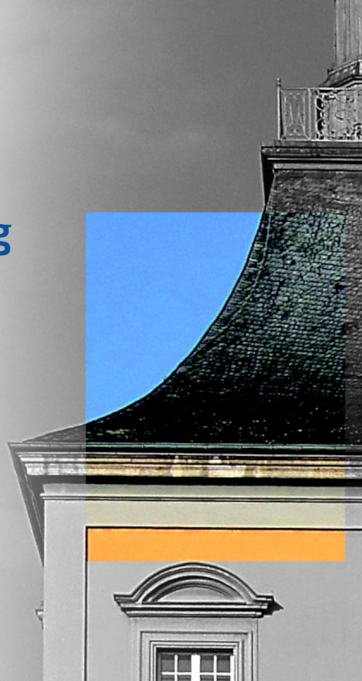
UNIVERSITÄT BONN

Algorithmen und Programmierung

Objektorientierte Programmierung

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



Gitversion: '6b5fb3b5caf2a83471839f83633f915d7052c1dd'

KURZE WIEDERHOLUNG

UNIVERSITÄT BONN

Wiederholung I

Faustregel: Man nennt eine Eigenschaft **statisch**, wenn die Ausprägung zur Compilezeit feststeht. Man nennt eine Eigenschaft **dynamisch**, wenn die Ausprägung zur Laufzeit feststeht.

Subtypbildung wird typischerweise durch **öffentliche Subklassenbildung** realisiert.

Bei der **reinen Typerweiterung** wird eine Oberklasse um Attribute und Memberfunktionen erweitert. Falls dabei Member erneut genannt werden, werden diese verdeckt. Bei der **reinen Typkonkretisierung** werden die virtuellen Memberfunktionen überschrieben. In Realweltbeispielen wird Typerweiterung und Typkonkretisierung verwendet.

Objekte mit virtuellen Memberfunktionen heißen **polymorph**.

Referenzen oder Zeiger auf polymorphe Objekte haben einen **statischen** und einen **dynamischen Typ**.

UNIVERSITÄT BONN

Wiederholung II

Wird ein Objekt via Referenz oder Zeiger aufgefasst oder übergeben, werden nicht-virtuellen Memberfunktionen **statisch gebunden**. Dazu wird der statische Typ des Objekts verwendet. Die virtuellen Memberfunktion werden **dynamisch gebunden**. Dazu wird der dynamische Typ des Objekts verwendet.

In **Java** und **Python** sind praktisch alle Objekte polymorph. In **C++** sind Objekte polymorph, wenn sie virtuelle Methoden erben oder deklarieren.

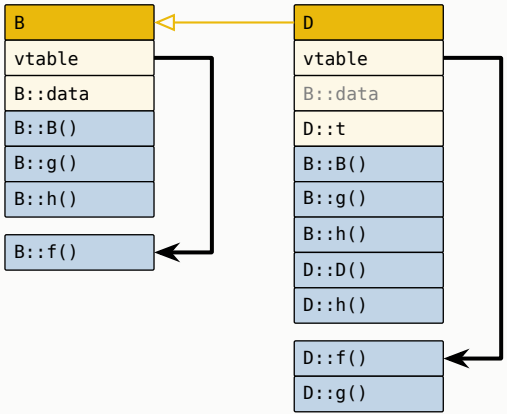
UNIVERSITÄT BONN

Wiederholung III

Virtuelle Memberfunktionen werden im Attributblock als `vtable` organisiert. Dies ist ein Tabelle von Funktionszeigern. Das Überschreiben ersetzen Einträge in der `vtable`.

```
class B {
public:
  virtual void f();
  void g();
  void h();
private:
  int data;
};

class D : public B {
public:
  void f();
  virtual void g();
  void h();
private:
  std::string t;
};
```



Subtypenbeziehungen

Typkonvertierung

Offene Frage

Wie wird der Typ eines Objekts konvertiert?

UNIVERSITÄT BONN

Typen konvertieren

Um den Wert einer Expression vom statischen Typ D zu einem Wert vom statischen Typ B zu konvertieren, verwendet man **Typkonvertierung**. Die **Typkonvertierung** kann **implizit** oder **explizit** sein und es ist möglich den Typ zur Compilezeit oder zur Laufzeit zu konvertieren.

Typkonvertierung wird fast immer verwendet, um den **gegebenen Typ** eines Werts oder eines Objekts in den **benötigten Typ** eines Befehls umzuwandeln.

