

```
template<typename T>
concept bool ForwardIterator = requires(T it1, T it2) {
   { it1 == it2 } -> bool;
   { it1 = it2 } -> T&;
   { *it1 };
   { ++it1 } -> T;
};
```

- *ForwardIterator* ist ein Beispiel für ein *concept*, das die wichtigsten Anforderungen eines solchen Iterators aufführt.
- Hier sind die Anforderungen erfüllt wenn die einzelnen ⟨compound-requirement⟩s zulässig sind und, sofern angegeben, den entsprechenden Datentyp zurückliefern.
- Bei Concepts TS muss da noch jeweils **bool** als Datentyp für das Konzept angegeben werden, bei C++20 wird dies wohl entfallen, weil in diesem Kontext nur **bool** sinnvoll ist.

Concepts TS: Anforderungen bei Template-Deklarationen

Bei den Template-Deklarationen kommt dann die Möglichkeit hinzu, Anforderungen an die Template-Parameter hinzuzufügen:

⟨template-declaration⟩ ⟶ **template** „<“ ⟨template-parameter-list⟩ „>“
[⟨requires-clause⟩] ⟨declaration⟩

⟨requires-clause⟩ ⟶ **requires** ⟨constraint-expression⟩

⟨constraint-expression⟩ ⟶ ⟨logical-or-expression⟩

range.hpp

```
template <typename T>
requires std::is_integral<T>::value
class IntegralRange {
   // ...
};
```

- Anforderungen können jetzt sehr leicht auch in Template-Deklarationen integriert werden.
- Und bei Template-Funktionen muss nicht mehr auf SFINAE zurückgegriffen werden:

range.hpp

```
template<typename T>
requires std::is_integral<T>::value
IntegralRange<T> range(T begin_val, T end_val) {
   return IntegralRange<T>(begin_val, end_val);
}
```

range.hpp

```
template <typename IT>
requires ForwardIterator<IT>
class IteratorRange {
   // ...
};
```

- Anforderungen können sich auf benannte Konzepte beziehen, sowohl bei Template-Klassen als auch bei Template-Funktionen.

range.hpp

```
template<typename T>
requires ForwardIterator<T>
IteratorRange<T> range(T begin_it, T end_it) {
   return IteratorRange<T>(begin_it, end_it);
}
```

- 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.
- Über <*type_traits*> können diverse Eigenschaften von Typen abgefragt oder getestet werden.

Sum.hpp

```
#ifndef SUM_HPP
#define SUM_HPP

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 <cstdlib>
#include <iostream>
#include "Sum.hpp"
template<typename T, std::size_t N>
constexpr std::size_t dim(T (&vec)[N]) { return N; }
int main() {
   int numbers[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   std::cout << "sum of numbers[] = " <<
      sum(numbers, numbers + dim(numbers)) << std::endl;
   float floats[] = {1.2, 3.7, 4.8};
   std::cout << "sum of floats[] = " <<
      sum(floats, floats + dim(floats)) << std::endl;
   char text[] = "Hallo zusammen, dies ist etwas Text!!";
   std::cout << "sum of text[] = " << sum(text, text + dim(text)) <<
      std::endl;
}
```

- In den beiden Tests mit **int** und **float** klappt des problemlos, jedoch nicht mit **char**...
- 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$
```

- Wieso wird „,“ ausgegeben, wenn wir eine numerische Summe erwartet hätten?

Definition einer Traits-Template-Klasse

SumTraits.hpp

```
#ifndef SUM_TRAITS_HPP
#define SUM_TRAITS_HPP

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

// special case for char
template <>
class SumTraits<char> {
   public:
      using SumValue = int;
};
#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*.

Sum2.hpp

```
#ifndef SUM2_HPP
#define SUM2_HPP

#include "SumTraits.hpp"

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

#endif
```

- Statt *T* wird hier jetzt *SumTraits*<*T*>::*SumValue* als Typ für die Summe verwendet.

TestSum2.cpp

```
#include <cstdlib>
#include <iostream>
#include "Sum2.hpp"
template<typename T, std::size_t N>
constexpr std::size_t dim(T (&vec)[N]) { return N; }
int main() {
   int numbers[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   std::cout << "sum of numbers[] = " <<
      sum(numbers, numbers + dim(numbers)) << std::endl;
   float floats[] = {1.2, 3.7, 4.8};
   std::cout << "sum of floats[] = " <<
      sum(floats, floats + dim(floats)) << std::endl;
   char text[] = "Hallo zusammen, dies ist etwas Text!!";
   std::cout << "sum of text[] = "
      << sum(text, text + dim(text)) << std::endl;
}
```

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

Sum3.hpp

```
#ifndef SUM3_HPP
#define SUM3_HPP

#include "SumTraits.hpp"

template <typename T, typename ST = SumTraits<T>>
inline typename ST::SumValue sum(const T* begin, const T* end) {
   using SumValue = typename ST::SumValue;
   auto result = SumValue();
   for (auto it = begin; it < end; ++it) {
      result += *it;
   }
   return result;
}

#endif
```

- C++ unterstützt voreingestellte Template-Parameter, seit C++11 auch bei Template-Funktionen.
- Diese Konstruktion ermöglicht dann einem Nutzer dieser Konstruktion die Voreinstellung zu übernehmen oder bei Bedarf eine eigene Traits-Klasse zu spezifizieren.

TestSum3.cpp

```
#include <cstdlib>
#include <iostream>
#include "Sum3.hpp"

template<typename T, std::size_t N>
constexpr std::size_t dim(T (&vec)[N]) { return N; }

struct MyTraits {
   using SumValue = long long int;
};

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

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

Sum4.hpp

```
template <typename ST, typename T>
inline typename ST::SumValue sum(const T* begin, const T* end) {
   using SumValue = typename ST::SumValue;
   auto result = SumValue();
   for (auto 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<SumTraits<T>, T>(begin, end);
}
```

- Den automatisierbar bestimmbaren Template-Parametern sollten diejenigen vorangehen, die u.U. abweichend bestimmt werden.
- Umgekehrt gilt, dass den Template-Parameter mit Voreinstellungen nicht solche ohne Voreinstellungen folgen dürfen.
- Der Konflikt lässt sich durch zwei Varianten lösen: einer generellen mit zwei Template-Parametern und dem Spezialfall mit nur einem Template-Parameter.

TestSum4.cpp

```
#include <cstdlib>
#include <iostream>
#include "Sum4.hpp"

template<typename T, std::size_t N>
constexpr std::size_t dim(T (&vec)[N]) { return N; }

struct MyTraits {
   using SumValue = long long int;
};

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

- Nun muss nur noch die Traits-Klasse angegeben werden, jedoch nicht mehr der Elementtyp des Arrays, der sich aus dem Parameter ableiten lässt.

- Statischer Polymorphismus basiert in C++ auf Templates und der Verlagerung der semantischen Überprüfung auf den Zeitpunkt der Instantiierung.
- Ein wesentliches Element ist die automatisierte Auswahl der am besten passenden Template-Klasse oder Template-Funktion zur Übersetzzeit.
- Die elegante Erweiterbarkeit ergibt sich aus der Möglichkeit, dass Varianten einfach per **#include** hinzugefügt werden können. Dann werden sie implizit überall berücksichtigt, wo sie anwendbar sind.
- Traits erlauben es, Eigenschaften von Typen zur Übersetzzeit zu spezifizieren und auszuwerten.
- Sowohl bei Template-Klassen als auch bei Template-Funktionen sind per SFINAE frei definierbare Einschränkungen möglich. Bei generischen Klassen ohne Einschränkungen besteht die Gefahr, dass die elegante Erweiterbarkeit für weitere Spezialfälle nicht mehr möglich ist.
- Mit *concepts* wird dies hoffentlich bald sehr viel eleganter möglich sein.

- Funktionsobjekte sind Objekte, die bei einem Funktionsaufruf zulässig sind, etwa indem der **operator**() für sie definiert ist.
  (Siehe ISO 14882-2017, Abschnitt 23.14.)
- 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 std::function<T(T)> {
   public:
      T operator()(T x) const noexcept { return x * x; }
};
```

- Die von *std::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<typename R, typename... 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() {
   std::list<int> ints;
   for (int i = 1; i <= 10; ++i) {
      ints.push_back(i);
   }

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

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

- *std::transform* gehört zu den in der STL definierten Operatoren, die auf durch Iteratoren spezifizierten Sequenzen arbeiten.
- Die ersten beiden Parameter von *std::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

