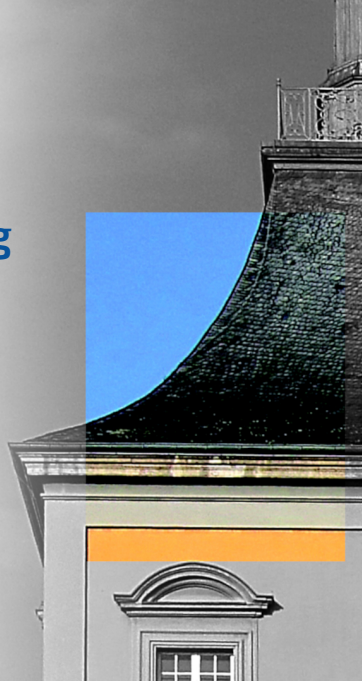
UNIVERSITÄT BONN

Algorithmen und Programmierung

Objektorientierte Programmierung

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



KURZE WIEDERHOLUNG

Einschub: Qualified Identifier in C++

UNIVERSITÄT BONN

Typen und Objekte eindeutiger benennen

Um einen Typ oder ein Objekt eindeutiger zu benennen, verwenden wir **Qualified Identifier**.

Absolute Identifier haben die Form `::NAME::NAME::...` und geben den vollen Namenspfad, beginnend bei globalen Namespace, an.

Relative Identifier haben die Form `NAME::NAME::...` und geben einen Namen an, der anschließend aufgelöst wird. Dabei wird der linke Name lokal, in Richrung des globalen Namespace gesucht und von dort auf weiter aufgelöst.

Subtypenbeziehungen

Subtypenbeziehung durch Typerweiterung

UNIVERSITÄT BONN

Öffentliche Subklassenbildung und Typerweiterung

Wir konzentrieren uns zunächst auf **Subtypbildung durch Typerweiterung**.

- Hierbei werden alle **hinzugefügten** Membervariablen und Memberfunktionen genannt.
- Die öffentlichen Membervariablen und Memberfunktionen der Oberklasse stehen automatisch zur Verfügung. Man sagt, dass die öffentlichen Membervariablen und Memberfunktionen **vererbt** wurden.
- Die privaten Membervariablen und Memberfunktionen der Oberklasse sind Teil der Unterklasse, aber Sie dürfen in den hinzugefügten Memberfunktionen der Unterklasse nicht direkt verwendet werden.

UNIVERSITÄT BONN

Öffentliche Subklassenbildung in C++

Um in **C++** auszudrücken, dass die Klasse `DerivedClass` eine öffentliche Subklasse von `BaseClass` ist, verwendet man die folgende Konstruktion.

```
class BaseClass {
  ...
};

class DerivedClass : public BaseClass {
  ...
};
```

UNIVERSITÄT BONN

Typerweiterung im Speicher

Erinnerung: Jedes Objekt verfügt über (einmal im Programmspeicher abgelegte) Memberfunktionen und besteht aus (pro Objekt existierenden, auf dem Stack oder Heap abgelegten) Attributen.

Bei der **reinen Typerweiterung** durch öffentliche Subklassenbildung, stehen der Subklasse alle öffentlichen Memberfunktionen der Oberklasse zur Verfügung und werden durch neue Memberfunktionen ergänzt. Im (Stack- oder Heap-)Speicher wird der Attributblock der Oberklasse durch die Attribute der Subklasse ergänzt. Allerdings verbietet der Compiler den neuen Memberfunktion den Zugriff auf die privaten Attribute der Oberklasse.

Beim Instanziieren von Objekten wird zuerst der Konstruktor der Oberklasse aufgerufen und anschließend der Konstruktor der Subklasse. Dadurch ist sichergestellt, dass die Oberklasse einen zulässigen Zustand enthält, bevor die Subklasse mit der Oberklasse interagieren kann.

