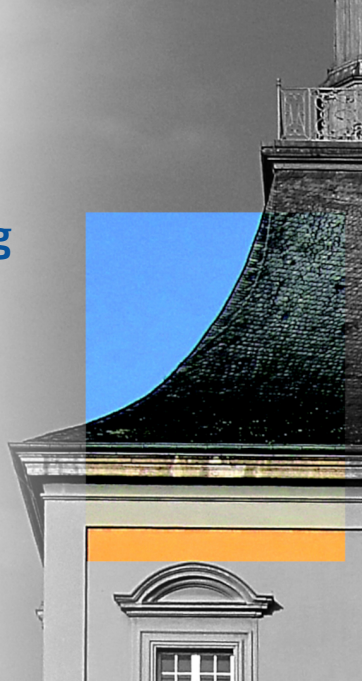
UNIVERSITÄT BONN

Algorithmen und Programmierung

Objektorientierte Programmierung

Dr. Felix Jonathan Boes
boes@cs.uni-bonn.de
Institut für Informatik



KURZE WIEDERHOLUNG

Objektorientierte Modellierung

Disclaimer

In dieser und den folgenden Vorlesungseinheiten diskutieren wir zuerst allgemeine Aspekte der objektorientierten Modellierung (z.T. in UML)

Anschließend diskutieren wir, wie diese Aspekte in einer objektorientierten Programmiersprache umgesetzt werden (hauptsächlich C++)

Objektorientierte Modellierung

Einleitung Unified Modeling Language (UML)

Zusammenfassung

Objektorientierte Modellierung wird in UML ausgedrückt

Wir haben erste UML-Vokabeln kennen gelernt

Interaktionsschnittstelle, Aggregation und Zugriffsbeschränkung

Offene Fragen

Wie werden Interaktionsschnittstellen beschrieben?

Wie wird die Objektaggregation (Objektzusammensetzung) beschrieben?

Wie werden Zugriffsbeschränkungen beschrieben?

UNIVERSITÄT BONN

Überblick

Wir beantworten diese Fragen in der vergangenen und dieser Vorlesungseinheiten. Dabei gehen wir wie folgt vor.

Objektorientierte Modellierung (mit UML)

- ✓ Interaktionsschnittstellen
- ✓ Objektaggregation
- ✓ Zugriffsbeschränkungen

Objektorientierte Programmierung (mit C++)

- ✓ Interaktionsschnittstellen, Objektaggregation, Zugriffsbeschränkungen

Technische Umsetzung im virtuellen Adressraum

- Interaktionsschnittstellen, Objektaggregation, Zugriffsbeschränkungen

Interaktionsschnittstelle, Aggregation und Zugriffsbeschränkung

Schnittstellen, Aggregation und Zugriff (ohne Vererbung) im Speicher

Offene Frage

Wir werden Schnittstellen, Aggregationen und Zugriffsspezifikation unterhalb von C++ realisiert?

UNIVERSITÄT BONN

Der Compiler

Der Compiler übersetzt C++-Projekte in ausführbare Programme. Dabei überprüft der Compiler, ob der C++-Programmcode zulässig ist.

Während der Compilezeit wird unter anderem geprüft:

- Ist die Syntax korrekt?
- Sind alle verwendeten Identifier (Variablennamen, Typnamen, ...) bekannt?
- Passen die Expressiontypen zu den angeforderten Operationen?
- Wird Const Correctness eingehalten?
- Werden alle Zugriffseinschränkungen respektiert?

Falls das Kompilieren erfolgreich ist, respektiert das zugehörige Programm die Const Correctness und die Zugriffseinschränkungen zur Laufzeit.

UNIVERSITÄT BONN

Beispiel

Im folgenden Beispiel ist die Syntax korrekt, alle Identifier bekannt, alle Expressiontypen passen zu den angeforderten Operationen, die Const Correctness ist erfüllt, aber die Zugriffseinschränkungen werden nicht vollständig respektiert.

```
class Studi {
public:
  Studi(const std::string& vorname, const std::string& nachname);

private:
  std::string vname;
  std::string nname;
};

int main() {
    Studi s("Alex", "Platz");
    s.vname = "Pascal";
    // Compiler error: 'std::string Studi::vname' is private within this context
}
```

UNIVERSITÄT BONN

Erinnerung
Virtuelle Adressräume im Allgemeinen

Auf aktuellen Endnutzerrechnern steht jedem Nuzterprozess ein eigener, virtueller Adressraum zur Verfügung. Imperative und objektorientierte Programmiersprachen organisieren Variablen in diesem virtuellen Speicher.

