Zugriffsattribut `protected`

Analog zu `public` und `private` lassen sich die Komponenten von Klassen auch mit `protected` als Sichtbarkeit markieren.

Abseits von Auswirkungen bzgl. abgeleiteter Klassen verhält sich eine als `protected` markierte Klassenkomponente identisch als wäre sie mit `private` markiert.

Vereinbarung von Klassenkomponenten als *explicitly defaulted*

Bei der Definition einiger Klassenkomponenten kann statt eines Körpers `= default` angegeben werden. Es wird dann die ansonsten implizite Standardimplementierung für diese Klassenkomponente verwendet.

Dies ist natürlich nur bei Klassenkomponenten möglich, die ansonsten einen implizite Standardimplementierung hätten. Dies sind der Standardkonstruktor, der Kopierkonstruktor, der Zuweisungsoperator (sowohl kopierend wie auch verschiebend), der Verschiebekonstruktor und der Destruktor.

Beispiel (Explizite Standardimplementierung). Wir definieren den Standardkonstruktor der Klasse `A` als explicitly defaulted.

`explicitly_defaulted_demo.cpp`

```cpp
#include <iostream>

using namespace std;

class A {
  public:
    A() = default;
    void f() { cout << "A" << endl; }
};

int main() {
  A a{}; a.f();
}
```

```
A
```

Deleted Definitions

Funktionen und Methoden können als *deleted* definiert werden. Hierfür wird statt eines Körpers `= delete` angegeben.

Programme in denen eine Funktion bzw. eine Methode verwendet wird, die als deleted definiert wurde, werden vom Compiler nicht akzeptiert.

Beispiel (Gelöschter Kopierkonstruktor). Wir vereinbaren eine Klasse `A`, jedoch den Kopierkonstruktor von `A` als deleted. Im Programm wird der Kopierkonstruktor zur Initialisierung der Variable `a2` jedoch verwendet, sodass sich das Programm nicht übersetzen lässt.

deleted_copy_demo.cpp

```cpp
#include <iostream>

using namespace std;

class A {
  A(const A&) = delete;
};

int main() {
  A a1{};
  A a2{a1};
}
```

```
deleted_copy_demo.cpp: In function ‘int main()’:
deleted_copy_demo.cpp:11:10: error: use of deleted function ‘A::A(const A&)’
   11 |   A a2{a1};
      |          ^
deleted_copy_demo.cpp:6:3: note: declared here
    6 |   A(const A&) = delete;
      |   ^
```

Abgeleitete Klassen

⟨Basis⟩ → “:” ⟨BasisSpec⟩ {“,” ⟨BasisSpec⟩}
⟨BasisSpec⟩ → [⟨AccessSpec⟩] [“`virtual`”] ⟨ClassName⟩
⟨AccessSpec⟩ → “`private`” | “`protected`” | “`public`”

Durch Einschub der Basisklassenangabe ⟨Basis⟩ in die Definition einer Klasse (nach Schlüsselwort `class` und Namen der Klasse jedoch vor der öffnenden geschweiften Klammer) wird die neu definierte Klasse erweitert durch die Komponenten der angegebenen Basis-Klasse(n). Man sagt die neu Klasse *erbt* die Komponenten der Basisklasse (*Vererbung*).

Die Sichtbarkeit der geerbten Klassenkompenenten bestimmt sich nach der Sichtbarkeit der jeweiligen Komponente der Basisklasse und der Sichtbarkeitsangabe in ⟨AccessSpec⟩. Voreinstellung bei Weglassen der ⟨AccessSpec⟩ ist `private`.

Die resultierende Sichtbarkeit ist das „Minimum“ aus ⟨AccessSpec⟩ und Sichtbarkeit der Komponente der Basisklasse, wie folgt:

Abgeleitet als …	Sichtbarkeit in Basisklasse		
	`private`	`protected`	`public`
`private`	—	`private`	`private`
`protected`	—	`protected`	`protected`

Abgeleitet als …	Sichtbarkeit in Basisklasse		
	`private`	`protected`	`public`
`public`	—	`protected`	`public`

Komponenten der abgeleiteten Klasse können auf geerbte Komponenten, die in der Basisklasse als `private` markiert wurden, abgesehen von einer Markierung als `friend`, nie zugreifen.