```
std::transform(ints.begin(), ints.end(),
   std::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.
- *std::transform* selbst fügt also keine Objekte irgendwo ein, sondern nimmt Zuweisungen vor.
- Funktionen wie *std::back_inserter* erzeugen einen speziellen Iterator für einen Container, der neue Objekte einfügt (hier immer an das Ende der Sequenz).
- Bei *std::transform* wäre auch eine direkte Ersetzung möglich gewesen der ursprünglichen Objekte:

transform.cpp

```
std::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 in den C++-Standard ISO-14882-2012.

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

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

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

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

   for (int val: squares) {
      std::cout << val << std::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⟩ ⟶ ⟨simple-capture⟩
⟶ ⟨init-capture⟩
⟨simple-capture⟩ ⟶ ⟨identifier⟩
⟶ „&“ ⟨identifier⟩
⟶ **this**
⟶ „*“ **this**
⟨init-capture⟩ ⟶ ⟨identifier⟩ ⟨initializer⟩
⟶ „&“ ⟨identifier⟩ ⟨initializer⟩
⟨lambda-declarator⟩ ⟶ „(“ ⟨parameter-declaration-clause⟩ „)“
[⟨decl-specifier-seq⟩] [⟨noexcept-specifier⟩]
[⟨attribute-specifier-seq⟩]
[⟨trailing-return-type⟩]

- Die ⟨init-capture⟩ kommt mit dem C++14-Standard hinzu.

- 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>
auto create_multiplier(T factor) {
   return [=](T val) { return factor*val; };
}

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

- *create_multiplier* ist eine Template-Funktion, die ein mit einem vorgegebenen Faktor multiplizierendes Funktionsobjekt erzeugt und zurückliefert.
- Die *lambda-capture* [=] legt fest, dass die aus der Umgebung referenzierten Variablen beim Erzeugen des Funktionsobjekts kopiert werden.
- Hinweis: In dieser Form geht es nur ab C++14, bei C++11 wäre es notwendig, den Return-Typ explizit hinzuschreiben. Um das zu erreichen, wäre eine Verpackung in *std::function* notwendig.

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>
auto create_multiplier(T factor) {
   return 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>
auto create_counter(T val) {
   auto p = std::make_shared<T>(val);
   auto incr = [=]() { return ++*p; };
   auto decr = [=]() { return --*p; };
   auto getval = [=]() { return *p; };
   return std::make_tuple(incr, decr, 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 *std*::*shared_ptr*.
- Aber normalerweise ist es einfacher, eine entsprechende Klasse zu schreiben.

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

- *create_counter* erzeugt ein Tupel (Datenstruktur aus **#include** <tuple>) und beginnend mit C++17 kann so ein Tupel mit Hilfe einer sogenannten *structured binding declaration* an mehrere Variablen zugewiesen werden, ohne den Typ benennen zu müssen.
- Danach bleibt die gemeinsame private Variable solange bestehen, bis diese von *shared_ptr* freigegeben wird, d.h. sobald die letzte Referenz darauf verschwindet.

```
std::vector<int> values(10);
int count = 0;
std::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.
- *std::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.

OutOfMemory.cpp

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

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

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

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

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

Crash.cpp

```
#include <iostream>

int main() {
   int count = 0;
   for(;;) {
      char* megabyte = new char[1048576]; *megabyte = 0;
      std::cout << " " << ++count << std::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.

```
theon$ ulimit -d 8192
theon$ ./crash 2>&1 | fold -w 60
 1 2 3 4 5 6 7terminate called after throwing an instance of
 'std::bad_alloc'
  what():  std::bad_alloc
theon$
```

- 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.

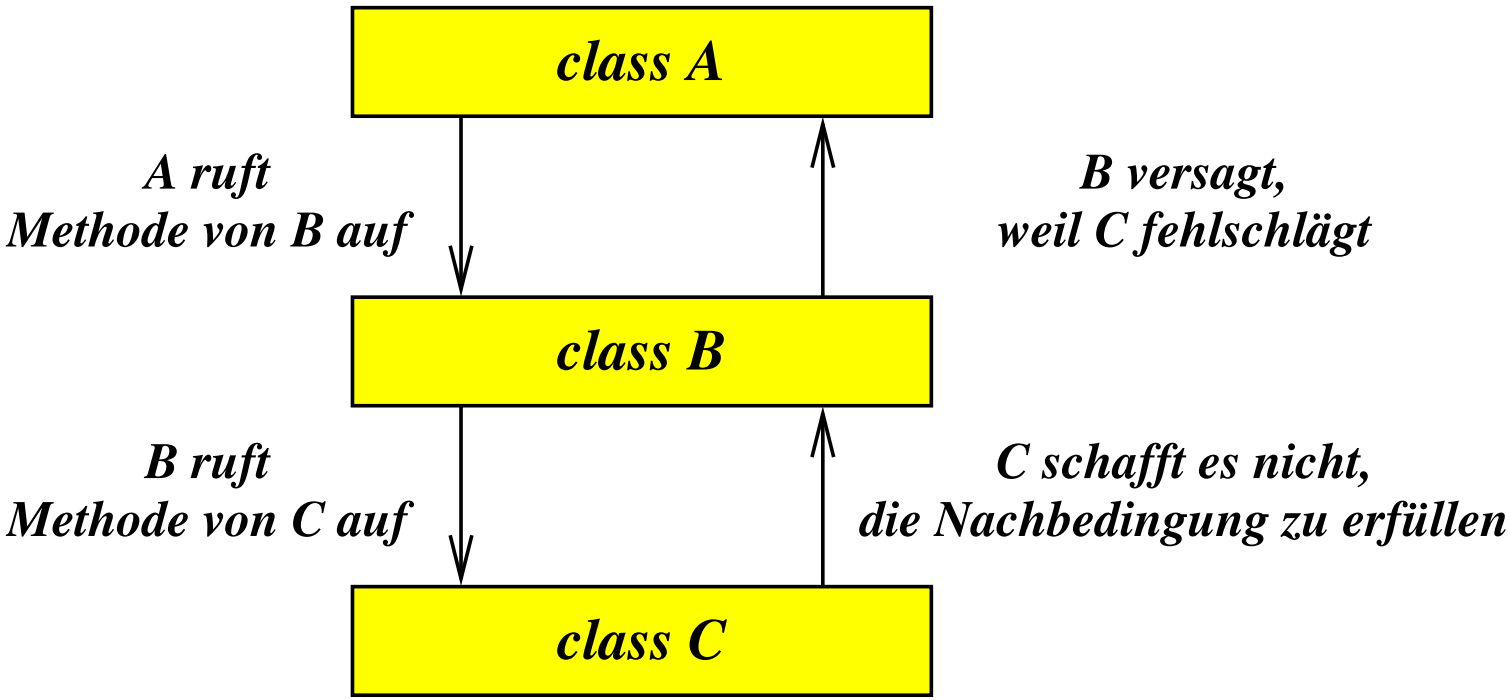
Beispiel: Übersetzer für C++:

- 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

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

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


```
theon$ ulimit -d 8192
theon$ ./VerboseOutOfMemory
 1 2 3 4 5 6 7 ... Game over!
This hit me: std::bad_alloc
theon$
```

- 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.

⟨noexcept-specifier⟩ ⟶ **noexcept** „(“ ⟨constant-expression⟩ „)“
⟶ **noexcept**
⟶ **throw** „(“ „)“

- Früher war mit **throw** eine Aufzählung der denkbaren Ausnahmen möglich – dies wurde inzwischen abgeschafft.
- Mit **noexcept** wird zugesichert, dass keine Ausnahme auftritt. Dies lässt Optimierungen zu.
- Wenn ein ⟨constant-expression⟩ angegeben wird, dann hängt dies von dem zur Übersetzzeit erfolgten Bewertung des Ausdrucks ab: **true** entspricht der Zusicherung, **false** heißt, dass die Zusicherung nicht gegeben ist.
- Wenn kein ⟨noexcept-specifier⟩ gegeben ist, entspricht dies **noexcept**(**false**).
- Wenn trotz einer **noexcept**-Zusicherung eine Methode oder Funktion direkt oder indirekt Ausnahmen auslöst, wird *std*::*unexpected* bzw. *std*::*terminate* aufgerufen.

ArrayedStack.hpp

```
template<class Item, std::size_t SIZE = 4>
class ArrayedStack {
   public:
      ArrayedStack() noexcept : index(0) {};

      bool empty() const noexcept { return index == 0; }
      bool full() const noexcept { return index == SIZE; }
      const Item& top() const { /* ... */ }
      void push(const Item& item) { /* ... */ }
      void pop() { /* ... */ }

   private:
      std::size_t index;
      Item items[SIZE];
}; // class ArrayedStack
```

- Wenn zugesichert werden kann, das keine Ausnahmen direkt oder indirekt ausgelöst werden, dann sollte **noexcept** in die Signatur aufgenommen 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.hpp