Subtypenbeziehungen

Subtypenbeziehung durch Typkonkretisierung

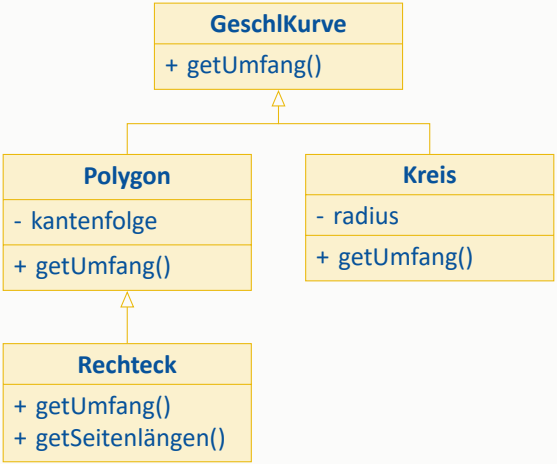
Offene Frage

Warum reicht die reine Typerweiterung nicht aus um Objekte zu modellieren?

Wie löst die Typkonkretisierung die auftretenden Probleme?

UNIVERSITÄT BONN

Motivierendes Beispiel



Die Umfangsberechnung von Polygonen und Kreisen geschieht unterschiedlich. Hier kann keine allgemeine, direkte Umfangsberechnungsfunktion angegeben werden ohne die Modellierung von Polygonen oder Kreisen zu ändern.

Dennoch ist es offensichtlich, dass wir einen möglichst konkreten Obertyp haben möchten. Dieser soll eine Memberfunktionen `getUmfang` besitzen, die dann von den Subtypen **konkretisiert** wird.

UNIVERSITÄT BONN

Wir wünschen uns überschreibbare Funktionen

Wir wünschen uns, dass **ausgewählte Memberfunktionen überschrieben** werden können, um konkrete Implementierungen bereitzustellen. Die konkreteren Memberfunktionenen sollen auch dann verwendet werden, wenn das Objekt (via Referenz oder Pointer) als allgemeinerer Typ aufgefasst wird.

UNIVERSITÄT BONN

Beispiel mit überschriebenen Funktionen

Angenommen die oben genannte Funktion `GeschlKurve::getUmfang` wird von Polygon und Kreis überschrieben. Dann sollte beim Einsetzen eines Polygons in die folgende Funktion die Memberfunktion `Polygon::getUmfang` ausgewählt werden (statt `GeschlKurve::getUmfang`).

```
void drucke_umfang(const GeschlKurve& K) {
  std::cout << "Der Umfang ist" << K.getUmfang() << "." << std::endl;
}

int main() {
  Polygon p = ...;
  drucke_umfang(p);
  // p wird als GeschlKurve-Ref übergeben.
  // Falls getUmfang   _verdeckt_    ist, wird die Funktion GeschlKurve::getUmfang verwendet.
  // Falls getUmfang _überschrieben_ ist, wird die Funktion Polygon::getUmfang verwendet.
}
```

UNIVERSITÄT BONN

Typerweiterung vs. Typkonkretisierung
Verdecken vs. Überschreiben

Angenommen `D` **verdeckt** die Funktion `B::f` und wir haben ein `D`-Objekte `obj`.

- Falls `obj` als `B`-Referenz oder `B`-Zeiger übergeben wird, dann entspricht `obj.f` der Memberfunktion `B::f`.
- Falls `obj` als `B`-Kopie übergeben wird, dann entspricht `obj.f` der Memberfunktion `B::f`.

Angenommen `D` **überschreibt** die Funktion `B::f` und wir haben ein `D`-Objekte `obj`.

- Falls `obj` als `B`-Referenz oder `B`-Zeiger übergeben wird, dann entspricht `obj.f` der Memberfunktion `D::f`.
- Falls `obj` als `B`-Kopie übergeben wird, dann entspricht `obj.f` der Memberfunktion `B::f`.

UNIVERSITÄT BONN

Überschreiben und Verdecken in objektorientierten Sprachen

Objektorientierte Programmiersprachen verhalten sich unterschiedlich bei der Frage welche Memberfunktionen überschrieben oder verdeckt werden können.