Beispiel (Einfache Ableitung). Die abgeleitete Klasse `B` erbt die Komponente `x` von ihrer Basisklasse `A`.

simple_derived_class_demo.cpp

```cpp
#include <iostream>

using namespace std;

class A {
  public:
    int x;
};
class B : public A {};

int main() {
  B b;
  b.x = 7;
  cout << b.x << endl;
}
```

```
7
```

Unterobjekte/Konstruktoren für abgeleitete Klassen

Objekte abgeleiteter Klassen enthalten vollständige *Unterobjekte* aller ihrer Basisklassen. Die Namen von geerbten Komponenten referenzieren bei Verwendung die jeweiligen Komponenten des passenden Unterobjekts. Bei Konstruktion eines Objektes einer abgeleiteten Klasse müssen auch die jeweiligen Unterobjekte konstruiert werden. Dies geschieht jeweils durch Aufruf eines der Konstruktoren der Basisklasse.

Die Konstruktoren aller Basisklassen werden, zur Initialisierung der Komponenten der Unterobjekte, ausgeführt bevor nicht-geerbte Komponenten initialisiert werden (in der gleichen Reihenfolge in der sie in ⟨BasisSpec⟩ angegeben sind).

Durch die Initialisierungsliste der Konstruktoren der abgeleiteten Klasse können den Konstruktoren der Basisklassen, die verwendet werden um die Unterobjekte zu initialisieren, jeweils Parameter übergeben werden. Kommt eine Basisklasse in der Initialisierungsliste einer der Konstruktoren der abgeleiteten Klasse nicht vor, so wird dort stattdessen der Standardkonstruktor der Basisklasse verwendet.

Beispiel (Initialisierung bei abgeleiteten Klassen). Die abgeleitete Klasse `B` erbt die Komponente `x` von ihrer Basisklasse `A`. Der Konstruktor mit einem Parameter der Klasse `B` verwendet zur Initialisierung des Unterobjekts der Klasse `A` des anzulegenden Objektes der Klasse `B` den Konstruktor mit einem Parameter der Klasse `A` und übergibt dem Konstruktor einen aus seinem eigenen Parameter berechneten Wert.

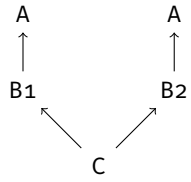
init_derived_class_demo.cpp

```cpp
#include <iostream>

using namespace std;

class A {
  public:
    int x;

    A() : x(7) {}
    A(int x_) : x(x_) {} };
class B : public A {
  public:
    B() {}
    B(int x_) : A(x_ / 2) {} };

int main() {
  B b1{}; cout << b1.x << " ";
  B b2{7}; cout << b2.x << endl;
}
```

```
7 3
```

Die „ist-Unterobjekt-von“ Relation bzgl. eines Objektes einer (mehrfach) abgeleiteten Klasse lässt sich visualisieren als *directed acyclic graph* (DAG).

Beispiel. Wir visualisieren die Struktur eines Objektes einer mehrfach abgeleiteten Klasse `C`. Die mehreren Vorkommen von `A` entsprechen der Tatsache, dass `C` mehrere Unterobjekte der Klasse `A` enthält.



(a) DAG für `C`

```cpp
class A { public: int x; };
class B1 : public A {};
class B2 : public A {};
class C : public B1, public B2 {};
```

(b) Zugehörige Klassendeklarationen

Überschattung von Namen von Komponenten abgeleiteter Klassen

In abgeleiteten Klassen können Komponenten mit Namen vereinbart werden, die identisch sind zu jenen geerbter Komponenten. Die Komponenten der abgeleiteten Klasse haben syntaktisch Vorrang vor den geerbten Komponenten; die geerbten Komponenten werden *überschattet*. Soll dennoch eine Komponente eines Unterobjekts referenziert werden, lässt sich dies durch Verwendung des scope resolution operators (`::`) angewandt auf den Komponentennamen erreichen.

Beispiel (Überschattung in abgeleiteten Klassen). Die abgeleitete Klasse `B` erbt die Komponente `x` von ihrer Basisklasse `A`, hat jedoch selbst zusätzlich auch eine Komponente namens `x`.

shadowing_derived_class_demo.cpp

```cpp
#include <iostream>

using namespace std;

class A { public: int x; };
class B : public A {
  public: int x;
};

int main() {
  B b;
  b.x = 7; b.A::x = 3;
  cout << b.x << " " << b.A::x
       << endl;
}
```

```
7 3
```

Bei mehrfacher Vererbung kann es vorkommen, dass ein Objekt mehrere Unterobjekte enthält die jeweils gleichnamige Komponenten enthalten. Verwendete Komponentennamen müssen eindeutig zugeordnet werden können. Hierfür wird die Komponente mit dem kürzesten Pfad im „ist-Unterobjekt-von“ DAG vorgezogen. Der scope resolution operator (`::`) kann verwendet werden um einen Präfix des Pfads vorzugeben bis Eindeutigkeit erreicht ist.