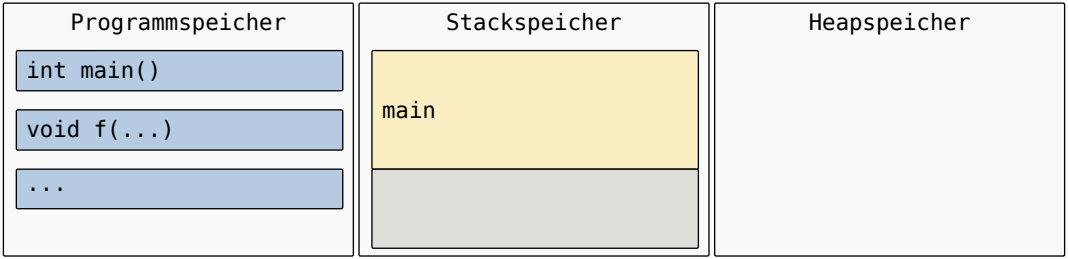
Der virtuelle Adressraum besteht vereinfacht gesagt aus **Programmspeicher**, **Stackspeicher** und **Heapspeicher**.

Der **Stackspeicher** ist funktionsorientiert. Der **Heapspeicher** wird verwendet, um Objekte **dynamisch** zu organisieren, also unabhängig vom aktuellen Stackframe.

UNIVERSITÄT BONN

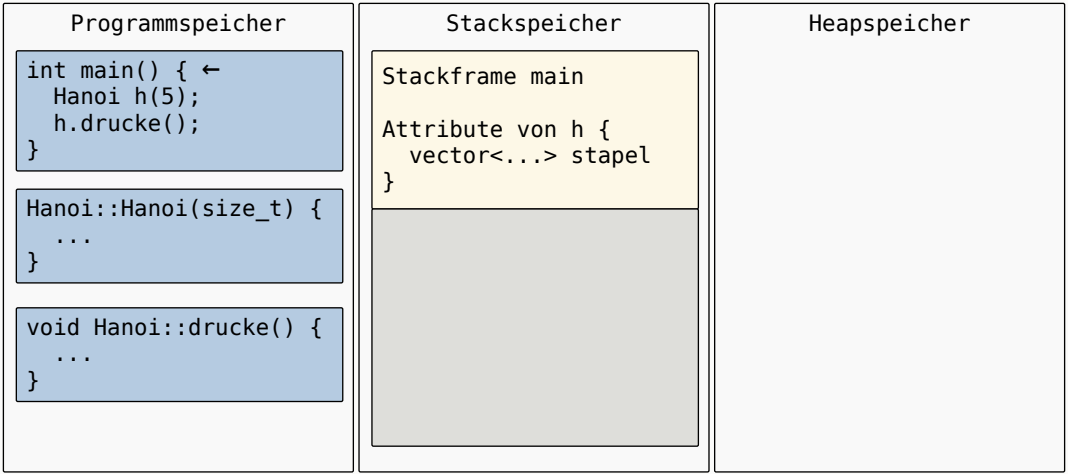
Programmausführung

Ausführbare Dateien werden beim Start in den virtuellen Adressraum geladen und dort ausgeführt. Dabei wird jede definierte (Member)funktion genau einmal in den Programmspeicher geladen. Der Heapspeicher ist zunächst leer. Der Stackspeicher enthält den Stackframe der Funktion `main`.