- In **C++** können Memberfunktionen überschrieben oder verdeckt werden. Außerdem muss angegeben werden, ob Memberfunktionenen (nicht) überschreibbar sind.
- In **Java** können Memberfunktion praktisch nur überschrieben werden. Praktisch alle Memberfunktionen sind überschreibbar, aber es kann angegeben werden, dass Memberfunktionenen nicht überschreibbar sind.
- In **Python** können Memberfunktion praktisch nur überschrieben werden. Praktisch alle Memberfunktionen sind überschreibbar. Es kann nicht angegeben werden, dass Memberfunktionenen nicht überschrieben werden dürfen.

Zusammenfassung

Die Typkonkretisierung erlaubt es uns Memberfunktionen zu überschrieben

Überschriebene Memberfunktionen werden auch verwendet, wenn ein Objekt als allgemeinere Referenz behandelt wird

Haben Sie Fragen?

Subtypenbeziehungen

Subtypenbeziehung durch Typkonkretisierung modellieren

Offene Frage

Wie drückt man Typkonkretisierung beim Modellieren aus?

UNIVERSITÄT BONN

Virtuelle Memberfunktionen und polymorphe Typen

Überschreibbare Memberfunktionen nennen wir auch **virtuell**.

Ein Typ heißt **polymorph**, falls er mindestens eine virtuelle Memberfunktion enthält.

In **C++** sind alle Typen **polymorph**, die eine virtuelle Memberfunktion erben oder eine hinzufügte Memberfunktion als virtuell deklarieren.

In **C++** sind alle Typen **nicht polymorph**, die durch reine Typerweiterung entstanden sind und keine Memberfunktion als virtuell deklarieren.

In **Java** und **Python** enthält praktisch jeder Typ mindestens eine virtuelle Funktion. Deshalb sind in **Java** und **Python praktisch alle** Typen **polymorph**.

UNIVERSITÄT BONN

Memberfunktionen überschreiben in UML

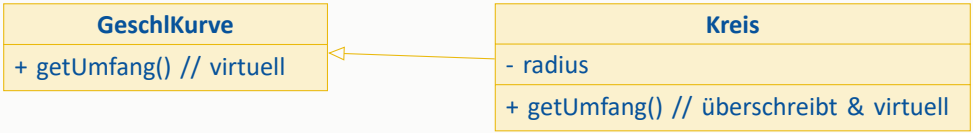
In UML ist es besonders simpel auszudrücken, dass eine Memberfunktion **virtuell** ist oder dass eine Memberfunktion eine virtuelle Funktion **überschreibt**.

Im Normalfall sind alle Memberfunktionen virtuell. Durch das erneute Nennen der Memberfunktion in der Subklasse wird erklärt, dass eine Überschreibung stattfinden soll.

UNIVERSITÄT BONN

Beispiel

Wir betrachten unser Beispiel erneut.



Die Memberfunktion `GeschlKurve::getUmfang` ist virtuell. Jeder Typ der von dem Typ `GeschlKurve` erbt, erhält die Memberfunktion `GeschlKurve::getUmfang`, es sei denn, er überschreibt `GeschlKurve::getUmfang`.

Hier überschreibt `Kreis` die Memberfunktion `GeschlKurve::getUmfang` mit der Memberfunktion `Kreis::getUmfang`. Diese ist selbst wieder virtuell.

Wird ein Objekt vom Typ `Kreis` als `GeschlKurve`-Referenz übergeben, wird die Funktion `Kreis::getUmfang` verwendet.

Zusammenfassung

Überschreibbare Memberfunktionen heißen virtuell

Typen sind polymorph wenn sie über mindestens eine virtuelle Memberfunktion verfügen

Haben Sie Fragen?

Subtypenbeziehungen

Subtypenbeziehung durch Typkonkretisierung in C++

Offene Frage

Wie werden polymorphe Typen in C++ realisiert?

UNIVERSITÄT BONN

Überschreibbare Memberfunktionen

In C++ wird durch die Angabe des Keywords **`virtual`** spezifiziert, dass eine Memberfunktion virtuell (also überschreibbar) ist.

```
class B {
public:
  virtual void f(); // B::f ist virtuell
  void g();         // B::g ist nicht virtuell
};
```

Subklassen von `B` können die Memberfunktion `B::f` übeschreiben. Die Memberfunktion `B::g` ist nicht virtuell (also nicht überschreibbar).