UNIVERSITÄT BONN

Implizite, statische Typkonvertierung

Der implizite, statische Typkonvertierung sind wir bereits an mehreren Stellen begegnet, ohne sie hervorzuheben. Einfach gesagt wird sie immer dann verwendet, wenn ein Befehl ein Expression vom Typ B benötigt aber die Expression einen Typ D hat, welcher ein Subtyp von B ist.

```
int x = 5;
const int y = x; // int zu const int

float z = x;     // int zu float

D d;
B b = d;    // Unterklasse D zu Oberklasse B

B& bref = d // Unterklasse D zu Oberklasse B&
```

UNIVERSITÄT BONN

Explizite, statische Typkonvertierung

Um den statischen Typ einer Expression explizit zur **Compilezeit** zu ändern, verwendet wir die Operation **`static_cast`**`<NeuerTyp>(Expression)`. Dabei muss der **statische Typ der Expression** in den **neuen statische Typ** überführbar sein. Übliche Beispiele sind die Folgenden, wobei D $\rightarrow$ B:

```
D d;
B b = static_cast<B>(d); // Das wäre auch mit impliziter Typkonvertierung möglich

double x = 3.9;
int i = static_cast<int>(x); // 3.9 wird zu 3

B& bref = d; // Überschriebene Funktionen bleiben erhalten, evtl findet Verdeckung statt
D& dref = static_cast<D&>(bref) // Verdeckung wird wieder aufgehoben
```

Achtung: Statische Typkonvertierung findet zur Compilezeit statt. Im letzten Beispiel wird nicht geprüft, ob **`static_cast`**`<D&>(...)` zulässig ist. Das führt ggf. zu Laufzeitfehlern.

UNIVERSITÄT BONN

Explizite, dynamische Typkonvertierung

Um den **statischen Typ einer Expression** explizit zur **Laufzeit** zu ändern, verwendet wir die Operation **dynamic_cast**<NeuerTyp&>(Expression). Dabei muss der **dynamische Typ der Expression** in den **neuen dynamische Typ** überführbar sein. Übliche Beispiele sind die Folgenden, wobei D ⟶▷B:

```
D d;
B& bref = d;
D& dref = dynamic_cast<D&>(bref) // Es wird zur Laufzeit geprüft, ob die Konvertierung zulässig ist
```

Falls der dynamische Typ nicht zum neuen statischen Typ passt, wird die Ausnahme **std::bad_cast** geworfen (gleich mehr).

UNIVERSITÄT BONN

Beispiel

```
// Header
class GeschlKurve ...
class Polygon : public GeschlKurve ...
class Kreis   : public GeschlKurve ...

// Demo
int main() {
  GeschlKurve g;
  Polygon p;
  Kreis k;

  GeschlKurve& gref = p;
  static_cast<Kreis&>(gref);  // Produziert Laufzeitfehler oder unerwartetes Verhalten
  dynamic_cast<Kreis&>(gref); // Wirft in jedem Fall Ausnahme std::bad_cast.
}
```

UNIVERSITÄT BONN

Typkonvertierung von Pointern

Die statische und dynamische Typkonvertierung steht nicht nur Referenzen zur Verfügung, sondern kann auch bei Shared Pointer verwendet werden. Hier stehen uns die `static_pointer_cast` und `dynamic_pointer_cast` zur Verfügung. Falls die Zuweisung bei `dynamic_pointer_cast` nicht zulässig ist, wird nicht `std::bad_cast` geworfen, sondern ein Nullpointer zurückgegeben.

```
class GeschlKurve ...
class Polygon : public GeschlKurve ...
class Kreis   : public GeschlKurve ...

std::shared_ptr<GeschlKurve> k = std::make_shared<Polygon>();
std::cout << "Dynamischer Typ von K: " << typeid(*K).name() << std::endl;
std::dynamic_pointer_cast<Polygon>(K); // verschieden vom Nullpointer
std::dynamic_pointer_cast<Kreis>(K);   // ist ein Nullpointer
```