```
const Item& top() const {
   // PRE: not empty()
   if (index > 0) {
      return items[index-1];
   } else {
      throw EmptyStack();
   }
}

void push(const Item& item) {
   // PRE: not full()
   if (index < SIZE) {
      items[index] = item;
      index += 1;
   } else {
      throw FullStack();
   }
}
```

⟨throw-expression⟩ ⟶ **throw** [⟨assignment-expression⟩]

- Ein ⟨throw-expression⟩ löst eine Ausnahmenbehandlung aus, die durch das angebene Objekt repräsentiert wird, das das Problem beschreiben sollte.
- Innerhalb einer Ausnahmenbehandlung ist auch ein ⟨throw-expression⟩ ohne einen Ausdruck sinnvoll. In diesem Fall wird die Ausnahme an den Aufrufer weitergeleitet.
- Es ist hierbei erlaubt, temporäre Objekte zu verwenden, da diese bei Bedarf implizit kopiert werden.

- Nach dem Auflösen werden entsprechend der Aufrufverschachtelung sukzessive Blöcke abgebaut, bis ein passende Ausnahmenbehandlung gefunden wird.
- Bei jedem abgebauten Block werden alle zugehörigen Variablen dekonstruiert.
- Wird keine passende Ausnahmenbehandlung gefunden, wird *std::terminate* aufgerufen.

⟨statement⟩ ⟶ [⟨attribute-specifier-seq⟩] ⟨try-block⟩

⟨try-block⟩ ⟶ **try** ⟨compound-statement⟩ ⟨handler-seq⟩

⟨handler-seq⟩ ⟶ ⟨handler⟩ [⟨handler-seq⟩]

⟨handler⟩ ⟶ **catch** „(“ ⟨exception-declaration⟩ „)“ ⟨compound-statement⟩

⟨exception-declaration⟩ ⟶ [⟨attribute-specifier-seq⟩] ⟨type-specifier-seq⟩ ⟨declarator⟩

⟶ [⟨attribute-specifier-seq⟩] ⟨type-specifier-seq⟩ [⟨abstract-declarator⟩]

⟶ „...“

⟨function-body⟩ ⟶ [⟨ctor-initializer⟩] ⟨compound-statement⟩
⟶ ⟨function-try-block⟩
⟶ „=“ **default** „;“
⟶ „=“ **delete** „;“
⟨function-try-block⟩ ⟶ **try** [⟨ctor-initializer⟩]
⟨compound-statement⟩ ⟨handler-seq⟩

- Auch innerhalb eines ⟨ctor-initializer⟩ bei einem Konstruktor kann es zum Auslösen von Ausnahmen kommen.
- In diesem Falle käme eine Ausnahmenbehandlung innerhalb des regulären ⟨compound-statement⟩ zu spät.
- Daher kann dies beides durch einen ⟨function-try-block⟩ ersetzt werden.

LeakingObject.cpp

```
struct A {
   A(int i) : i(i) {
      if (i < 0) {
         throw i;
      }
   }
   int i;
};
```

- Prinzipiell können Konstruktoren auch Ausnahmen auslösen und das kann auch sehr sinnvoll sein, da kaum andere Wege der Fehlerbehandlung zur Verfügung stehen. Das erscheint vorteilhafter als der Umgang mit „Zombie-Objekten“, die scheinbar fertig konstruiert sind, aber die gewünschte Funktionalität nicht erfüllen.

LeakingObject.cpp

```
struct B {
   B(int i, int j) : p1(new A(i)), p2(new A(j)) { }
   ~B() { delete p1; delete p2; }
   A* p1; A* p2;
};
```

- Was passiert, wenn ein Konstruktor von einer Ausnahme betroffen ist?
  - Falls es bei **new** passierte, wird der Speicher umgehend freigegeben.
  - Dann werden alle bereits konstruierten Unterobjekte in umgekehrter Reihenfolge abgebaut.
  - Der *destructor* wird nie aufgerufen.
- Was passiert, wenn **new** $A(j)$ schiefgeht? Dann wird der hierfür belegte Speicher freigegeben und der Zeiger *p1* abgebaut. Das Abbauen eines Zeigers ist aber nicht mit einer Aktion verbunden. Wir hätten also ein Speicherleck.
- Deswegen sollte innerhalb eines ⟨ctor-initializer⟩ nie mehr als ein **new** vorkommen, um Lecks zu vermeiden.

CatchLeakingObject.cpp

```
struct B {
   B(int i, int j) try : p1(new A(i)), p2(new A(j)) {
   } catch(int val) {
      // what are our options here?
   }
   ~B() {
      delete p1; delete p2;
   }
   A* p1; A* p2;
};
```

- Gab es da nicht die Option, so etwas abzufangen?
- Ja, aber das ändert nichts daran, dass das Objekt nach dem unterbrochenen Abarbeiten des ⟨ctor-initializer⟩ nicht mehr erfolgreich konstruiert werden kann.
- Das bedeutet, dass es in jedem Fall mit einer Ausnahme weitergeht. Entweder die gleiche oder eine andere.
- Somit ist das nur in Ausnahmesituationen sinnvoll, wenn etwa eine zusätzliche Aktion für den korrekten Abbau notwendig ist.

CatchLeakingObject.cpp

```
struct B {
   B(int i, int j) try : p1((++count, new A(i))), p2((++count, new A(j))) {
   } catch(int val) {
      if (count == 2) delete p1;
   }
   ~B() {
      delete p1; delete p2;
   }
   int count = 0;
   A* p1; A* p2;
};
```

- Prinzipiell wäre es denkbar, fehlende Abbau-Aktionen wie etwa die fehlende Freigabe nachzuholen.
- Erstrebenswert oder lesbar ist das nicht.

Calculator.hpp

```
template<typename Value, typename Stack>
class Calculator {
   public:
      class Exception : public std::exception {};
      // ...

      Value calculate(const std::string& expr) {
         // PRE: expr in RPN (reversed polish notation) syntax
         // ...
      }

   private:
      Stack 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

Calculator.hpp

```
template<typename Value, typename Stack>
class Calculator {
   public:
      class Exception : public std::exception {};
      class SyntaxError : public Exception {
         public:
            virtual const char* what() const noexcept {
               return "syntax error";
            };
      }; // class SyntaxError
      class BadExpr : public Exception {
         public:
            virtual const char* what() const noexcept {
               return "invalid expression";
            };
      }; // class BadExpr
      class StackFailure : public Exception {
         public:
            virtual const char* what() const noexcept {
               return "stack failure";
            };
      }; // class StackFailure
   // ...
}; // class Calculator
```

- 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.

Calculator.hpp

```
Value calculate(const std::string& expr) {
   // PRE: expr in RPN (reversed polish notation) syntax
   std::istringstream in(expr);
   Value result; // return value
   try {
      std::string token;
      while (in >> token) {
         // ...
      }
      result = opstack.top(); opstack.pop();
      if (!opstack.empty()) {
         throw BadExpr();
      }
   } catch(FullStack) {
      throw StackFailure();
   } catch(EmptyStack) {
      throw BadExpr();
   }
   return result;
}
```

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

Calculator.hpp

```
while (in >> token) {
   if (token == "+" || token == "-" ||
         token == "*" || token == "/") {
      Value op2{opstack.top()}; opstack.pop();
      Value op1{opstack.top()}; opstack.pop();
      Value 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 {
      std::istringstream vin{token};
      Value value;
      if (vin >> value) {
         opstack.push(value);
      } else {
         throw SyntaxError();
      }
   }
}
result = opstack.top(); opstack.pop();
```

TestCalculator.cpp

```
int main() {
   std::string expr;
   while (std::cout << ": " && std::getline(std::cin, expr)) {
      try {
         Calculator<double, ArrayedStack<double>> calc;
         std::cout << calc.calculate(expr) << std::endl;
      } catch(std::exception& exc) {
         std::cerr << exc.what() << std::endl;
      }
   }
} // main
```

- Ausnahmen werden hier innerhalb der **while**-Schleife abgefangen, so dass ein Weiterarbeiten nach einem Fehler möglich ist.

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

- 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.

Ausnahmenbehandlungen brechen Abstraktionsgrenzen und können eine regelrechte Verwüstung hinterlassen, da ganze Ketten von Funktionsaufrufen abgeräumt werden können. Je nach Umfang des Schutzes lassen sich verschiedene Grade voneinander unterscheiden:

- Gar kein Schutz.
- Elementarer Schutz gegen Speicherlecks und das Hinterlassen offener Ressourcen. Dieser Schutz basiert auf der konsequenten Anwendung von RAII-Objekten.
- Transaktionsbasierter Schutz: Entweder ist die Operation erfolgreich oder sie schlägt fehl mit einer Ausnahmenbehandlung. Im letzteren Fall bleibt der Stand vor dem Aufruf der Operation erhalten.
- Zusicherung von **noexcept**.

- Bereits 2000 propagierten Andrei Alexandrescu und Petru Marginean in Ihrem Paper die Idee, dass spezielle RAII-Objekte dazu genutzt werden könnten, um ein *rollback* im Falle einer abgebrochenen Transaktion durchzuführen.
- Die speziellen RAII-Objekte wurden *scope guards* genannt, die die Aufgabe hatten, die Rollback-Aktionen zu übernehmen, wenn die Transaktion abgebrochen wird.
- Die Realisierung ist recht einfach: Ein *scope guard* führt den Rollback beim Abbau aus, es sei denn, es wurde zuvor die Methode *commit* aufgerufen.

```
template<typename T>
bool add(Database<T>& db1, Database<T>& db2, T object) {
   try {
      db1.add(object);
      auto guard = make_guard([&]() { db1.remove(object); });
      db2.add(object);
      guard.commit();
      return true;
   } catch (...) {
      return false;
   }
}
```

- Die Funktion *add* soll transaktionsbasiert ein Objekt in zwei Datenbanken einfügen.
- Entsprechend der Transaktion soll entweder beide Operationen durchgeführt werden oder keine davon.
- Die Funktion *make_guard* erzeugt hier einen *scope guard*, der den angegebenen Lambda-Ausdruck ausführt, wenn nicht zuvor *commit* aufgerufen wurde.