UNIVERSITÄT BONN

Memberfunktionen überschreiben I

Wird in einer Subklasse `D` ein Memberfunktion `funk` deklariert, die in der Oberklasse `B` deklariert ist, so wird diese überschrieben falls `B::funk` virtuell ist. Andernfalls wird sie verdeckt.

Um sich klarer auszudrücken, darf das Keyword `override` bei überschreibenden Memberfunktionen angegeben werden. Der Compiler prüft dann, ob das Überschreiben zulässig ist.

```
class B {
public:
  virtual void f();  // B::f ist virtuell
};

class D : public B { // D erbt von B
public:
  void f() override; // D::f überschreibt B::f
};
```

UNIVERSITÄT BONN

Memberfunktionen überschreiben II

Wird eine Klasse `B` ein Memberfunktion `funk` deklariert, kann diese in der Subklasse `B` nur dann überschrieben werden, falls die folgenden Bedingungen erfüllt sind.

- Die Memberfunktion `B::funk` muss als **`virtual`** deklariert sein.
- Die Memberfunktion in `D` muss den Namen `funk` tragen.
- Alle Parameter müssen identisch sein (inklusive Reihenfolge, `const`-Quailfiern und `Reference`-Qualifiern).
- Der Rückgabetyp muss identisch.

Wir lernen später, dass die Parametertypen auch allgemeiner und die Rückgabetypen auch spezieller sein dürfen.

UNIVERSITÄT BONN

Überschreiben und Verdecken

Virtuelle Memberfunktionen sind wieder virtuell. Verdeckende Memberfunktionen können virtuell sein.

```
class B {
public:
  virtual void f();   // B::f ist virtuell
  void g();           // B::g ist nicht virtuell
  void h();           // B::h ist nicht virtuell
};

class D : public B {  // D erbt von B
public:
  void f();           // D::f überschreibt B::f und ist virtuell
  virtual void g();   // D::g verdeckt B::g und ist virtuell
  void h();           // D::h verdeckt B::h und ist nicht virtuell
// void h() override; // Deklaration erzeugt Compilerfehler, weil B::h nicht virtuell ist
};
```

UNIVERSITÄT BONN

Beispiel

```
// Header
class GeschlKurve {
public:
  virtual double getUmfang() const;
};

class Polygon : public GeschlKurve {
public:
  Polygon();
  double getUmfang() const override;
private:
  std::vector<double> kanten;
};

class Kreis : public GeschlKurve {
public:
  Kreis();
  double getUmfang() const override;
private:
  double radius;
};
```

UNIVERSITÄT BONN

Beispiel

```
// Quellen
double GeschlKurve::getUmfang() const {
  std::cout << "Leider unklar was man hier Implementierungen soll..." << std::endl;
  return -1.0;
}

Polygon::Polygon() : kanten({3.0, 4.0, 5.0}) {}

double Polygon::getUmfang() const {
  double umfang = 0.0;
  for (const double& e : kanten) {
    umfang += e;
  }
  return umfang;
};

Kreis::Kreis() : radius(6.0) {}

double Kreis::getUmfang() const {
  return radius*radius*3.141592;
}
```

UNIVERSITÄT BONN

Beispiel

```
// Demo
void drucke_umfang(const GeschlKurve& K) {
  std::cout << "Der Umfang beträgt " << K.getUmfang() << "." << std::endl;
}

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

  drucke_umfang(g);
  drucke_umfang(p);
  drucke_umfang(k);
}
```

Zwischenfrage

Wie definiert man virtuelle Memberfunktionen, die keine direkte Implementierung zulassen?