UNIVERSITÄT BONN

Beispiel

Wir wollen mit einer alternierende Folge von Polygonen und Kreisen arbeiten. Dazu nutzen wir ein Array, das Shared Pointer auf Objekte vom Typ `GeschlKurve` speichert.

```
typedef std::shared_ptr<GeschlKurve> GeschlKurvePtr; // und genauso PolygonPtr und KreisPtr;

int main() {
  std::vector<GeschlKurvePtr> v;

  // Erstelle alternierende Folge aus Userinput
  while (userinput_vorhanden) {
    v.push_back(std::make_shared<Polygon>(/* User Input */));
    v.push_back(std::make_shared<Kreis>(/* User Input */));
  }

  // Verarbeite alternierende Folge
  for (size_t i = 0; i < v.size()/2; ++i) {
    PolygonPtr p_aktuell = std::dynamic_pointer_cast<Polygon>(v[2*i]);
    KreisPtr   k_aktuell = std::dynamic_pointer_cast<Kreis>(v[2*i + 1]);
    /* Tue Dinge mit den spezielleren, geschlossenen Kurven */
  }
}
```

UNIVERSITÄT BONN

Reinterpret Cast

Um eine Bytefolge im Speicher als Wert eines Typs zu interpretieren, verwendet man **`reinterpret_cast`**<NeuerTyp&>. Diese Operation veranlasst den Compiler diese Bytefolge sofort und ohne weitere Anpassungen als den Wert des neuen Typs aufzufassen.

Damit verhält sich **`reinterpret_cast`**<neuerTyp&> in manchen Fällen anders als **`static_cast`**<neuerTyp&>. Das ist zum Beispiel dann der Fall wenn wir ein Objekt verwenden, welches durch Mehrfachvererbung aus mehreren Attributblöcken besteht. Wenn `neuerTyp` eine der Oberklassen ist, ändert **`static_cast`**<neuerTyp&> die Referenz auf den zugehörigen Attributblock, aber **`reinterpret_cast`**<neuerTyp&> ändert die Referenz nicht.

Die Unsachgemäße Verwendung von **`reinterpret_cast`** führt zu unerwartetem Laufzeitverhalten.

Zusammenfassung

Wir haben gelernt, wie wir den statischen Typ eines Objekts konvertieren

Die Konvertierung kann statisch oder dynamisch geschehen

Haben Sie Fragen?

Subtypenbeziehungen

Ersetzbarkeit von Parametern und Rückgaben

Ziel

Welche Auswirkungen hat die Ersetzbarkeit auf Funktionsparameter und Funktionsrückgaben?

UNIVERSITÄT BONN

Ersetzbarkeit in Parametern
Call by Value

```
class B {
public:
  void f()         {std::cout << "B::f" << std::endl;}
  void virtual g() {std::cout << "B::g" << std::endl;}
};

class D: public B {
public:
  void g() override {std::cout << "D::g" << std::endl;}
};

void call_test(B b) { // Call by Value
  b.f(); // Verwendet B::f(), denn b ist vom Typ B
  b.g(); // Verwendet B::g(), denn b ist vom Typ B
}

int main() {
  D d;
  call_test(d);
}
```

Bei der Parameterübergabe *Call by Value* wird ein neues B-Objekt angelegt, welches den B-Anteil von d als Kopie erhält (exklusive der `vtable`).

Also dürfen speziellere Parameter als Kopie übergeben werden.

UNIVERSITÄT BONN

Ersetzbarkeit in Parametern
Call by Reference

```
class B {
public:
  void f()         {std::cout << "B::f" << std::endl;}
  void virtual g() {std::cout << "B::g" << std::endl;}
};

class D: public B {
public:
  void g() override {std::cout << "D::g" << std::endl;}
};

void call_test(B& b) { // Call by Reference
  b.f(); // Verwendet B::f(), denn b ist vom Typ B-Ref
  b.g(); // Verwendet D::g(), denn b ist vom Typ B-Ref
}

int main() {
  D d;
  call_test(d); // d wird als B-Referenz übergeben
}
```

Bei der Parameterübergabe *Call by Reference* wird `d` als B-Referenz übergeben. Deshalb werden die, in `B` als nicht-virtuell deklarierten Funktionen, als `B::` aufgerufen und die, in `B` als virtuell deklarierten Funktionen, als `D::` aufgerufen.

Also dürfen speziellere Parameter als Referenz übergeben werden.

UNIVERSITÄT BONN

Ersetzbarkeit bei Rückgaben

```
class B {
public:
  void f()         {std::cout << "B::f" << std::endl;}
  void virtual g() {std::cout << "B::g" << std::endl;}
};

class D: public B {
public:
  void g() override {std::cout << "D::g" << std::endl;}
};

D return_test() {
  return D();
}

int main() {
  B b;
  b = return_test();
}
```

Bei der Rückgabe wird eine Expression vom Typ D erzeugt. Dieser Wert wird einem Objekt vom Typ B zugewiesen (exklusive der `vtable`).