Einfache Implementierung eines *scope guards* 355

ScopeGuard.hpp

```
template<typename T>
class ScopeGuard {
   public:
      ScopeGuard(T rollback) : rollback(std::move(rollback)) {
      }
      ~ScopeGuard() {
         if (!committed) rollback();
      }
      void commit() {
         committed = true;
      }
   private:
      T rollback;
      bool committed = false;
};
```

- Der Template-Typparameter T ist der des Funktionsobjekts, der beim Abbau aufzurufen ist, wenn nicht zuvor *commit* aufgerufen wurde.
- In dieser Form ist es unhandlich zu benutzen. Deswegen kommt noch eine Template-Funktion *make_guard* hinzu...

ScopeGuard.hpp

```
#include <utility>

/* ... */

template<typename T>
ScopeGuard<T> make_guard(T&& rollback) {
   return ScopeGuard<T>(std::forward<T>(rollback));
}
```

- Um auf das unhandliche Spezifizieren des Typparameters verzichten zu können, hilft es, eine Template-Funktion hinzuzufügen, die den Typ automatisch ableitet.
- Wenn *T&&* für einen Template-Typparameter *T* spezifiziert wird, werden automatisch beide Fälle unterstützt: *lvalue reference* und *rvalue reference*.
- Um einen Parameter bedingt per *std*::*move* weiterzureichen, falls es sich um eine *rvalue reference* handelte, gibt es *std*::*forward*. Diese Technik nennt sich *perfect forwarding*, d.h. entweder geben wir die *lvalue reference* weiter oder Verschieben es mit *std*::*move*.

ScopeGuard.hpp

```
template<typename T>
class ScopeGuard {
   public:
      ScopeGuard(T rollback) : rollback(std::move(rollback)) {
      }
      ~ScopeGuard() {
         if (std::uncaught_exception()) rollback();
      }
   private:
      T rollback;
};
```

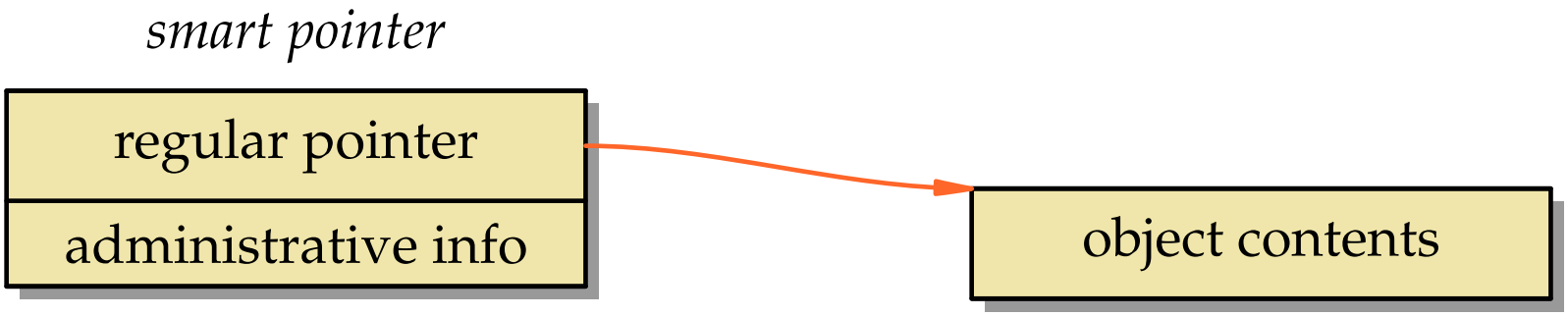
- Lässt sich feststellen, ob der Stack gerade im Rahmen einer Ausnahmenbehandlung abgebaut wird?
- Ja, das geht mit der Funktion *std::uncaught_exception*, die mit C++11 eingeführt wurde.
- Idee: Auf *commit* verzichten und *rollback* nur ausführen, wenn gerade eine Ausnahmenbehandlung läuft.

Bessere Lösung (ab C++17)

ScopeGuard.hpp

```
template<typename T>
class ScopeGuard {
   public:
      ScopeGuard(T rollback) :
         rollback(std::move(rollback)),
         exceptions(std::uncaught_exceptions()) {
      }
      ~ScopeGuard() {
         if (std::uncaught_exceptions() > exceptions) rollback();
      }
   private:
      T rollback;
      int exceptions;
};
```

- Problem: Auch während einer laufenden Ausnahmenbehandlung können Objekte (wie *scope guards*) regulär angelegt und abgebaut werden.
- Beginnend ab C++17 gibt es *std::uncaught_exceptions*, die die Zahl der gerade laufenden Stack-Abbauten liefert.
- Jetzt wird *rollback* nur dann aufgerufen, wenn der *scope guard* in Folge einer Ausnahmenbehandlung abgebaut wird.



- Intelligente Zeiger (*smart pointers*) entsprechen 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.
- Sie unterstützen das RAII-Prinzip für Zeiger.

- 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 gemeinsames 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 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.
- Normalerweise sollten entsprechend des RAII-Prinzips „nackte“ Zeiger außerhalb isolierter Fälle konsequent vermieden werden.

- 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

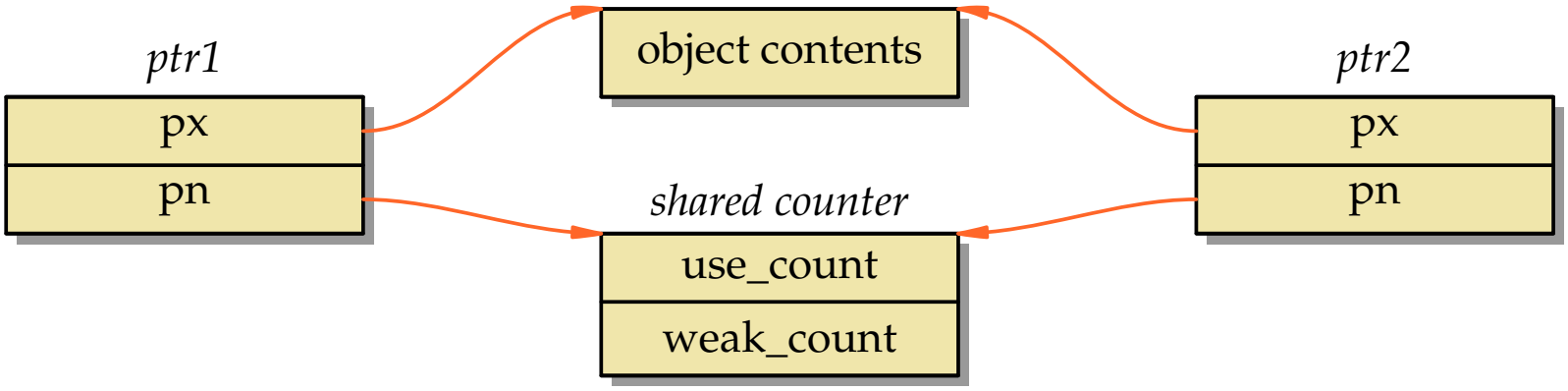
```
void f(int i) {
   Object* ptr = new Object(i);
   if (i == 2) {
      throw something();
   }
   delete ptr;
}
```

- 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.

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

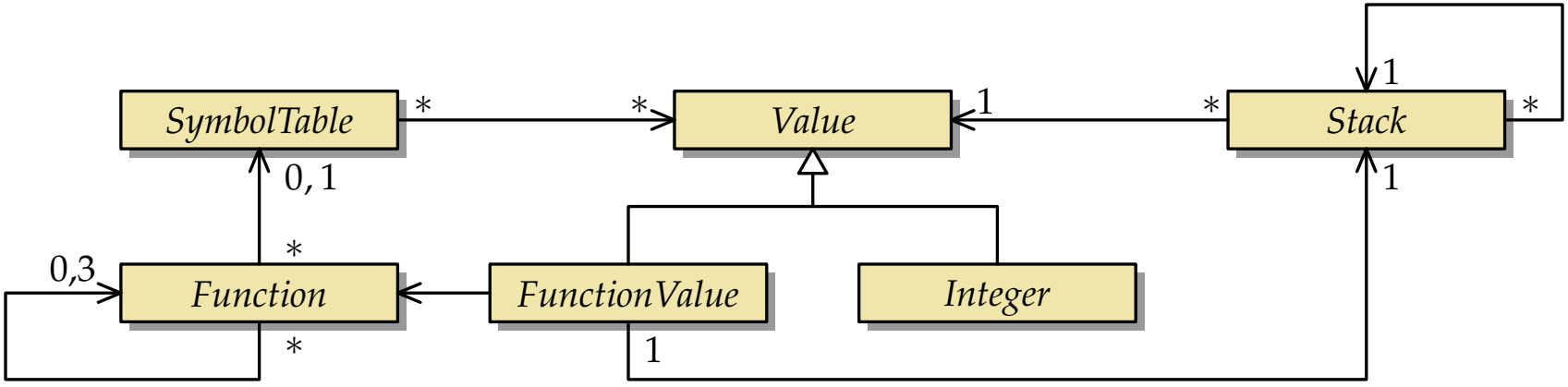
- *ptr* kann hier wie ein normaler Zeiger verwendet werden, abgesehen davon, dass eine Zuweisung an einen anderen Zeiger nicht zulässig ist.
- Dann erfolgt die Freigabe des Objekts vollautomatisch über den Dekonstruktor.
- Beginnend ab C++14 gibt es die Funktion *std::make_unique*, die im Vergleich zur früheren Variante *exception safeness* anbietet:

```
void f(int i) {
   auto ptr = std::make_unique<Object>(i);
   if (i == 2) {
      throw something();
   }
}
```



- 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.
- Das Zählerobjekt wird erst freigegeben, wenn neben *use_count* auch *weak_count* auf 0 sinkt.
- Es ist darauf zu achten, dass für jedes Objekt nur ein gemeinsames Zählerobjekt existiert.