UNIVERSITÄT BONN

Rein virtuelle Funktionen

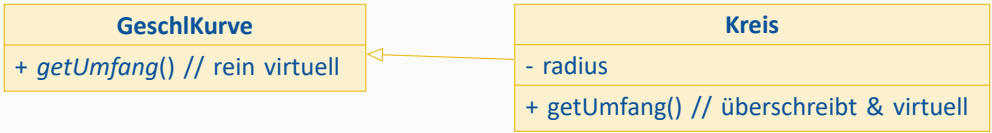
An unserem Beispiel haben wir bereits Folgendes festgestellt. Bei der Beschreibung der Interaktionsschnittestelle eines Obertyps wollen wir ausdrücken können, dass gewisse Memberfunktionen keine Implementierung zulassen, sondern diese von den Subtypen konkretisiert werden müssen. Diese virtuellen Memberfunktionen nennt man auch **rein virtuell**.

Wir diskutieren später Modellierungsansätze, die sich rein virtuelle Memberfunktionen zu Nutze machen.

UNIVERSITÄT BONN

Rein virtuelle Funktionen in UML

In UML werden rein virtuelle Memberfunktionen kursiv geschrieben, oder enthalten die Eigenschaftsbeschreibung {abstract}.



UNIVERSITÄT BONN

Rein virtuelle Funktionen in C++

In C++ werden virtuelle Memberfunktionen rein virtuell, indem die Funktionsdeklaration mit `= 0` beendet wird.

Klassen mit rein virtuellen Memberfunktionen können nicht instanziiert werden.

```
class GeschlKurve {
public:
  virtual double getUmfang() const = 0; // Rein virtuelle Funktion
};

...

int main() {
  GeschlKurve g; // Compilerfehler, denn GeschlKurve enthält eine rein virtuell Funktion
  Polygon p;     // Da getUmfang in Polygon überschrieben wird, gelingt die Instanziierung
}
```

Zusammenfassung

Wir haben gelernt, wie man polymorphe Typen in C++ realisiert

Virtuelle Memberfunktionen die keine Implementierung zulassen sind rein virtuell

Haben Sie Fragen?

Subtypenbeziehungen

Subtypenbeziehung durch Typkonkretisierung im Speicher

Offene Frage

Wie werden polymorphe Typen unterhalb von C++ realisiert?

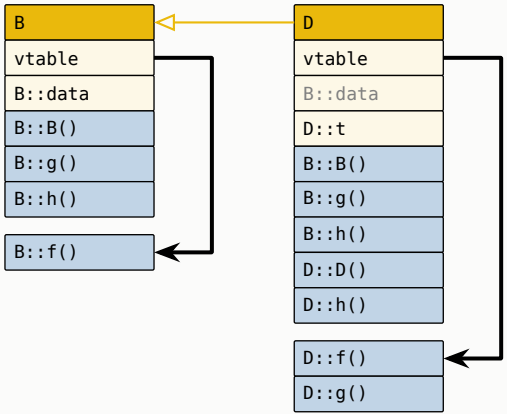
UNIVERSITÄT BONN

Überschreiben und Verdecken

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;
};
```



Überschriebene Funktionen bei Parameterübergabe

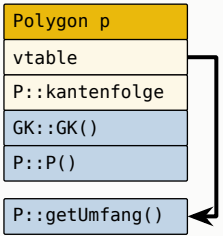
```
void call_by_value(const GeschlKurve K) {
  std::cout << "Der Umfang ist " << K.getUmfang();
}

void call_by_ref(const GeschlKurve& K) {
  std::cout << "Der Umfang ist " << K.getUmfang();
}

void call_by_ptr(std::shared_ptr<GeschlKurve> K) {
  std::cout << "Der Umfang ist " << K->getUmfang();
}

int main() {
  Polygon p = ...;  ←
  call_by_value(p);
  call_by_ref(p);
  call_by_ptr(std::make_shared<Polygon>(p));
}
```

Wir kürzen `GeschlKurve` durch `GK` und `Polygon` durch `P` ab.



Überschriebene Funktionen bei Parameterübergabe

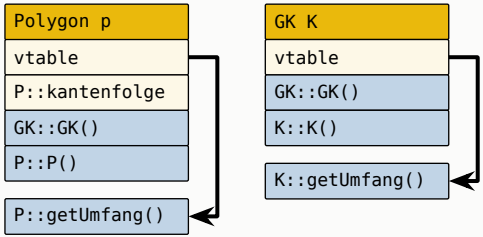
```
void call_by_value(const GeschlKurve K) {
  std::cout << "Der Umfang ist " << K.getUmfang();
}