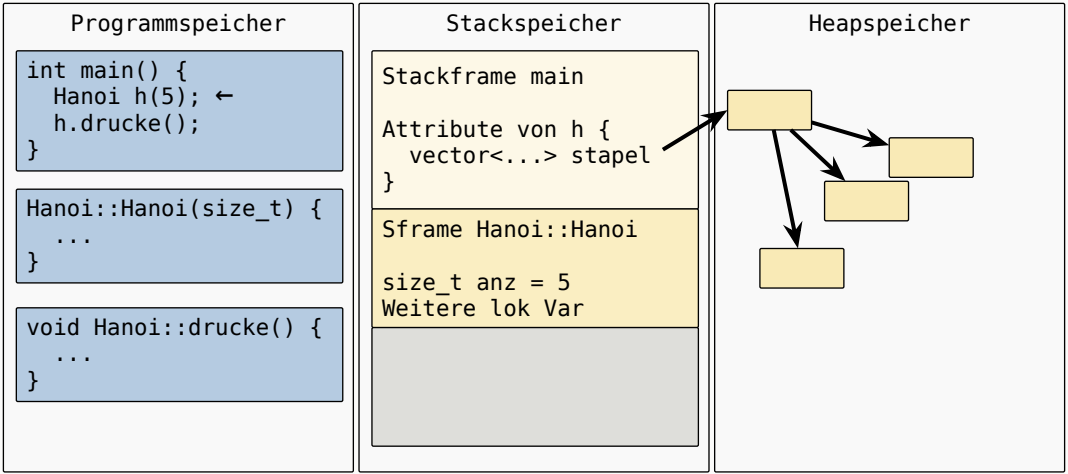
UNIVERSITÄT BONN

Beispiel



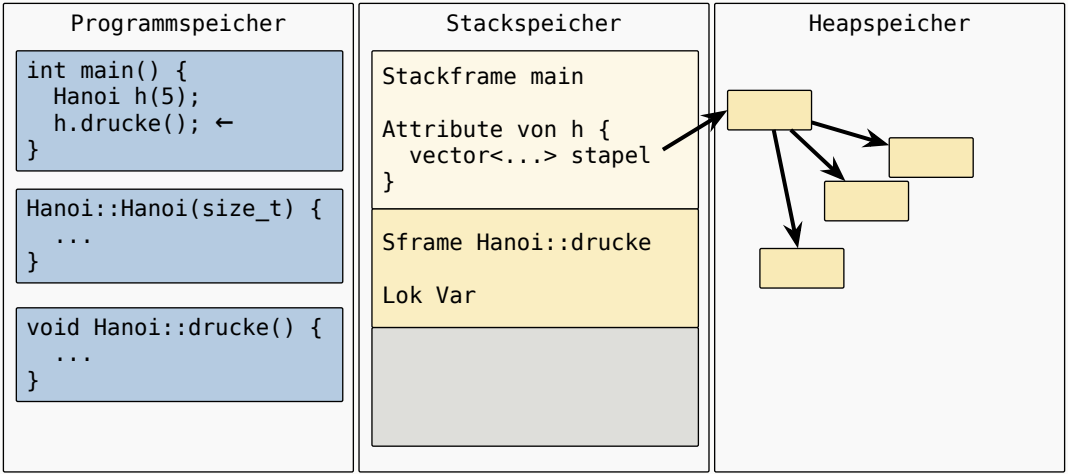
UNIVERSITÄT BONN

Beispiel



UNIVERSITÄT BONN

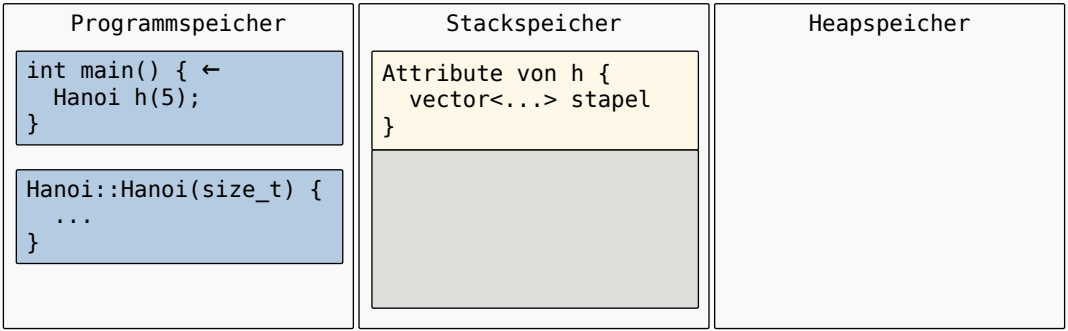
Beispiel



UNIVERSITÄT BONN

Attribute

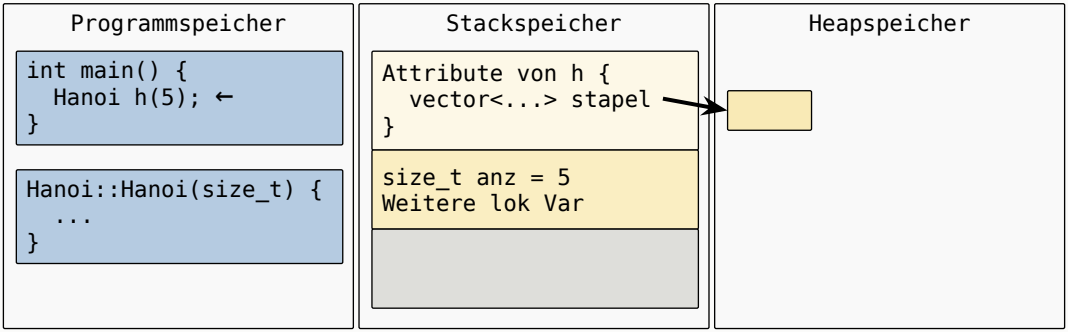
Beim Anlegen eines neuen Objekts als lokale Variable, werden die Attribute des Objekts auf dem Stackframe abgelegt. Die Attribute haben im Speicher eine feste Reihenfolge und werden als zusammenhängender Block abgelegt. Anschließend wir der Konstruktor aufgerufen. Das Anlegen eines Objekts auf dem Heap verläuft genauso.



UNIVERSITÄT BONN

Attribute

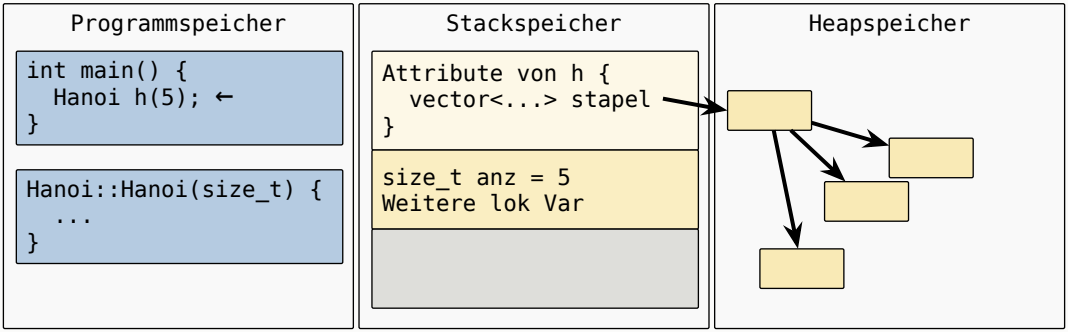
Beim Anlegen eines neuen Objekts als lokale Variable, werden die Attribute des Objekts auf dem Stackframe abgelegt. Die Attribute haben im Speicher eine feste Reihenfolge und werden als zusammenhängender Block abgelegt. Anschließend wir der Konstruktor aufgerufen. Das Anlegen eines Objekts auf dem Heap verläuft genauso.