Beispiel: Kleine Sprache mit Lambda-Ausdrücken 367



- Im 7. Übungsblatt hatten wir eine kleine Sprache mit λ-Ausdrücken.
- Die zugehörige Datenstruktur hat den varianten Datentyp *Value*. Funktionswerte (*FunctionValue*) bestehen aus einem Zeiger auf den Kaktusstack (*closure*) und einen Zeiger auf das konstruierte Funktionsobjekt, das (im Falle einer *if*-Anweisung) auf bis zu drei weitere Funktionsobjekte verweisen kann.
- Die Datenstruktur ist komplex und es lässt sich nicht trivial feststellen, wann der letzte Verweis auf ein Objekt wegfällt.

types.hpp

```
class Value;
using ValuePtr = std::shared_ptr<Value>;

class Stack;
using StackPtr = std::shared_ptr<Stack>;

using Function = std::function<ValuePtr(StackPtr)>;
using FunctionPtr = std::shared_ptr<Function>;
```

- Bei zyklischen Typreferenzen ist es sinnvoll, alle Klassen zuerst zu deklarieren.
- Dann können auch sofort die entsprechenden intelligenten Zeigertypen definiert werden.

stack.hpp

```
class Stack {
   public:
      Stack(StackPtr next, ValuePtr value) :
         next(next), value(value), len(next? next->len+1: 1) {
      }
      ValuePtr operator[](unsigned int index) const {
         assert(index < len);
         if (index == 0) return value;
         return (*next)[index-1];
      }
   private:
      ValuePtr value;
      StackPtr next;
      unsigned int len;
};
```

- Statt *Value*∗ oder *Stack*∗ wird hier konsequent *ValuePtr* bzw. *StackPtr* eingesetzt.

value.hpp

```
class Value {
   public: virtual ~Value() {};
};

class Integer: public Value { /* ... */ };
using IntegerPtr = std::shared_ptr<Integer>;

class FunctionValue : public Value { /* ... */ };
using FunctionValuePtr = std::shared_ptr<FunctionValue>;
```

- Statt der varianten Klasse könnte die Implementierung auch eine Typenhierarchie vorsehen.
- Die Kompatibilität innerhalb der *Value*-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

```
FunctionPtr Parser::parseExpression() {
   // ...
   // expr --> integer
   if (getToken().symbol == Token::INTEGER) {
      int integer = getToken().integer;
      nextToken();
      return std::make_shared<Function>([=] (StackPtr sp) {
         return std::make_shared<Integer>(integer);
      });
   }
   // ...
}
```

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

lambda.cpp

```
FunctionPtr f;
while (f = parser.getFunction()) {
   ValuePtr value = (*f)(nullptr);
   auto intval = std::dynamic_pointer_cast<Integer>(value);
   if (intval) {
      std::cout << intval->get_integer() << std::endl;
   }
}
```

- Statt **dynamic_cast** ist bei intelligenten Zeigern *dynamic_pointer_cast* zu verwenden, um eine ungewollte Neu-Erzeugung eines Zählerobjekts zu vermeiden.
- Genauso wie bei **dynamic_cast** wird ein Nullzeiger geliefert, falls der angegebene Zeiger nicht den passenden Typ hat.

```
theon$ cd lambda
theon$ time lambda <primes.lambda
541

real    0m0.330s
user    0m0.322s
sys     0m0.004s
theon$ cd ../lambda-hier
theon$ time lambda <primes.lambda
541

real    0m0.371s
user    0m0.359s
sys     0m0.006s
theon$
```

- Genauso wie **dynamic_cast** ist auch *std*::*dynamic_pointer_cast* nicht ohne Kosten.
- Variante Klassen können daher von Vorteil sein, wenn klar ist, dass eine Erweiterung der Vielfalt nicht vorgesehen ist.
- Variante Objekte benötigen aber mehr Speicher, da immer das Maximum zum Zuge kommt. Im Beispiel: 122 MB vs. 110 MB.

- 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;
      using Link = std::shared_ptr<Element>;
      using WeakLink = std::weak_ptr<Element>;
      struct Element {
         Element(const T& elem) : elem(elem) { }
         T elem;
         Link next;
         WeakLink prev;
      };
      Link head;
      Link tail;
   public:
      class Iterator {
         // ...
      };
      // ...
};
```

Beispiel: Doppelt verkettete Liste

list.hpp

```
using Link = std::shared_ptr<Element>;
using WeakLink = std::weak_ptr<Element>;
struct Element {
   Element(const T& elem) : elem(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.hpp

```
void 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 eines *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 {
         // ...
      };
      bool valid() const { /* ... */ }
      T& operator*() { /* ... */ }
      Iterator& operator++() { /* ... */ }
      Iterator operator++(int) { /* ... */ }
      Iterator& operator--() { /* ... */ }
      Iterator operator--(int) { /* ... */ }
      bool operator==(const Iterator& other) { /* ... */ }
      bool operator!=(const Iterator& other) { /* ... */ }
   private:
      friend class List;
      Iterator() {}
      Iterator(WeakLink ptr) : ptr(ptr) {}
      WeakLink ptr;
};
```

list.hpp

```
T& 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, erhalten wir das Äquivalent eines **nullptr**.

Vergleich von schwachen Zeigern

list.hpp

```
bool operator==(const Iterator& other) {
   Link p1 = ptr.lock();
   Link p2 = other.ptr.lock();
   return p1 == p2;
}
bool 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;
using ObjectPtr = std::shared_ptr<Object>;
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.

```
std::shared_ptr<int[]> p(new int[10]);
p[7] = 42;
```

- *std*::*shared_array* kann nicht ohne weiteres mit Arrays verwendet werden, da sich die Operatoren **new** und **delete** jeweils davon abhängen, ob es sich um Arrays handelt oder nicht.
- Erst ab C++17 wurde *std*::*shared_array* dahingehend erweitert, dass auch Arrays unterstützt werden. Hinzugekommen ist der Index-Operator.
- Das dazu passende *std*::*make_shared* ist für C++20 geplant.

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

- Iteratoren sind eine Verallgemeinerung von Zeigern und dienen als universelle Schnittstelle für den Zugriff auf eine Sequenz von Objekten.
- Zu den Algorithmen gehören Abfragen auf Sequenzen, die die Objekte nicht verändern (diverse Suchen, Vergleiche), solche die sie verändern (Kopieren, Verschieben, Transformieren) und sonstige Operationen (Sortieren, binäre Suche, Mengen-Operationen auf sortieren Sequenzen). Die Algorithmen arbeiten allesamt mit Iteratoren und sichern wohldefinierte Komplexitäten zu unabhängig von den verwendeten Datenstrukturen.
- Container bieten eine breite Vielfalt an Datenstrukturen, um Objekte zu beherbigen. Dazu gehören u.a. Arrays, lineare Listen, sortierte balancierte binäre Bäume und Hash-Verfahren.

Implementierungstechnik	Name der Template-Klasse	
Lineare Listen	*list*	*forward_list*
Dynamische Arrays	*vector*	*string*
	deque	
Adapter	*stack*	*queue*
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. hatten.

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*)	entfernt das Element aus dem Container, auf das *it* zeigt

Methode	Beschreibung
emplace()	erwartet bei sequentiellen Containern einen Iterator und die Parameter für einen Konstruktor des Elementtyps. Das Objekt wird dann innerhalb des Containers *vor* der Position des Iterators plaziert. Bei assoziativen Containern werden nur die Parameter für den Konstruktor angegeben.
emplace_hint()	erlaubt bei sortierten assoziativen Containern die Angabe eines Iterators, der ggf. die Suche nach der richtigen Stelle vereinfacht.

Methode	Beschreibung
try_emplace()	Analog zu *emplace*() und *emplace_hint*() mit expliziter Angabe des Schlüssels bei assoziativen Containern. Wenn es bereits ein Objekt mit dem Schlüssel gibt, wird das Objekt nicht konstruiert. (Bei *emplace* wird im Konfliktfall das Objekt zuerst konstruiert und dann zerstört.)
extract()	Erlaubt das Umhängen von Objekten ohne Verschieben oder Kopieren von einem Container zu einem anderen.
merge()	Umhängen aller Objekte eines Containers in einen anderen.

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 mit Index-Überprüfung zur Laufzeit	*vector*, *deque*

Listen gibt es in zwei Varianten: *std*::*list* ist doppelt verkettet und wird überwiegend verwendet. Wenn der Speicherverbrauch minimiert werden soll, kann die einfach verkettete *std*::*forward_list* verwendet werden, die aber nicht mehr alle Vorteile der regulären Liste bietet:

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. (Beim Einfügen kann es aber Ausnahmen geben, siehe unten.)
- Geringerer Speicherverbrauch, weil es keinen Overhead für einzelne Elemente gibt.
- Cache-freundlich, da die Elemente des Vektors zusammenhängend im Speicher liegen.