void call_by_ref(const GeschlKurve& K) {
  std::cout << "Der Umfang ist " << K.getUmfang();
}

void call_by_ptr(std::shared_ptr<GeschlKurve> K) {
  std::cout << "Der Umfang ist " << K->getUmfang();
}

int main() {
  Polygon p = ...;
  call_by_value(p); ←
  call_by_ref(p);
  call_by_ptr(std::make_shared<Polygon>(p));
}
```

Wir kürzen `GeschlKurve` durch `GK` und `Polygon` durch `P` ab.



Überschriebene Funktionen bei Parameterübergabe

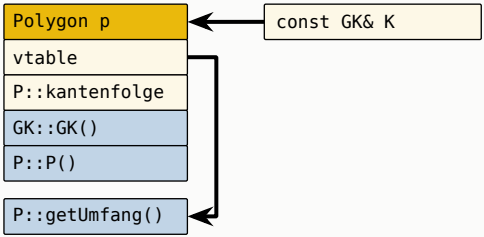
```
void call_by_value(const GeschlKurve K) {
  std::cout << "Der Umfang ist " << K.getUmfang();
}

void call_by_ref(const GeschlKurve& K) {
  std::cout << "Der Umfang ist " << K.getUmfang();
}

void call_by_ptr(std::shared_ptr<GeschlKurve> K) {
  std::cout << "Der Umfang ist " << K->getUmfang();
}

int main() {
  Polygon p = ...;
  call_by_value(p);
  call_by_ref(p);   ←
  call_by_ptr(std::make_shared<Polygon>(p));
}
```

Wir kürzen `GeschlKurve` durch `GK` und `Polygon` durch `P` ab.



UNIVERSITÄT BONN

Überschriebene Funktionen bei Parameterübergabe

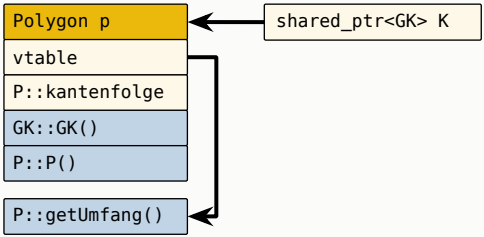
```
void call_by_value(const GeschlKurve K) {
  std::cout << "Der Umfang ist " << K.getUmfang();
}

void call_by_ref(const GeschlKurve& K) {
  std::cout << "Der Umfang ist " << K.getUmfang();
}

void call_by_ptr(std::shared_ptr<GeschlKurve> K) {
  std::cout << "Der Umfang ist " << K->getUmfang();
}

int main() {
  Polygon p = ...;
  call_by_value(p);
  call_by_ref(p);
  call_by_ptr(std::make_shared<Polygon>(p));   ←
}
```

Wir kürzen `GeschlKurve` durch `GK` und `Polygon` durch `P` ab.



UNIVERSITÄT BONN

Statisches und Dynamisches Binden

Welche (Member)funktion genau aufgerufen wird, steht bei nicht-virtuellen Funktionen zur Compilezeit fest. Bei virtuellen Memberfunktionen wird, mithilfe der `vtable`, zur Laufzeit bestimmt, welche Funktion genau aufgerufen wird.

Die Funktionsauswahl zur Compilezeit nennt man **statisches Binden** und die Funktionsauswahl zur Laufzeit nennt man **dynamisches Binden**.

Polymorphe Typen in Java und Python

Bei den Programmiersprachen **Java** und **Python** werden praktisch alle Objekte auf dem Heap angelegt und die Parameterübergabe geschieht praktisch immer via Call by Reference. Außerdem sind praktisch alle Memberfunktionen virtuell. Insbesondere können Memberfunktionen praktisch nicht verdeckt werden.

Also bieten **Java** und **Python** weniger Freiheit aber auch eine höhere Nachvollziehbarkeit bei Programmiernfänger:innen. In **C++** ist das Verdecken von Memberfunktionen erlaubt um höhere Ausführungsperformance zu bieten (in Anwendungsgebieten die keine virtuellen Funktionen benötigen).

Zusammenfassung

Nicht-virtuelle Funktionen werden durch statisches Binden aufgerufen

Virtuelle Memberfunktionen werden durch dynamisches Binden aufgerufen. Sie werden in einer Tabelle von Funktionspointern organisiert, die im Attributblock des Objekts referenziert wird.

Haben Sie Fragen?

Reaktion auf Intermezzo

Lautes Gemurmel und Gelächter wird sowohl bei inhaltlichen Themen als auch nicht inhaltlichen Themen als störend empfunden

Subtypenbeziehungen

Statischer und dynamischer Typ

Offene Frage

Variablen und Expressions haben einen festen Typ, verhalten sich bei polymorphen Typen aber manchmal wie Subtypen. Wie drückt man aus, über welchen Typ man spricht?

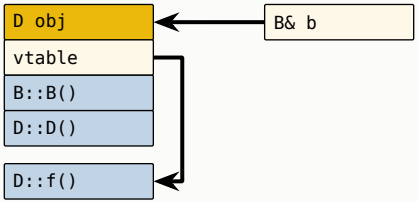
UNIVERSITÄT BONN

Motivation

Angenommen `D` **überschreibt** die virtuelle Funktion `B::f`.