Also dürfen speziellere Typen zurückgegeben werden.

UNIVERSITÄT BONN

Überladene Funktionen

Auch überladenen Funktionen dürfen speziellere Funktionsparameter erhalten. Einfach gesagt, sucht der Compiler hierbei die am besten passenste Funktion aus. Simples Beispiel mit B ◁—D ◁—E:

```
void f(const B&) { std::cout << "f(const B&)" << std::endl; }
void f(const D&) { std::cout << "f(const D&)" << std::endl; }
...
f(E(0.9)); // druckt "func(const D&)".
```

Falls keine beste Funktion gewählt werden kann, kann das Programm nicht kompiliert werden.

```
void f(const B&, const D&) { std::cout << "f(const B&, const D&)" << std::endl; }
void f(const D&, const B&) { std::cout << "f(const D&, const B&)" << std::endl; }
...
f(E(0.9), B(1.0)); // druckt "f(const D&, const B&)"
f(E(0.9), E(1.0)); // error: call of overloaded 'f(E&, E&)' is ambiguous
```

Zusammenfassung

Als Konsequenz der Ersetzbarkeit dürfen Funktionen speziellere Funktionsparameter erhalten und Funktionsrückgaben als allgemeinerer Typ aufgefasst werden

Bei überladenen Funktionen können speziellere Funktionsparameter ggf. zu mehreren überladenen Funktionen passen. Hier ist Vorsicht geboten.

Haben Sie Fragen?

Subtypenbeziehungen

Ersetzbarkeit bei überschriebenen Funktionen

Ziel

Welche Freiheiten bietet uns die Ersetzbarkeit von Parametertypen und Rückgabentypen, beim Überschreiben von Memberfunktionen?

Einleitung

Beim Überschreiben von Memberfunktionen fragen wir uns, welche Freiheiten uns bei der Wahl der Parametertypen und Rückgabetypen geboten sind. Dazu formulieren wir eine weitere Variante des Liskovsche Substitutionsprinzips.

(Liskovsches Substitutionsprinzip (Umformulierung))

Falls der Typ S ein Subtyp vom Typ T ist, dann fordert[1] S höchstens soviel wie T und bietet[2] mindestens soviel an wie T.

Diese Sichtweise erlaubt es uns, beim Überschreiben von virtuellen Funktionen die Parametertypen und Rückgabetypen zu verändern.

[1] Im Sinne der Memberfunktionen und deren Parametertypen

[2] Im Sinne der Memberfunktionen und deren Rückgabetypen

UNIVERSITÄT BONN

Anwendung

Wir betrachten Subtypen X ◁—Y ◁—Z und B ◁—D. Gegeben ein D-Objekt `obj` mit überschriebener Memberfunktion `B::f`, dass als D-Objekt aufgefasst wird.

```
class B {
public:
  virtual Y f(Y);
};

class D : public B {
  // Welche Typen
  // sind zulässig?
  ... f ( ... ) ;
};

int main() {
  D obj;
  B& bref = obj;
  // Ruft D::f auf
  Y y = bref.f(...);
}
```

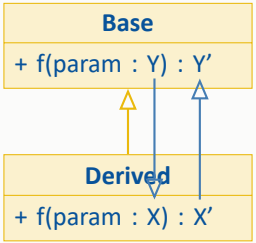
Damit `obj` als allgemeines B-Objekt verwendet werden kann, muss `obj` alle Parameter verarbeiten können, die ein B-Objekt verarbeiten kann. Also darf `D::f` nicht Z als Parametertyp fordern. Aber man darf X als Parametertyp fordern, denn wenn man einem B-Objekt ein Y-Parameter übergibt, kann `D::f` diesen als X-Parameter verwenden.

Damit `obj` als allgemeines B-Objekt verwendet werden kann, müssen alle Rückgabetypen, die ein B-Objekt produziert, auch von `obj` produziert werden. Also darf `D::f` nicht X zurückgeben. Aber `D::f` darf Z als Rückgabetyp anbieten, denn ein Z-Typ kann als Y-Typ verwendet werden.

UNIVERSITÄT BONN

Ko- und Kontrvariantes Verhalten

Beim Überschreiben von Memberfunktionen dürfen die Parametertypen allgemeiner und die Rückgabetypen spezieller werden.