Beispiel. Wir referenzieren die Komponente `x` aus zwei unterschiedlichen Unterobjekten der Klasse `A` in einem Objekt der abgeleiteten Klasse `C` indem als hinreichender (für Eindeutigkeit) Präfix des Pfads durch den DAG jeweils `B1` bzw. `B2` angegeben wird.

attribute_multiple_derived_class_demo.cpp

```cpp
#include <iostream>

using namespace std;

class A { public: int x; };
class B1 : public A {};
class B2 : public A {};
class C : public B1, public B2 {};

int main() {
  C c;
  c.B1::x = 3; c.B2::x = 7;
  cout << c.B1::x << " " << c.B2::x
       << endl;
}
```

```
3 7
```

Implizite Typkonvertierungen von Zeigern bzw. Referenzen auf Objekte abgeleiteter Klassen

Für A eine Basisklasse von B kann ein Wert vom Typ „Zeiger auf B“ stets implizit konvertiert werden zu einem Wert vom Typ „Zeiger auf A“. Analog auch für Referenzen.

Die Indirektion über Zeiger bzw. Referenzen ist notwendig, a priori ist keine implizite Typumwandlung der Objekte selbst möglich.

Beispiel. Wir wandeln einen Wert vom Typ „Referenz auf B“ implizit um in einen Wert vom Typ „Referenz auf A“.

typeconversion_derived_class_demo.cpp

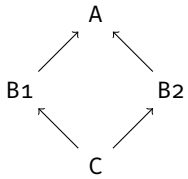
```cpp
#include <iostream>

using namespace std;

class A { public: int x; };
class B : public A {};

int main() {
  B b;
  b.x = 7;
  A& a = b;
  cout << a.x << endl;
}
```

```
7
```

Virtuelle Basisklassen

Für alle Vorkommen einer Klasse A im DAG eines Objektes, bei denen die *letzte Kante* zu A mit `virtual` markiert wurde, nur *genau ein* Unterobjekt der Klasse A. Gibt es weitere Vorkommen der Klasse A, bei denen die letzte Kante zu A nicht mit `virtual` markiert wurde, so enthält das Objekt noch weitere Unterobjekte der Klasse A.

Beispiel. Wir visualisieren die Struktur eines Objektes einer mehrfach abgeleiteten Klasse `C`. Da `B1` und `B2` jeweils `virtual` abgeleitet sind von `A`, enthalten Objekte der Klasse `C` nur jeweils ein Unterobjekt der Klasse `A`.



(a) DAG für `C`

Virtuell abgeleitete Klasse

```cpp
class A { public: int x; };
class B1 : public virtual A {};
class B2 : public virtual A {};
class C : public B1, public B2 {};
```

(b) Zugehörige Klassendeklarationen

virtual_derived_class_demo.cpp

```cpp
#include <iostream>

using namespace std;

  Virtuell abgeleitete Klasse

int main() {
  C c;
  c.x = 7;
  cout << c.x << endl;
}
```

```
7
```

Virtuelle Methoden

Gewöhnlicherweise wird anhand des Typs eines Objekts ausgewählt welche Methode ausgeführt wird, wenn sie anhand ihres Namens auf ein Objekt angewandt wird. Dies gilt auch im Falle von überschatteten Methodennamen bei abgeleiteten Klassen und bei Aufruf durch einen Zeiger bzw. durch eine Referenz.

Ist jedoch eine Methode in ihrer Deklaration als `virtual` markiert, so sind zunächst automatisch alle sie überschattenden Methoden ebenfalls `virtual`. Unter als `virtual` markierten Methoden erfolgt die Auswahl nicht anhand des Typs, sondern anhand dessen welcher Konstruktor zuletzt auf das konkrete Objekt angewandt wurde. Es wird dann die Methode der Klasse gewählt dessen Konstruktor zuletzt auf das Objekt ausgeführt wurde.