```
void h(B& b) {
  b.f();
}

int main() {
  D obj;
  h(obj);
}
```



Wenn wir das `D`-Objekt `obj` als `B`-Referenz übergeben, hat das Objekt `b` den Typ `B`-Referenz. Beim Aufruf der virtuelle Funktion `b.f` verhält es sich jedoch wie eine `D`-Referenz.

Dass `b` eine `B`-Referenz ist steht zur **Compilezeit** fest. Welche virtuelle Funktion beim Ausdruck `b.f` aufgerufen wird, steht zur **Laufzeit** fest.

Statischer und dynamischer Typ

Wir betrachten polymorphe Klassen `Derived` $\rightarrow\triangleright$ `Base` und betrachten

```
Derived d;
Base& b = d;
```

Das Objekt `b` ist eine `Base`-Referenz und verhält sich (bei dieser Zuweisung) wie eine `Derived`-Referenz. Der **statische Typ** von `b` ist `Base` und der **dynamische Typ** von `b` ist `Derived`. Dieselbe Unterscheidung führen wir bei Zeigern ein.

Im Allgemeinen steht der **statische Typ** zur **Compilezeit** fest und der **dynamische Typ** erst zur **Laufzeit**.

UNIVERSITÄT BONN

Statischer und dynamischer Typ in objektorientierten Sprachen

Der dynamische Typ kann in objektorientierten Programmiersprachen zu Laufzeit bestimmt werden. Um den dynamischen Typ eines Objekts zu bestimmen, verwendet man

- in **Java** die, in allen Objekten vorhandene, Memberfunktion `getClass()`,
- in **Python** die Funktion `type(EXPRESSION)` und
- in **C++** die Funktion **`typeid`**`(EXPRESSION)`.

In **C++** wird der statische Typ durch die Operation `decltype(EXPRESSION)` erhalten. In **Java** kann der statische Typ nicht direkt bestimmt werden und **Python** besitzt praktisch keine statischen Typen.

UNIVERSITÄT BONN

Beispiel

```
#include <typeinfo> // Nötig um typeid zu verwenden.
// Weitere Includes

// polymorphic type: declares a virtual member
class Base1 {
  public virtual void f() {}
};

// polymorphic type: inherits a virtual member
class Derived1 : public Base1 {
};


// non-polymorphic type
class Base2 {
};

// non-polymorphic type
class Derived2 : public Base2 {
};
```

UNIVERSITÄT BONN

Beispiel

```
int main() {
  Derived1 obj1; // object1 created with type Derived1
  Derived2 obj2; // object2 created with type Derived2

  Base1& bref1 = obj1; // b1 refers to the object obj1
  Base2& bref2 = obj2; // b2 refers to the object obj2

  std::shared_ptr<Base1> bptr1 = std::make_shared<Derived1>();
  std::shared_ptr<Base2> bptr2 = std::make_shared<Derived2>();