In Bezug auf die Vererbungshierarchie dürfen Rückgabetyp in derselben Richtung spezieller werden. Parametertypen dürfen aber nur in der umgekehrten Richtung spezieller werden. Man spricht bei Rückgabetypen von **kovariantem Verhalten** und bei Parametertypen von **kontravariantem Verhalten**.

UNIVERSITÄT BONN

Substitutionsprinzip und allgemeiner Sprachgebrauch

Das Liskovsche Substitutionsprinzip fällt nicht immer mit unserem alltäglichen Sprachgebrauch zusammen. Beispiel:

Offenbar benötigen alle `Tiere` auch `Nahrung`. Wenn eine `Katze` ein spezielles `Tier` ist und `Katzenfutter` eine spezielle `Nahrung` ist, ist eine `Katze` die `Katzenfutter` verlangt wirklich ein `Tier`?

Offenbar kann man einer `Bildermenge` neue `Bilder` hinzufügen. Wenn `Memes` spezielle `Bilder` sind und eine `Mememenge` eine spezielle `Bildermenge` ist, ist eine `Mememenge` der man nur `Memes` hinzufügen kann wirklich eine `Bildermenge`?

Zusammenfassung

Beim Überschreiben von Memberfunktionen, dürfen die Parametertuypen allgemeiner werden (kontravariant) und die Rückgabetypen spezieller werden (kovariant).

Haben Sie Fragen?

Vererbung - Zusammenfassung

Ziel

Wir fassen die wichtigsten Aspekte der Vererbung zusammen

Die zusammengefassten Inhalte sind grundlegend

Die richtige Verwendung der zusammenfassten Inhalte müssen selbstständig eingeübt werden

Ergänzen Sie die hier zusammengefassten Inhalte selbstständig um die ausgelassenen Details

UNIVERSITÄT BONN

Motivation

Vererbung wird verwendet, um wiederkehrende Codeabschnitte von „verwandten“ Typen inhaltlich sinnvoll zu Bündeln. Durch die Ersetzbarkeit von „verwandten“ Typen können Subtypen einfach ergänzt werden und an Stelle der Obertypen übergeben werden, ohne dass das bereits vorhandene Projekt modifiziert werden muss.

Das führt zu wartbarerem, flexiblerem und nachvollziehbarerem Programmcode und macht den Aufbau und die Funktionsweise des gesamten Projekts nachvollziehbarer.

UNIVERSITÄT BONN

Identifier und implizite Instanzreferenz

Alles was in **C++**, **Java** oder **Python** einen Namen hat, kann mithilfe von relativen oder absoluten einschränkenden Identifiern relativ oder absolut eindeutig benannt werden. Das ist z.B. bei der Benennung von Typen in Namespaces, Membern in Oberklassen oder bei der Funktionsbestimmung von überladenen Funktionen nötig.

Bei der Funktionsauswahl in **C++** wird zuerst der Name aufgelöst und anschließend bestimmt, welche der Überladungskandidaten verwendet werden kann. Das kann zu unerwarteten Nebeneffekten führen.

Innerhalb einer Memberfunktion wird das aufrufende Objekt durch die implizite Instanzreferenz benannt. In **C++** ist die implizite Instanzreferenz **`this`** ein Zeiger.

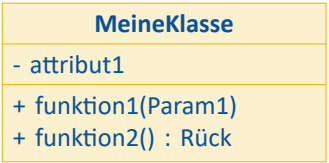
UNIVERSITÄT BONN

Klassen und Objekte

Objekte sind Instanzen von zugehörigen Klassen und bestehen aus Attributen sowie einer Interaktionsschnittstelle, die jeweils mit Zugriffsspezifizierungen versehen sind.

Dass Attribute nur durch Objekte desselben Typs direkt gelesen oder geändert werden können, bezeichnet man als Kapselung. Kapselung wird verwendet, damit jedes Objekt einen zulässigen Zustand hat. Indirekter Attributzugriff kann durch Setter und Getter realisiert werden.

Klassen werden in UML und objektorientierten Sprachen beschrieben. Wir drücken Modellierung üblicherweise im Sketchingdialekt aus.




```
class MeineKlasse {
public:
  void funktion1(Parametertyp parametername);
  Rueckgabetyp funktion2();
private:
  Attributtyp1 attribut1;
};
```

UNIVERSITÄT BONN

Die Ist-Ein-Faustregel und das Liskovsche Substitutionsprinzip

(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

(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.