Nachteile:

- Da der belegte Speicher zusammenhängend ist, kann eine Vergrößerung eines Vektors zu einer Umkopieraktion führen mit linearem Aufwand.
- Weder *push_front* noch *pop_front* werden unterstützt.

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).
- Erheblich erhöhter Speicheraufwand im Vergleich zu einem Vektor, da es sich letztlich um einen Vektor von Vektoren handelt.
- Der indizierte Aufwand ist zwar konstant, hat aber eine Indirektion mehr als beim Vektor.
- Eine Deque ist somit ineffizienter als ein Vektor oder eine Liste auf deren jeweiligen Paradedisziplinen. Sie ist nur sinnvoll, wenn der indizierte Zugriff und beidseitiges Einfügen und Löschen wichtig sind.

std::*queue* und *std*::*stack* basieren auf einem anderen Container-Typ (zweiter Template-Parameter, *std*::*deque* per Voreinstellung) und bieten dann nur die entsprechende Funktionalität an:

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 *std*::*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.

- 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: *std*::*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 *std*::*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 „$\hat{}$“-Operator verknüpft werden.

Persons.cpp

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

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

Persons.cpp

```
auto hash = [](const Name& name) {
   return std::hash<std::string>()(name.first) ^
      std::hash<std::string>()(name.last);
};
```

- *hash*<*std*::*string*>() erzeugt ein temporäres Hash-Funktionsobjekt, das einen Funktions-Operator mit einem Parameter (vom Typ *std*::*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() {
   auto hash = [](const Name& name) { /* ... */ };
   std::unordered_map<Name, std::string, decltype(hash)> address(32,
      hash);
   address[Name("Marie", "Maier")] = "Ulm";
   address[Name("Hans", "Schmidt")] = "Neu-Ulm";
   address[Name("Heike", "Vogel")] = "Geislingen";
   std::string first; std::string last;
   while (std::cin >> first >> last) {
      auto it = address.find(Name(first, last));
      if (it != address.end()) {
         std::cout << it->second << std::endl;
      } else {
         std::cout << "Not found." << std::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:
      TemplateTester();
      TemplateTester(const TemplateTester& orig);
      TemplateTester(const BaseType& val);
      TemplateTester(TemplateTester&& orig);
      TemplateTester(BaseType&& val);
      ~TemplateTester();

      TemplateTester& operator=(const TemplateTester& orig);
      TemplateTester& operator=(const BaseType& val);
      TemplateTester& operator=(TemplateTester&& orig);
      TemplateTester& operator=(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<typename 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;
}
```

```
TemplateTester: CREATE #0 (constructor with parameter 0)
TemplateTester: CREATE #1 (move constructor of 0)
TemplateTester: DELETE #0
TemplateTester: CREATE #2 (constructor with parameter 1)
TemplateTester: CREATE #3 (move constructor of 2)
TemplateTester: DELETE #2
TemplateTester: CONVERT #1 to 0
TemplateTester: CONVERT #3 to 1
TemplateTester: DELETE #1
TemplateTester: DELETE #3
clonmel$
```

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;
}
```

```
clonmel$ TestVector >/dev/null
TemplateTester: CREATE #0 (default constructor)
TemplateTester: CREATE #1 (default constructor)
TemplateTester: ASSIGN value 0 to #0
TemplateTester: ASSIGN value 1 to #1
TemplateTester: CONVERT #0 to 0
TemplateTester: CONVERT #1 to 1
TemplateTester: DELETE #0
TemplateTester: DELETE #1
clonmel$
```

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;
}
```

```
clonmel$ TestMap >/dev/null
TemplateTester: CREATE #0 (default constructor)
TemplateTester: ASSIGN value 0 to #0
TemplateTester: CREATE #1 (default constructor)
TemplateTester: ASSIGN value 1 to #1
TemplateTester: CONVERT #0 to 0
TemplateTester: CONVERT #1 to 1
TemplateTester: DELETE #1
TemplateTester: DELETE #0
clonmel$
```

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;
}
```

```
clonmel$ TestMapIndex >/dev/null
TemplateTester: CREATE #0 (constructor with parameter 0)
TemplateTester: CREATE #1 (move constructor of 0)
TemplateTester: DELETE #0
TemplateTester: CREATE #2 (constructor with parameter 1)
TemplateTester: COMPARE #1 with #2
TemplateTester: CREATE #3 (move constructor of 2)
TemplateTester: COMPARE #1 with #3
TemplateTester: COMPARE #3 with #1
TemplateTester: DELETE #2
TemplateTester: DELETE #3
TemplateTester: DELETE #1
clonmel$
```

- Die Algorithmen der STL sind über **#include** <algorithm> zugänglich.
- Die Algorithmen arbeiten alle auf Sequenzen, die mit Iteratoren spezifiziert werden.
- Sie unterteilen sich in
  - nicht-modifizierende Algorithmen auf Sequenzen
  - Algorithmen, die Sequenzen verändern und
  - weitere Algorithmen, wie Sortieren, binäre Suche, Mengen-Operationen, Heap-Operationen und die Erzeugung von Permutationen.

Der Standard legt folgende Komplexitäten fest. Hierbei ist n normalerweise die Länge der Sequenz.

$O(1)$	*swap*(), *iter_swap*()
$O(\log n)$	*lower_bound*(), *upper_bound*(), *equal_range*(), *binary_search*(), *push_heap*(), *pop_heap*()
$O(n \log n)$	*inplace_merge*(), *stable_partition*(), *sort*(), *stable_sort*(), *partial_sort*(), *partial_sort_copy*(), *sort_heap*()
$O(n^2)$	*find_end*(), *find_first_of*(), *search*(), *search_n*()
$O(n)$	alle anderen Funktionen

(Siehe Abschnitt 25 im Standard und 32.3.1 bei Stroustrup.)

```
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** zu arbeiten, weil die hier angegebenen Ausdrücke zur Übersetzzeit berechnet werden müssen.
- Da **constexpr** erst mit C++11 eingeführt wurde, wurde früher auf **enum** zurückgegriffen. Auch einfache Funktionen mit einer **return**-Anweisung können mit **constexpr** deklariert werden.
- Beginnend mit C++14 sind lokale Variablen und Schleifen in **constexpr**-Funktionen zugelassen.

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

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

- Klassen-Templates können teilweise oder vollständig spezialisiert werden. (Bei Template-Funktionen ist das nicht sinnvoll, da das im Konflikt zum Überladen ist.)
- In jedem Fall müssen sie dann als Template deklariert werden, selbst wenn wie hier die Template-Parameter-Liste leer ist.
- Hinter dem deklarierten Namen (hier *Fibonacci*) werden dann alle Template-Parameter aufgezählt, die dann fest vorgegeben werden können bzw. von den verbliebenen Template-Parametern abhängen können.
- Vollständig oder partiell spezialisierte Templates ermöglichen es, eine Rekursion bei Templates zu beenden.

Rekursion mit using

```
template<std::size_t N>
struct Counter {
   using next = Counter<N+1>;
   static constexpr std::size_t value = N;
};
```

- Die Rekursion kann auch indirekt erfolgen mit Hilfe einer **using** bzw. **typedef**-Deklaration, bei der auf eine andere Template-Instanz verwiesen wird.
- Der Übersetzer wertet das nur aus, wo dies notwendig ist. Somit haben wir hier keine Endlos-Rekursion.

```
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<std::extent<T>::value,
      typename std::remove_extent<T>::type>::result(a);
}

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

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

for_values.hpp

```
template<typename Body, typename Value>
inline auto for_values(Body body, Value value)
      -> decltype(body(value)) {
   return body(value);
}

template<typename Body, typename Value, typename... Values>
inline auto for_values(Body body, Value value, Values... values)
      -> decltype(body(value)) {
   return body(value), for_values(body, values...);
}
```

- Beginnend mit C++11 werden auch Templates mit variabel langen Parameterlisten unterstützt. Dies geht hier unter Verwendung von *Values... values*.

```
for_values([](unsigned int i) {
   std::cout << i << std::endl;
}, 4, 6, 7);
```

for_values.hpp

```
template<typename Body, typename Value>
inline auto for_values(Body body, Value value)
      -> decltype(body(value)) {
   return body(value);
}

template<typename Body, typename Value, typename... Values>
inline auto for_values(Body body, Value value, Values... values)
      -> decltype(body(value)) {
   return body(value), for_values(body, values...);
}
```

- Diese Template-Funktion ist rekursiv organisiert, wobei die Rekursion zur Übersetzzeit aufgelöst wird.
- Der erste Fall dient dem Ende der Rekursion, *body* wird hier nur für einen einzigen Wert *value* aufgerufen.
- Der zweite Fall ist für den Induktionsschritt. Der erste Wert wird herausgegriffen, *body* dafür aufgerufen und der Rest der Rekursion überlassen.