  // Mit decltype(EXPRESSION) erhalten wir denden statischen Typ der Expression
  std::cout << "Expr type of bref1: " << typeid(decltype(bref1)).name() << std::endl // Base1
            << "Expr type of bref2: " << typeid(decltype(bref2)).name() << std::endl // Base2
            << "Obj  type of bref1: " << typeid(bref1).name() << std::endl           // Derived1
            << "Obj  type of bref2: " << typeid(bref2).name() << std::endl           // Base2
            << "Obj  type of bprt1: " << typeid(*bptr1).name() << std::endl          // Derived1
            << "Obj  type of bprt2: " << typeid(*bptr2).name() << std::endl;         // Base2
}
```

UNIVERSITÄT BONN

Dynamische Typen stehen erst zur Laufzeit fest

An diesem Beispiel sehen wir, das der dynamische Typ erst zur Laufzeit feststeht.

```
#include <typeinfo> // Nötig um typeid zu verwenden
// Weitere Includes
// ...

int main() {
  std::shared_ptr<GeschlKurve> k;
  // Statischer Typ von  k: std::shared_ptr<GeschlKurve>
  // Statischer Typ von *k: GeschlKurve

  int zufall = ...;

  if (zufall % 2 == 0) {
    k = std::make_shared<Polygon>();
  } else {
    k = std::make_shared<Kreis>();
  }
  std::cout << "Dynamischer Typ von *k: " << typeid(*k).name() << std::endl;
}
```

UNIVERSITÄT BONN

Statischer Typ und auto

Der statische Typ von Expressions steht zur Compilezeit fest. Diesen Umstand kann man sich bei der Variablendefinition zunutze machen.

Um eine Variable `var` anzulegen, die den statischen Typ der Expression EXPR hat, verwendet man den Platzhalter `auto`.

```
class X { ... };
X x;
// Die Expression hat den statischen Typ X, also wird auto durch X ersetzt
auto var1 = x;
// Die Expression hat den statischen Typ X, also wird const auto durch const X ersetzt
const auto var2 = x;
// Die Expression hat den statischen Typ X, also wird auto durch X& ersetzt
auto& var3 = x;
```

UNIVERSITÄT BONN

Beispiel

Bei Containern ist es üblich die Elemente als (konstante) Referenz zu durchlaufen. Dazu verwendet man üblicherweise einen Range-Based-For-Loop. Da der statische Elementtyp feststeht, kann hier `auto&` oder `const auto&` verwendet werden.

```
std::vector<int> v({1,2,3,4,5,6});

for (auto& elem : v) {
  elem = elem*elem;
}

for (const auto& elem : v) {
  std::cout << elem << " ";
}
std::cout << std::endl;
```

Zusammenfassung

Der statische Typ einer Expression steht zur Compilezeit fest. Das macht sich das Keyword `auto` zunutze.

Der dynamische Typ einer Expression steht zur Laufzeit fest

Haben Sie Fragen?

Subtypenbeziehungen

Zusammenfassung Typerweiterung und Typkonkretisierung

Ziel

Wir fassen die Inhalte zur Typerweiterung und Typkonkretisierung zusammen

UNIVERSITÄT BONN

Zusammenfassung 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

Zusammenfassung 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.

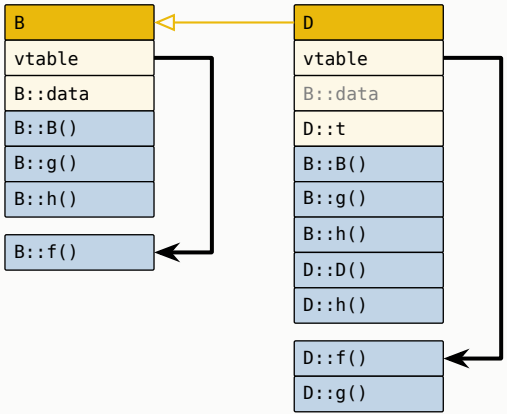
UNIVERSITÄT BONN

Zusammenfassung 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;
};
```



Haben Sie Fragen?