UNIVERSITÄT BONN

Attribute

Beim Anlegen eines neuen Objekts als lokale Variable, werden die Attribute des Objekts auf dem Stackframe abgelegt. Die Attribute haben im Speicher eine feste Reihenfolge und werden als zusammenhängender Block abgelegt. Anschließend wir der Konstruktor aufgerufen. Das Anlegen eines Objekts auf dem Heap verläuft genauso.



Der Konstruktor und alle anderen Memberfunktionen sind Funktion

Zwischenfrage

Woher wissen Memberfunktionen, wo sich der zugehörige Attributblock genau befindet?

UNIVERSITÄT BONN

Implizite Instanzreferenz I

Wird bei einem Objekt `obj` eine Memberfunktion `fkt` aufgerufen, enthält die `fkt` eine **implizite Instanzreferenz** auf `obj`. Die implizite Instanzreferenz ist ein Zeiger oder eine Referenz auf `obj`. So kann innerhalb der Funktion `fkt` auf `obj` zugegriffen werden.

In Python heißt die implizite Instanzreferenz typischerweise `self`. In C++ und Java ist das implizite Instanzreferenz durch das Keyword **`this`** gegeben. In C++ ist **`this`** ein Zeiger.

Implizite Instanzreferenz II

Die implizite Instanzreferenz wird als implizites Funktionsargument realisiert.

```
// DIES IST KEIN C++ CODE

class MeineKlasse {
  // Die Memberfunktion
  void fkt(String s, int x);
  // wird technisch wie folgt umgesetzt
  void fkt(MeineKlasseReferenz this, String s, int x);
  ...
};

int main() {
  MeineKlasse obj;
  // Der Aufruf der Memberfunktion
  obj.fkt("Parameter1", 44);
  // wird technisch wie folgt umgesetzt
  MeineKlasse::fkt( ReferenzAuf(obj), "Parameter1", 44 );
}
```

UNIVERSITÄT BONN

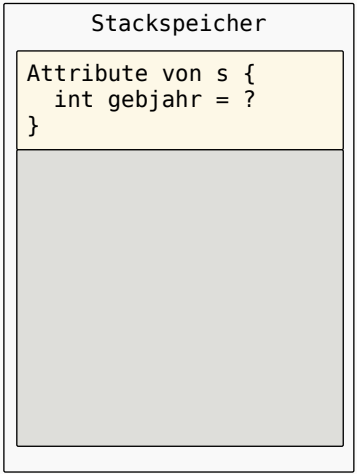
Implizite Instanzreferenz III

```
class Studi {
public:
  Studi(int geburtsjahr) :
    gebjahr(geburtsjahr)
  {}

  void drucken() {
    std::cout << this->gebjahr << std::endl;
  }

private:
  const int gebjahr;
}

int main() { ←
  Studi s(1952);
  s.drucken();
}
```



UNIVERSITÄT BONN

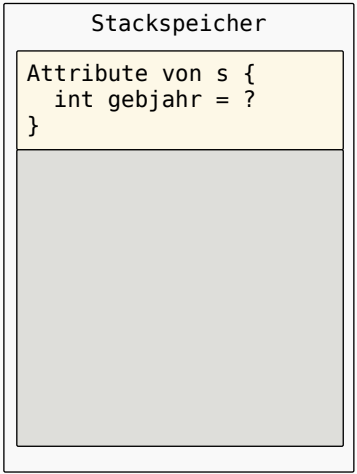
Implizite Instanzreferenz III

```
class Studi {
public:
  Studi(int geburtsjahr) :
    gebjahr(geburtsjahr)
  {}

  void drucken() {
    std::cout << this->gebjahr << std::endl;
  }

private:
  const int gebjahr;
}

int main() {
  Studi s(1952); ←
  s.drucken();
}
```



UNIVERSITÄT BONN

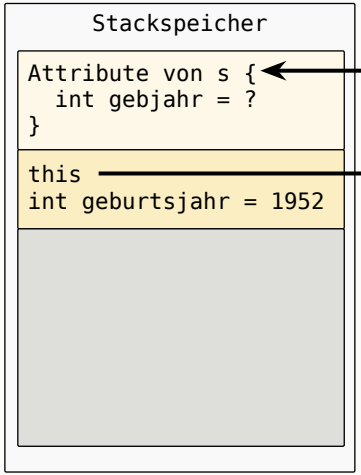
Implizite Instanzreferenz III

```
class Studi {
public:
  Studi(int geburtsjahr) : ←
    gebjahr(geburtsjahr)
  {}

  void drucken() {
    std::cout << this->gebjahr << std::endl;
  }

private:
  const int gebjahr;
}

int main() {
  Studi s(1952); ←
  s.drucken();
}
```