(Liskovsches Substitutionsprinzip (Umformulierung))

Falls der Typ S ein Subtyp vom Typ T ist, dann fordert S höchstens soviel wie T und bietet mindestens soviel an wie T.

UNIVERSITÄT BONN

Subtypenbildung

Üblicherweise werden Subtypen durch Typerweiterung sowie Typkonkretisierung gebildet.

Durch das Deklarieren von Membern im Subtyp werden neue Member hinzugefügt und identisch benannte Member des Obertyps verdeckt bzw. virtuelle Memberfunktionen überschrieben.

Subtypen die mindestens eine virtuelle Funktion erben oder deklarieren heißen polymorph. Subtypen die über keine virtuelle Funktion verfügen sind nicht polymorph.

In UML wird die Subtypenbeziehung durch `Base` ◁—`Derived` angegeben und Memberfunktionen sind üblicherweise virtuell. In `C++` wird die Subtypenbeziehung durch öffentliche Subklassenbildung angegeben und virtuelle Memberfunktion müssen mit **`virtual`** gekennzeichnet werden. Damit der Compiler Programmierfehler findet, soll das Übschreiben mit `override` gekennzeichnet werden.

UNIVERSITÄT BONN

Ersetzbarkeit
Statischer und dynamischer Typ

Wir betrachten polymorphe Typen `B` ◁—`D` und ein `D`-Objekt `obj`.

Wenn nun `obj` vermöge eine Referenz `ref` als `B`-Objekt behandelt wird, dann ist der statische Typ `B` und der dynamische Typ `D`.

Wird nun die Funktion `ref.f()` verwendet, wird zur Compilezeit entschieden, welche Funktion verwendet wird, falls `B::f` nicht virtuell ist. Anderenfalls wird zu Laufzeit entschieden welche Funktion verwendet wird.

Wird nun die Funktion `ref.f()` verwendet, wird `D::f` aufgerufen falls `B::f` virtuell ist und in `D` überschrieben wird. In allen anderne Fällen wird `B::f` aufgerufen.

UNIVERSITÄT BONN

Ersetzbarkeit
Parameter- und Rückgabetypen

Wir betrachten eine (Member)funktion `funk`, die Parametertypen `T1, ..., Tn` verlangt und einen Rückgabetyp `Y` anbietet.

```
Y funk(T1, ..., Tn)
```

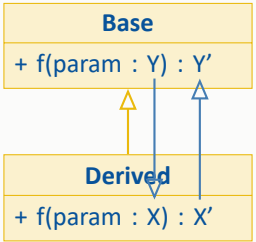
Nun können auch speziellere Parametertypen `Si` $\rightarrowtriangle$`Ti` an `funk` übergeben werden oder die Rückgabe von einem Objekt vom Typ `X` $\trianglelefteq$`Y` entgegen genommen werden.

Bei Überladenen Funktionen ist das nur dann möglich, wenn der Compiler einen eindeutig besten Funktionskandidaten bestimmen kann.

UNIVERSITÄT BONN

Ersetzbarkeit
Ko- und kontravarianz

Passend zu der Ersetzbarkeit von Parameter- und Rückgabetypen und dem umformulierten Liskovschen Substitutionsprinzip dürfen die Parametertypen von überschriebenen Funktionen kontravariant spezieller werden sowie die Rückgabetypen von überschrieben Funktionen kovariant spezieller werden.

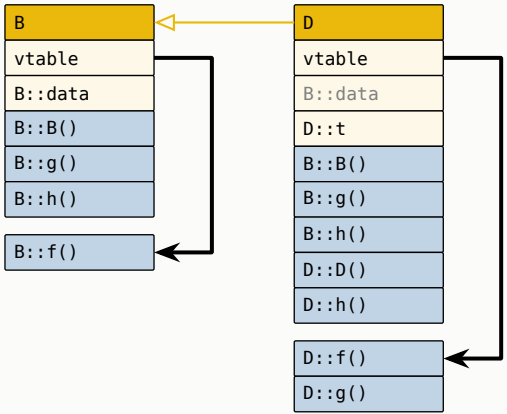


Umsetzung im Speicher

Virtuelle Memberfunktionen werden im Attributblock als `vtable` organisiert. Dies ist ein Tabelle von Funktionszeigern. Das Überschreiben ersetzen Einträge in der `vtable`.

```
class B {
public:
  virtual void f();
  void g();
  void h();
private:
  int data;
};

class D : public B {
public:
  void f();
  virtual void g();
  void h();
private:
  std::string t;
};
```