Variable Zahl von Template-Parametern

for_values.hpp

```
template<typename Body, typename Value>
inline auto for_values(Body body, Value value)
      -> decltype(body(value)) {
   return body(value);
}

template<typename Body, typename Value, typename... Values>
inline auto for_values(Body body, Value value, Values... values)
      -> decltype(body(value)) {
   return body(value), for_values(body, values...);
}
```

- Im Kontext eines Templates können auch Funktionen variable Zahlen von Argumenten haben. Diese ist aber nur zur Übersetzzeit variabel.
- Für jede vorkommende Parameterzahl wird eine entsprechende Funktion erzeugt. Normalerweise wird das alles aber per **inline** zur Übersetzzeit aufgelöst.
- Am Ende bleibt hier nur eine entsprechende Sequenz übrig.

```
        movl    $4, %eax
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        subl    $4, %esp
        call    _ZZ4mainENKUljE_clEj.isra.0
        movl    $6, %eax
        call    _ZZ4mainENKUljE_clEj.isra.0
        movl    $7, %eax
        call    _ZZ4mainENKUljE_clEj.isra.0
        addl    $4, %esp
        xorl    %eax, %eax
        popl    %ecx
        popl    %ebp
```

- Dies ist der von *g++* erzeugte Assemblertext für den Aufruf von *for_values* mit den Werten 4, 6 und 7.
- Beim Label *_ZZ4mainENKUljE_clEj.isra.0* ist der Programmtext des Lambda-Ausdrucks. Der erste Aufruf ist noch etwas aufwendiger, da der Stack vorbereitet werden muss. Die weiteren Aufrufe sind aber vereinfacht.

```
for_values([](auto value) {
   std::cout << value << std::endl;
}, 4, "Huhu", 9.3);
```

- Ab C++14 dürfen auch Lambda-Ausdrücke polymorph sein.
- Entsprechend darf hier der *value*-Parameter **auto** deklariert werden, so dass dieser jeweils von jedem der Argumente individuell abgeleitet werden kann.
- Nun zahlt sich aus, dass die *for_values*-Template-Funktion nicht auf einheitlichen Parametertypen bei dem Aufruf von *body* besteht.

```
template<unsigned int... Is> struct seq {
   using next = seq<Is..., sizeof...(Is)>;
};
template<unsigned int N> struct gen_seq {
   using type = typename gen_seq<N-1>::type::next;
};
template<> struct gen_seq<0> {
   using type = seq<>;
};
template<unsigned int N>
using make_seq = typename gen_seq<N>::type;
```

- Bei der Metaprogrammierung kann es sinnvoll sein, mit dynamischen Listen zu hantieren. Hierfür bieten sich variabel lange Template-Parameterlisten an.
- **sizeof**...(*Is*) liefert die Zahl der Elemente des Template-Parameter-Packs *Is*.
- Das Konstrukt dient dazu Template-Parameterlisten mit den Zahlen 0 bis zu einem gewünschten Limit aufzubauen.

```
template<typename F>
void process_values(F&& f) {
}
template<typename F, typename Arg, typename... Args>
void process_values(F&& f, Arg arg, Args... args) {
   f(arg); process_values(f, args...);
}
template<typename F, unsigned int... Is>
void do_values(F&& f, seq<Is...>) {
   process_values(std::forward<F>(f), Is...);
}

int main() {
   do_values([](unsigned int i) -> void {
      std::cout << " " << i;
   }, make_seq<20>());
   std::cout << std::endl;
}
```

- Mit Hilfe einer Template-Funktion lässt sich dann die Sequenz extrahieren, um z.B. sie in eine Parameterliste zu verwandeln.

```
template<unsigned int... Is>
constexpr auto make_array(seq<Is...>) ->
      std::array<unsigned int, sizeof...(Is)> {
   return {Is...};
}

auto values = make_array(make_seq<20>());

int main() {
   for (auto val: values) {
      std::cout << " " << val;
   }
   std::cout << std::endl;
}
```

- Alternativ kann das auch genutzt werden, um damit ein Array zu initialisieren.
- Man beachte, dass das Array global ist und bereits zur Übersetzzeit vollständig mit Werten gefüllt wird.

```
template<unsigned int... Is>
constexpr auto make_array_of_squares(seq<Is...>) ->
      std::array<unsigned int, sizeof...(Is)> {
   return {Is * Is...};
}

auto squares = make_array_of_squares(make_seq<20>());

int main() {
   for (auto val: squares) {
      std::cout << " " << val;
   }
   std::cout << std::endl;
}
```

- Wenn sogenannte *template parameter packs* mit Hilfe von ... expandiert werden, kann auch ein Konstrukt angegeben werden, das für jeden einzelnen Parameter expandiert wird und – durch Kommata getrennt – zusammengefügt wird.

```
template<typename Map, unsigned int... Is>
constexpr auto make_array(Map&& map, seq<Is...>) ->
      std::array<unsigned int, sizeof...(Is)> {
   return {map(Is)...};
}

auto squares = make_array([](unsigned int val) constexpr {
   return val * val;
}, make_seq<20>());

int main() {
   for (auto val: squares) {
      std::cout << " " << val;
   }
   std::cout << std::endl;
}
```

- Geht dies auch mit **constexpr**-Lambda-Ausdrücken? Ja, ab C++17!

```
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 abgeleitete 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 als Alternative zu *std*::*enable_if*. (Technik von Abrahams und Gurtovoy.)

Allzu leicht geraten Template-Funktionen zu allgemein:

```
template <typename Alpha, typename Matrix>
void scale(const Alpha& alpha, Matrix& A) {
   // scale a matrix
}

template <typename Alpha, typename Vector>
void scale(const Alpha& alpha, Vector& x) {
   // scale a vector
}
```

Hier stehen beide Definition zueinander in Konflikt. Wie lässt sich dieser lösen?

```
template <typename Alpha, typename Matrix>
typename std::enable_if<IsMatrix<Matrix>::value, void>::type
void scale(const Alpha& alpha, Matrix& A) {
   // scale a matrix
}

template <typename Alpha, typename Vector>
typename std::enable_if<IsVector<Vector>::value, void>::type
void scale(const Alpha& alpha, Vector& x) {
   // scale a vector
}
```

- Mit Hilfe der SFINAE-Technik können wir den Konflikt auflösen.
- Wir müssen hier nur für jeden weitere polymorphe *Matrix*- oder *Vector*-Variante ein entsprechendes *IsMatrix*- bzw. *IsVector*-Konstrukt ergänzen.

```
template <typename Derived> struct Matrix {};
template <typename Derived> struct Vector {};

template <typename Alpha, typename Matrix>
void scale(const Alpha& alpha, Matrix<MA>& A_) {
   MA& A = static_cast<MA&>(A_);
   // scale matrix A
}

template <typename Alpha, typename Vector>
void scale(const Alpha& alpha, Vector<VX>& x_) {
   VX& x = static_cast<VX&>(x_);
   // scale vector x
}
```

- Alternativ können wir mit der von Abrahams und Gurtovoy vorgeschlagenen Technik darauf bestehen, dass alle Matrix- und Vektorklassen via CRTP von den entsprechenden Basisklassen abgeleitet werden.
- Dann lassen sich die Konflikte ohne SFINAE auflösen.

- 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/`
  – diese Version ist von 1995 und lässt sich leider mit modernen C-Übersetzern nicht mehr übersetzen. Eine neuere Fassung steht auf github zur Verfügung.

- 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
```

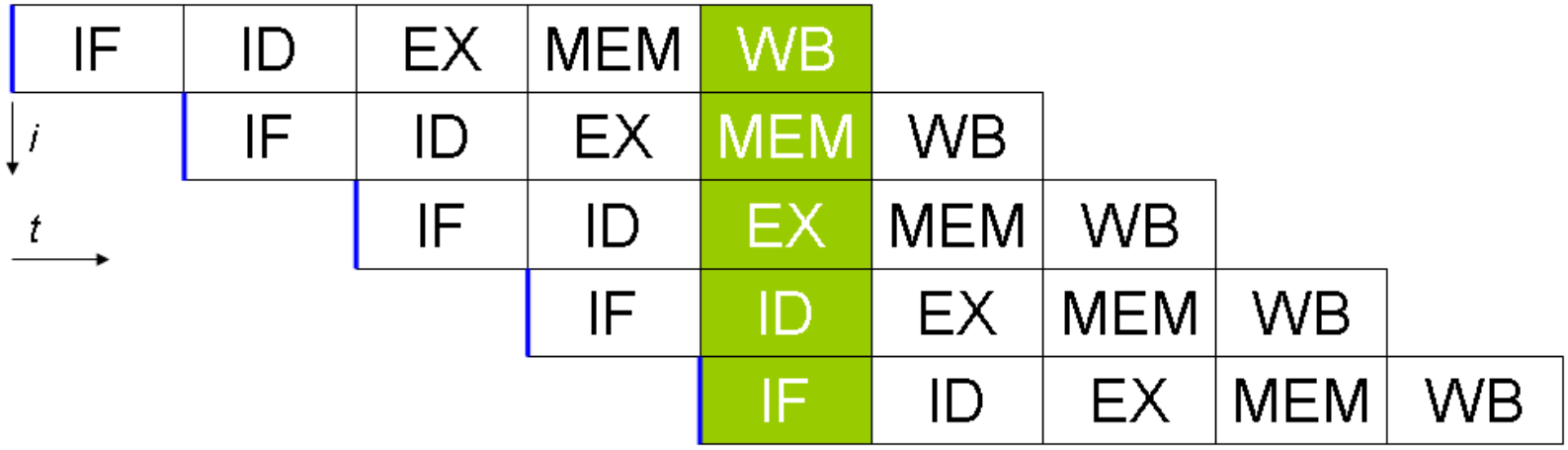
- Generell ist die Vermeidung bedingter Sprünge ein Gewinn, da diese das Pipelining erschweren.