UNIVERSITÄT BONN

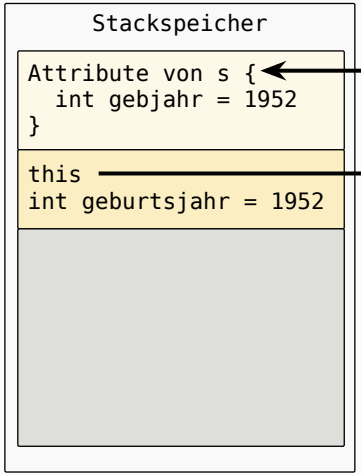
Implizite Instanzreferenz III

```
class Studi {
public:
  Studi(int geburtsjahr) :
    gebjahr(geburtsjahr)   ←
  {}

  void drucken() {
    std::cout << this->gebjahr << std::endl;
  }

private:
  const int gebjahr;
}

int main() {
  Studi s(1952); ←
  s.drucken();
}
```



UNIVERSITÄT BONN

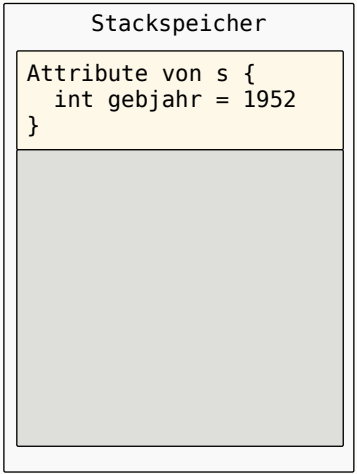
Implizite Instanzreferenz III

```
class Studi {
public:
  Studi(int geburtsjahr) :
    gebjahr(geburtsjahr)
  {}

  void drucken() {
    std::cout << this->gebjahr << std::endl;
  }

private:
  const int gebjahr;
}

int main() {
  Studi s(1952);
  s.drucken();   ←
}
```



UNIVERSITÄT BONN

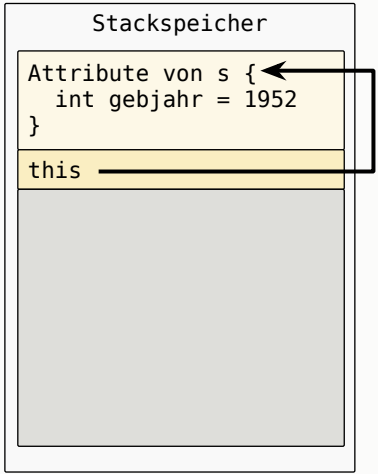
Implizite Instanzreferenz III

```
class Studi {
public:
  Studi(int geburtsjahr) :
    gebjahr(geburtsjahr)
  {}

  void drucken() { ←
    std::cout << this->gebjahr << std::endl;
  }

private:
  const int gebjahr;
}

int main() {
  Studi s(1952);
  s.drucken();   ←
}
```



UNIVERSITÄT BONN

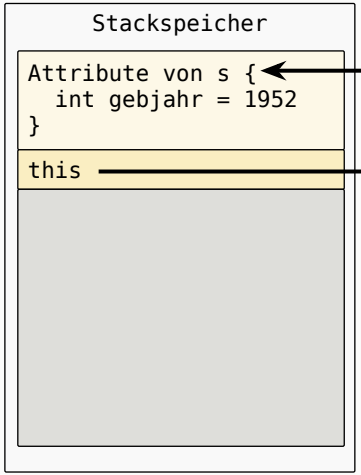
Implizite Instanzreferenz III

```
class Studi {
public:
  Studi(int geburtsjahr) :
    gebjahr(geburtsjahr)
  {}

  void drucken() {
    std::cout << this->gebjahr << std::endl; ←
  }

private:
  const int gebjahr;
}

int main() {
  Studi s(1952);
  s.drucken();   ←
}
```



UNIVERSITÄT BONN

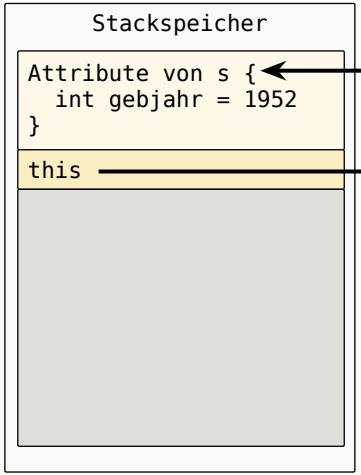
Implizite Instanzreferenz III

```
class Studi {
public:
  Studi(int geburtsjahr) :
    gebjahr(geburtsjahr)
  {}

  void drucken() {
    std::cout << this->gebjahr << std::endl;
  } ←

private:
  const int gebjahr;
}

int main() {
  Studi s(1952);
  s.drucken();   ←
}
```



UNIVERSITÄT BONN

Die implizite Instanzreferenz an sich

Innerhalb von Memberfunktion wird die implizite Instanzreferenz **`this`** verwendet, um auf Membervariablen und Memberfunktionen zuzugreifen. Dies ist auch der Fall, wenn **`this`** nicht explizit angegeben wird.

```
class Studi {
public:
  ...
  void drucken() {
    // Der Compiler findet den Identifier gebjahr
    std::cout << gebjahr << std::endl;
    // Und ersetzt diesen durch den folgenden Ausdruck
    std::cout << this->gebjahr << std::endl;
  }
  ...
}
```

Die implizite Instanzreferenz wird automatisch ergänzt (falls möglich).