Die zusammengefassten Inhalte sind grundlegend

Die richtige Verwendung der zusammenfassten Inhalte müssen selbstständig eingeübt werden

Ergänzen Sie die hier zusammengefassten Inhalte selbstständig um die ausgelassenen Details

Ausnahmebehandlung

Offene Frage

Wie schreibt man gut nachvollziehbaren Code der viele Voraussetzungsprüfungen durchführen muss?

UNIVERSITÄT BONN

Motivation I

Es gibt immer wieder Codeabschnitte, die Datenströme oder Nutzereingaben verarbeiten. Die fehlerfreie Verarbeitung kann nur dann gelingen, wenn stets alle Voraussetzungen erfüllt sind. Typische Voraussetzungen sind:

- Das Öffnen, Lesen oder Schreiben einer Datei ist zulässig und geschieht fehlerfrei.
- Das Lesen eines Netzwerkdatenstroms gelingt ohne Verbindungsabbruch.

UNIVERSITÄT BONN

Motivation II

Codeabschnitt die stets keine / alle Voraussetzungen prüfen, sehen wie folgt aus.

```
std::vector<MeineKlasse> v;
lies_aus_netz(v);
int i;
std::cin >> i;
std::cout << v[i] << std::endl;
```

Codeabschnitte die keine Voraussetzungen prüfen sind zwar einfacher zu verstehen führen aber in Ausnahmefällen zu unerwartetem Verhalten.

```
std::vector<MeineKlasse> v;
if (!lies_aus_netz(v)) { /* Ausnahme */ }
int i;
std::cin >> i;
if (cin.fail()) { /* Ausnahme */ }
if (i < 0 or i >= v.size()) { /* Ausnahme */ }
std::cout << v[i] << std::endl;
```

Codeabschnitte die alle Voraussetzungen prüfen sind zwar sehr unübersichtlich führen aber nie zu unerwartetem Verhalten.

Mit **Ausnahmebehandlungen** werden beide Vorteile vereint.

UNIVERSITÄT BONN

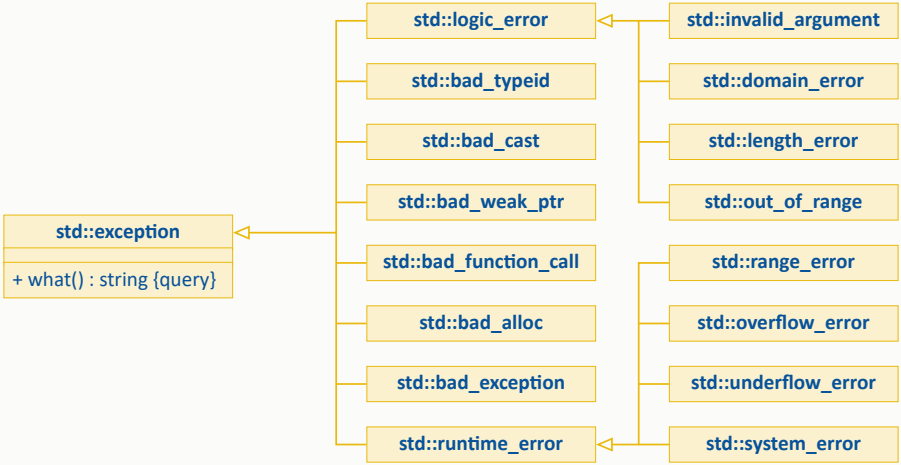
Ausnahmebehandlung

Bei der **Ausnahmebehandlung** trennt man inhaltlich zusammenhängende Codeblöcke und die Behandlung von dort auftretenden Ausnahmen voneinander. Beim Auftreten einer Ausnahme, wird der Codeblock verlassen und die Ausnahme behandelt. Anschließend wird der Codeblock nicht weiter ausgeführt, sondern der nächste Befehl nach dem Codeblock wird verarbeitet.

```
try {
  // Inhaltlich zusammenhängender Codeblock
  // der Ausnahmefälle enthalten kann
} catch (const AUSNAHMETYP1& e) {
  // Behandlung der ersten, auftretenden Ausnahme, falls die Ausnahme den Typ AUSNAHMETYP1 hat
  // Textdarstellung der Ausnahme: e.what()
} catch (const AUSNAHMETYP2& e) {
  // Behandlung der ersten, auftretenden Ausnahme, falls die Ausnahme den Typ AUSNAHMETYP2 hat
  // Textdarstellung der Ausnahme: e.what()
} catch (...) {
  // Behandlung der ersten, auftretenden Ausnahme, falls die Ausnahme einen anderen Typ hat
  // Textdarstellung der Ausnahme: e.what()
}
```