- 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)

- 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 451

```
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 452

```
        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.

```
std::size_t 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:

```
std::size_t 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[i] == 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;
}
```

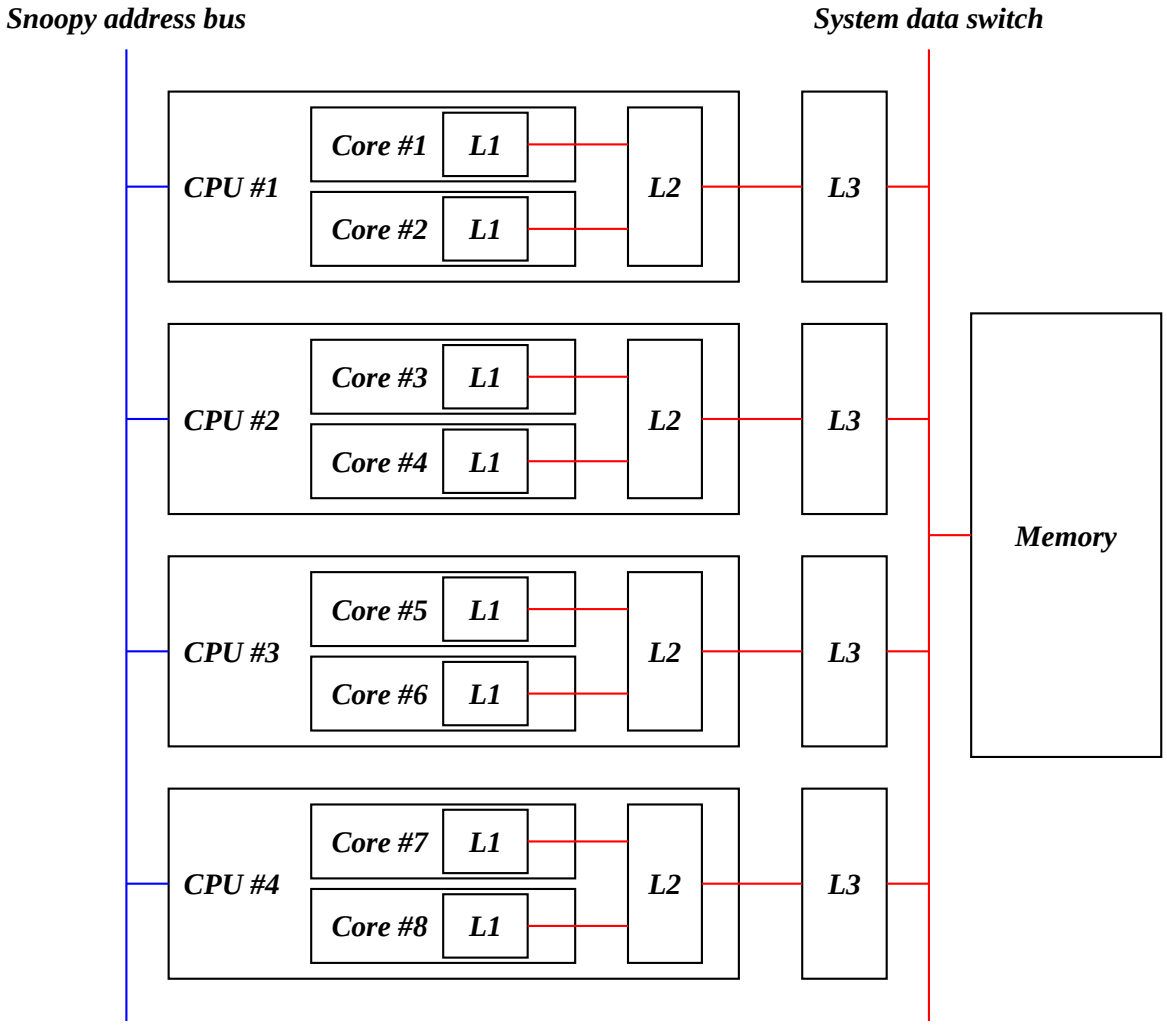
- Diese Technik reduziert deutlich die Zahl der bedingten Sprünge, indem mehrere Schleifendurchläufe in einem Zug erledigt werden.
- Das geht nur, wenn die einzelnen Schleifendurchläufe unabhängig voneinander erfolgen können, d.h. kein Schleifendurchlauf von den Ergebnissen der früheren Durchgänge abhängt.
- 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;
}
```

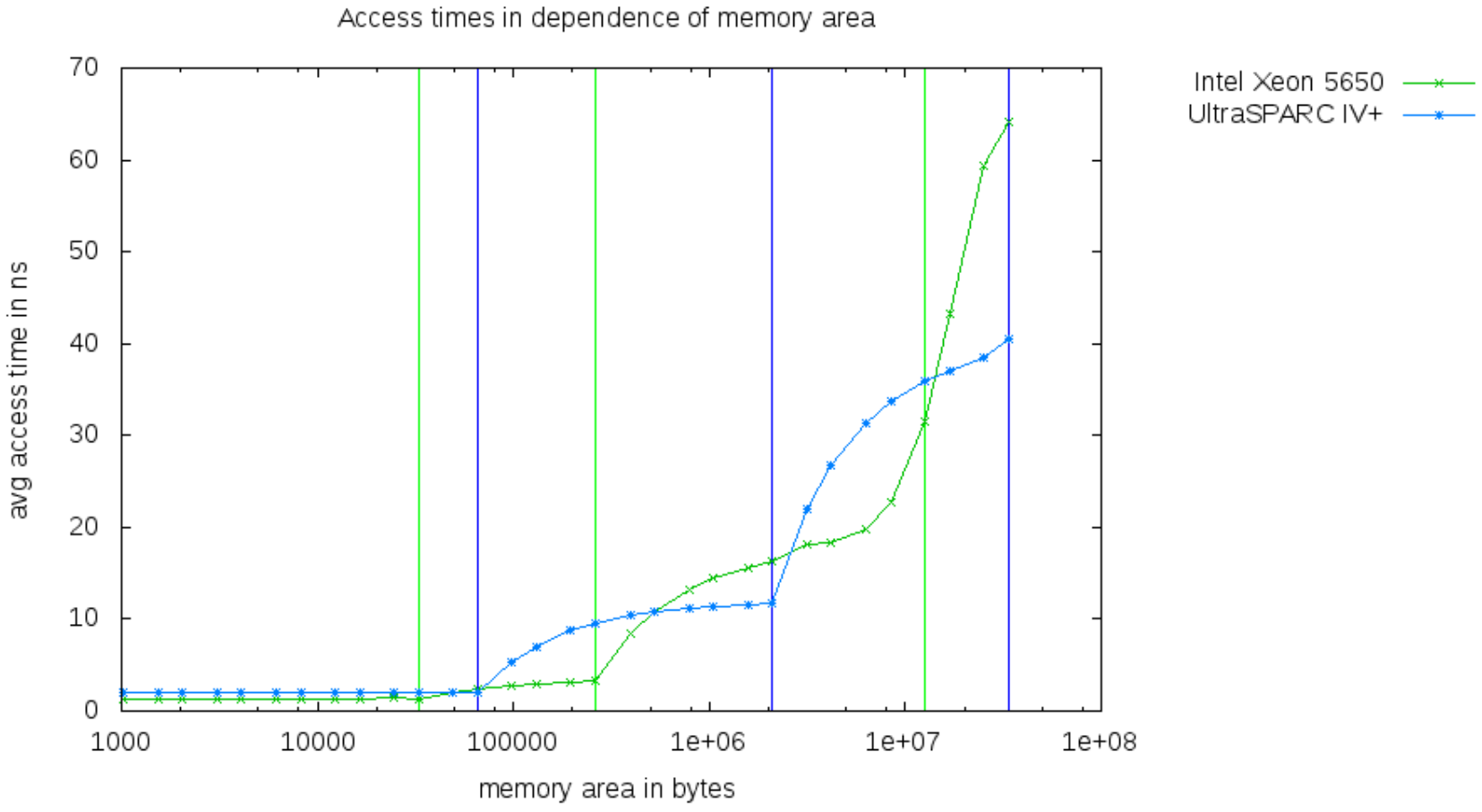
- Zugriffe einer CPU auf den primären Hauptspeicher sind vergleichsweise langsam. Obwohl Hauptspeicher generell schneller wurde, behielten die CPUs ihren Geschwindigkeitsvorsprung.
- Grundsätzlich ist Speicher direkt auf einer CPU deutlich schneller. Jedoch lässt sich Speicher auf einem CPU-Chip aus 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



Siehe `https://github.com/afborchert/pointer-chasing`

- 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.