Zwischenfazit

Memberfunktionen verwenden die implizite Instanzreferenz (this), um auf die Objektattribute zuzugreifen

Haben Sie Fragen?

Zusammenfassung

Wir haben gelernt, wie Interaktionsschnittstellen, Objektaggregationen und Zugriffsbeschränkungen in UML und C++ ausgedrückt werden

Wir haben gelernt, wie diese Modellierungen im virtuellen Adressraum umgesetzt werden

Haben Sie Fragen?

Vererbung

Disclaimer

In dieser und den folgenden Vorlesungseinheiten diskutieren wir zuerst allgemeine Aspekte der objektorientierten Modellierung (in UML)

Anschließend diskutieren wir, wie diese Aspekte in einer objektorientierten Programmiersprache umgesetzt werden (hauptsächlich C++)

Vererbung

Motivation

Ziel

Wir entwickeln eine Wunschliste, die durch die Prinzipien der Vererbung erfüllt werden wird

UNIVERSITÄT BONN

Motierendes Beispiel I

Wir haben die Breitensuche für allgemeine Graphen formuliert. Es scheint unnötig zu sein, die Breitensuche für Graphen mit gerichteten, gewichtete und gefärbten Kanten erneut zu formulieren.

Wunsch: Was für allgemeine Graphen algorithmisch funktioniert, soll auch für speziellere Graphen möglich sein.

Wunsch: Was für allgemeine Graphen implementiert wird, soll auch für speziellere Graphen ohne Mehraufwand verfügbar sein.

UNIVERSITÄT BONN

Motierendes Beispiel II

Zur Erkennung von Botnetzwerken in sozialen Netzwerken, erhalten wir einen gerichteten, gewichteten Graphen (aus einem gegebenen sozialem Netzwerk). Unser Verfahren zur Angriffserkennung verwendet die Kantengewichte nicht.

Wunsch: Da unser Verfahren für allgemeine Graphen algorithmisch funktioniert, soll es auch für Graphen mit Kantengewichten funktionieren.

Wunsch: Die Implementierung unseres Verfahrens arbeitet mit gerichteten Graphen ohne Kantengewicht. Es soll auch für speziellere Graphen ohne (wesentliche) Änderungen verwendebar sein.

UNIVERSITÄT BONN

Motierendes Beispiel III

Beim Verarbeiten von Daten können verschiedene Ausnahmesituationen auftreten. Zum Beispiel:

- Divison durch Null
- Arrayzugriff an nicht vorhandenem Index
- Zugriff auf einen Sharedpointer der `null` referenziert
- Netzwerkfehler beim Versenden einer Nachricht

Wunsch: Der Umgang mit Ausnahmen im Allgemeinen (z.B. eine zugehörige Textdarstellung zu erhalten) soll einmal implementiert werden. Für speziellere Ausnahmen sollen nur die „zusätzlichen Interaktionsmöglichkeiten“ implementiert werden.

UNIVERSITÄT BONN

Wunschliste

Wir wollen folgenden Wünschen näher kommen.

- Wir wollen **kompakt** ausdrücken können, in welchem **Umfang** ein Typ S **allgemeiner** oder **spezieller** ist als ein Typ T.
- Wenn S spezieller ist als T, dann sollen Objekte vom Typ S überall verwendbar sein, wo Objekte vom Typ T verwendbar sind.
- Wenn S spezieller ist als T, dann wollen wir nur der Unterschied von S zu T implementieren.

Haben Sie Fragen?

Subtypenbeziehungen

Offene Frage

Wie und in welchem Umfang drücken wir aus, dass zwei Typen miteinander verwandt sind?

Einleitung

Wir beginnen mit einem **Beispiel**: Ein gerichteter, gewichteter Graph X ist auch ein gerichteter Graph. Also sind gerichtete, gewichtete Graphen spezieller als gerichtete Graphen. Genauso sind gerichtete Graphen allgemeiner als gerichtete, gewichtete Graphen.

Wir **konkretisieren** im Folgenden die folgende **Faustregel** zur Subtypenbeziehungen:

(Subtypenbeziehungen als Faustregel)

Ein Objekt X vom Typ S ist ein **Subtyp** von einem Typ T, falls es Sinn macht zu sagen:

X ist auch ein T

In diesem Fall sagt man S ist **spezieller** als T oder T ist **allgemeiner** als S.

UNIVERSITÄT BONN

Liskovsches Substitutionsprinzip

Um **Subtypenbeziehungen** nachvollziehbar zu modellieren, verwenden wir hier das **Liskovsche Substitutionsprinzip**[1].

(Liskovsches Substitutionsprinzip (vereinfacht))

Falls der Typ S ein Subtyp vom Typ T ist, dann können Objekte vom Typ T immer durch Objekte vom Typ S ersetzt werden, ohne die gewünschten Eigenschaften eines beliebigen Programms zu verändern.