UNIVERSITÄT BONN

Auswahl von Ausnahmetypen des C++-Standards



UNIVERSITÄT BONN

Ausnahmen werfen

Um Ausnahmen zu erzeugen, die dann behandelt werden müssen, verwendet man die folgende Konstruktion.

```
throw AUSNAHMETYP(/* Beschreibender Text */);
```

Die aktuelle Funktion bricht hier ab und reicht die Ausnahme an die **aufrufende** Funktion zur Behandlung gegeben. Wird die Ausnahme dort nicht behandelt, bricht auch diese Funktion ab und gibt die Ausnahme zur nächsten, aufrufenden Funktion weiter.

Dies geschieht sukzessive bis die Ausnahme behandelt wurde. Zum Beispiel indem der Prozess den folgenden Text druckt und anschließend beendet wird.

```
terminate called after throwing an instance of 'std::...'
  what():  .......
```

UNIVERSITÄT BONN

Beispiel

Hier ein sehr simples Beispiel.

```
void coole_frage() {
  std::cout << "Ist die AlPro gut? (Ja/Nein)" << std::endl;
  std::string antwort;
  std::cin >> antwort;
  if (antwort != "Ja") {
    throw std::logic_error("Unlogische Antwort erhalten");
  }
}

int main() {
try {
    coole_frage();
    std::cout << "Die Frage wurde gut beantwortet." << std::endl;
  } catch (const std::logic_error& e) {
    std::cout << "Ausnahme '" << e.what() << "' behandelt." << std::endl;
  }
  std::cout << "Bye Bye" << std::endl;
}
```

UNIVERSITÄT BONN

Weiter Anwendung von Ausnahmen

Eine weitere Anwendungskontext von Ausnahmen ist der Folgende. Es kommt manchmal vor, dass wir eine Funktion schreiben, die einen Typ zurückgibt, dessen Werte alle zulässig sind. Allerdings kann es bei der Ausführung der Funktion sehr selten zu einer Ausnahmesituation kommen, sodass keine sinnvolle Rückgabe produziert werden kann. In diesem Fall wirft man eine Ausnahme um die Funktionausführung führzeitig zu beenden und der **aufrufenden** Funktion die Ausnahmesituation zu kommunizieren.

```
// Gibt true zurück falls das Serverpasswort korrekt ist und
// false falls das Serverpasswort inkorrekt ist.
bool serverpassword_korrekt(const std::string& pw) {
  if (server.verbindungsaufbau() == false) {
    throw Verbindungsfehler("Verbindung zum Server kann nicht aufgebaut werden");
  }
  // ...
}
```

UNIVERSITÄT BONN

Eigene Ausnahme definieren

Da Ausnahmen eigene Klassen sind, können wir von diesen wie üblich erben. So können wir eigene Ausnahmen definieren, die wir anschließend werfen und behandeln können.

Beispiel

```
class unlogische_antwort : public std::logic_error {
public:
  unlogische_antwort(std::string s) : std::logic_error(s) {}
};

void coole_frage() {
  std::cout << "Ist die AlPro gut? (Ja/Nein)" << std::endl;
  std::string antwort;
  std::cin >> antwort;
  if (antwort != "Ja") {
    throw unlogische_antwort(antwort);
  }
}

int main() {
  try {
    coole_frage();
    std::cout << "Die Frage wurde gut beantwortet." << std::endl;
  } catch (const unlogische_antwort& e) {
    std::cout << "Unlogische Antwort '" << e.what() << "' behandelt." << std::endl;
  }
```

Zusammenfassung

Ausnahmbehandlungen erlauben es uns Code zu schreiben, der viele Voraussetzungsprüfungen benötigt und nachvollziehbar gestaltet ist

Funktionen, die ausnahmsweise keinen sinnvollen Rückgabewert produzieren können, sollen in diesem Fall eine Ausnahme werfen

Haben Sie Fragen?