Wenn S und T das Liskovsche Substitutionsprinzip erfüllen, dann sind Objekte vom Typ S auch Objekte vom Typ T.

[1] Das Liskovsche Substitutionsprinzip ist nicht perfekt und es gibt auch andere Prinzipien um Subtypenbeziehungen zu modellieren. Allerdings hat das Liskovsche Substitutionsprinzip bei Weitem die größte, praktische Relevanz.

Beispiel

(Liskovsches Substitutionsprinzip)

Falls der Typ S ein Subtyp vom Typ T ist, dann können Objekte vom Typ T immer durch Objekte vom Typ S ersetzt werden, ohne die gewünschten Eigenschaften eines beliebigen Programms zu verändern.

Wir bemerken: Gerichtete Graphen mit gefärbten Kanten als Subtyp von gerichteten Graphen ohne gefärbte Kanten aufzufassen, ist verträglich mit dem Liskovschen Substitutionsprinzip.

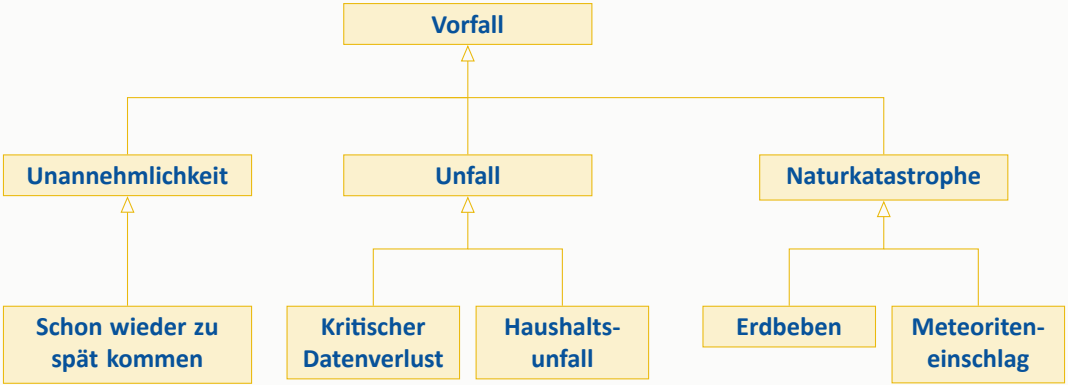
Subtypenbeziehungen in UML

In UML wird die Subtypenbeziehung durch den Pfeil ⟶▷ beschrieben. Dabei zeigt der Pfeil vom spezielleren Subtyp zum allgemeineren Obertyp.

Beim Modellieren soll das Liskovsches Substitutionsprinzip eingehalten werden. Das muss durch die Modellierer:innen selbst sichergestellt werden.

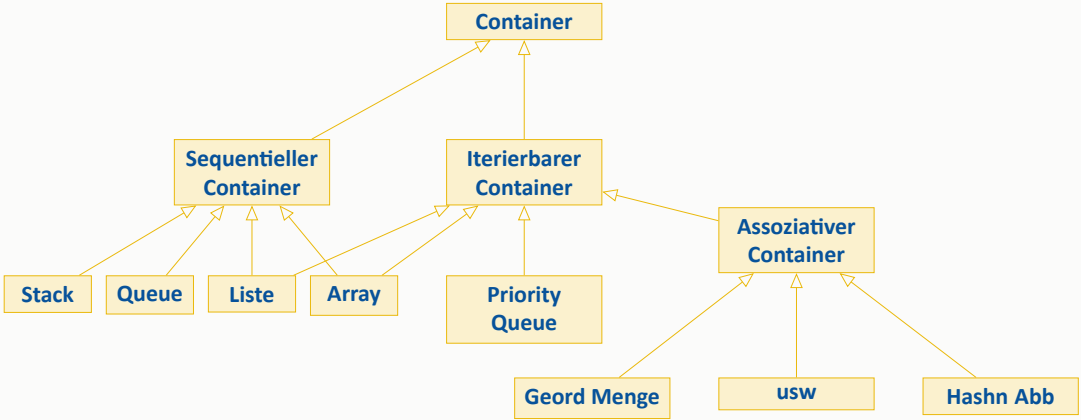
UNIVERSITÄT BONN

Beispiel Vorfälle und speziellere Vorfälle

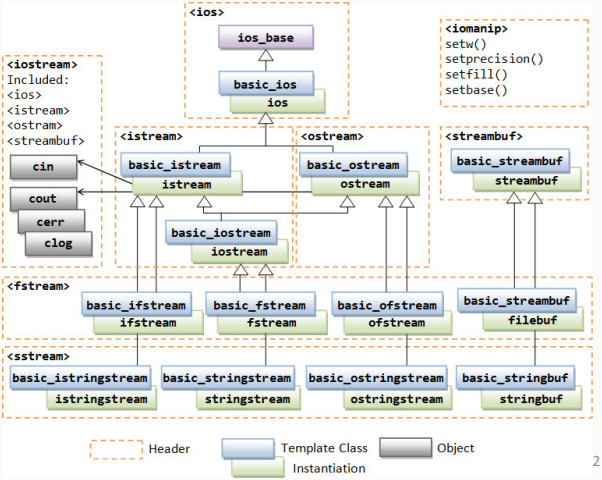


UNIVERSITÄT BONN

Beispiel Container und speziellere Container



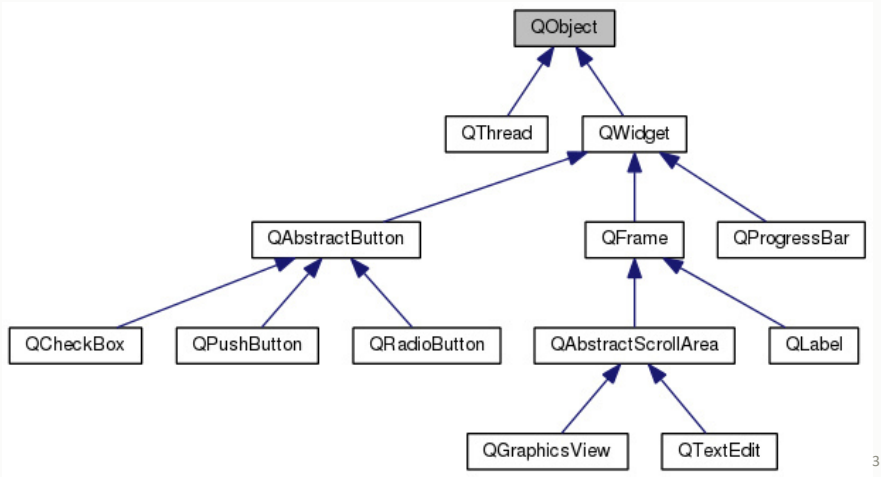
Beispiel C++-Typen für Input/Output



[2]

[2] Siehe https://www3.ntu.edu.sg/home/ehchua/programming/cpp/cp10_io.html

Beispiel
einige Typen des GUI-Frameworks QT



[3]

[3] Siehe https://wiki.qt.io/Qt_for_Beginners

Zusammenfassung

Typverwandschaft wird durch Subtypenbeziehungen ausgedrückt

Um Subtypenbeziehungen nachvollziehbar zu modellieren, verwenden wir das Liskovsche Substitutionsprinzip

Haben Sie Fragen?

Subtypenbeziehungen

Subtypenbeziehung durch Typerweiterung

Ziel

Um Subtypenbeziehungen zu erschaffen, gibt es viele Ansätze

Sie lernen zuerst die reine Typerweiterung kennen. Sie stellt den simpelsten Ansatz dar.

UNIVERSITÄT BONN

Typerweiterung

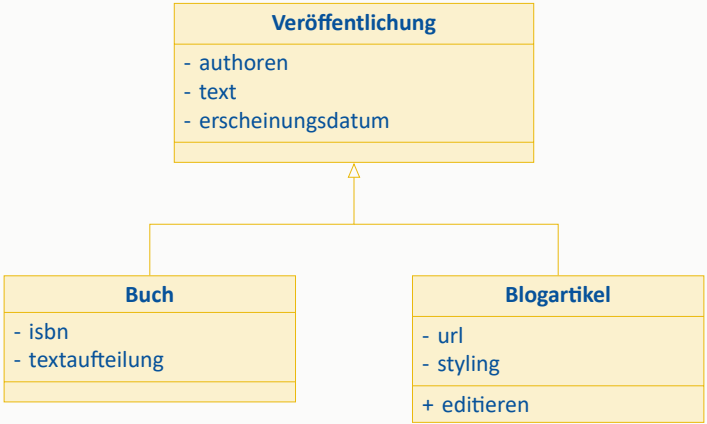
Die **Typerweiterung** stellt die einfachste Form der Subtypenbeziehung dar. Hierbei wird ein gegebener Typ T durch das **Hinzufügen von Attributen** und das **Erweitern der Interaktionsschnittestelle** zu einem Subtyp S erweitert.

Da S mindestens die Attribute von T enthält und mindestens die Interaktionsmöglichkeiten von T zur Verfügung stellt, können Objekte X vom Typ T immer durch Objekte von Typ S ersetzt werden, ohne die gewünschten Eigenschaften eines beliebigen Programms zu verändern. Nach dem **Liskovschen Substitutionsprinzip** darf S ein Subtyp von T sein.

UNIVERSITÄT BONN

Typerweiterung am Beispiel

Bei der Typerweiterung wird die Subtypenbeziehung angegeben und die zusätzlichen Attribute und Memberfunktionen beschrieben.



Ausblick

Subtypenbildung durch ausschließliche Typerweiterung ist untypisch. In Realweltanwendungen werden Subtypenbeziehung durch **Typerweiterung** und **Typkonkretisierung** gebildet. Später mehr.

Zusammenfassung

Bei der reinen Typerweiterung entstehen Subtypen durch das Hinzufügen von Attributen und die Erweiterung der Interaktionsschnittstelle

Dies ist die einfachste Möglichkeit um Subtypenbeziehungen zu erhalten

Haben Sie Fragen?

Erholsame Feiertage und einen guten Rutsch ins neue Jahr