Beispiel (Polymorphe Objekte). Wir rufen die überschattete, `virtual` Methode `f` auf ein Objekt der von `A` abgeleiteten Klasse `B` auf. Es wird hierbei erwartungsgemäß die Methode `f` aus `B` gewählt, genauso wie wenn `f` nicht `virtual` wäre. Es wird dann eine Referenz auf das Objekt `b` der Klasse `B` implizit umgewandelt in einen Wert `ap` vom Typ „Referenz auf `A`“. Wäre `f` nicht `virtual` würde ein Aufruf von `f` durch `ap` die Implementierung von `f` aus `A` wählen. Da `f` jedoch `virtual` ist, wird stattdessen die Implementierung aus `B` gewählt, da das Objekt das `ap` referenziert mit einem Konstruktor der Klasse `B` initialisiert wurde.

virtual_fun_demo.cpp

```cpp
#include <iostream>

using namespace std;

class A {
  public:
    virtual char f() { return 'A'; }
};
class B : public A {
  public:
    virtual char f() { return 'B'; }
};
```

```
int main() {
  B b; A& ap = b;
  cout << b.f() << ap.f() << endl;
}
```

```
BB
```

Attribute `override` und `final`

Methoden für die in einer der (transitiven) Basisklassen eine Methode mit selben Namen und Parametertypliste existiert, welche `virtual` ist, sind automatisch ebenfalls `virtual`.

Methoden-Deklarationen, die mit `override` markiert sind, müssen eine gleichnamige Methode aus einer der Basisklassen überschatten. Ansonsten wird das Programm vom Compiler nicht akzeptiert.

Methoden-Deklarationen, die mit `final` markiert sind, dürfen von keiner gleichnamigen Methode in einer ihrere abgeleiteten Klassen überschattet werden. Ansonsten wird das Programm ebenfalls nicht vom compiler akzeptiert.

Beispiel. Wir überschatten die Methode `f` in der von `A` abgeleiteten Klasse `B` markiert sowohl als `override`, wie auch `final`.

override_final_demo.cpp

```cpp
#include <iostream>

using namespace std;

class A {
  public:
    virtual char f() { return 'A'; }
};
class B : public A {
  public:
    char f() override final { return 'B'; }
};

int main() {
  B b; A& ap = b;
  cout << b.f() << ap.f() << endl;
}
```

```
BB
```

Object Slicing

Per Definition nehmen die impliziten Standarddefinitionen des Kopierkonstruktors und des Zuweisungsoperators ein Objekt der jeweiligen Klasse als *Referenzparameter*. Es können daher Referenzen auf Objekte abgeleiteter Klassen implizit konvertiert werden in Referenzen auf ihre Basisklassen bei Verwendung des Kopierkonstruktors bzw. des Zuweisungsoperators.

Da sich der Kopierkonstruktor bzw. Zuweisungsoperator jedoch nur mit den Attributen der jeweiligen Klasse befasst können Attribute hierbei „verloren gehen“. Man spricht von *object slicing*.

Beispiel (Konvertierung bei Zuweisungsoperator). Es wird für die Zuweisung der Variable `a` durch die Referenz `ap` auf `a` mit einer Kopie von `b` der Zuweisungsoperator von `A` verwendet (die Auswahl des Zuweisungsoperators geschieht anhand des Typs von `ap`).

assign_conversion_demo.cpp

```cpp
#include <iostream>

using namespace std;

class A { public:
  virtual char f() { return 'A'; }
};
class B : public A { public:
  virtual char f() { return 'B'; }
};
int main() {
  B b; A a; A& ap = a; ap = b;
  cout << b.f() << ap.f() << endl;
}
```

```
BA
```

Beispiel (Konvertierung bei Kopierkonstruktor). Es wird für die Initialisierung der Variable `a` der Kopierkonstruktor von `A` verwendet.

copy_conversion_demo.cpp

```cpp
#include <iostream>

using namespace std;

class A { public:
  virtual char f() { return 'A'; }
};
class B : public A { public:
  virtual char f() { return 'B'; }
};
int main() {
  B b; A a = b;
  cout << b.f() << a.f() << endl;
}
```

```
BA
```

Beispiel (Verletzung einer Invariante durch object slicing). Durch object slicing beim Zuweisen von `b1` durch die Referenz `ap` vom Typ „Referenz auf `A`“ werden nur die zu `A` gehörigen Attribute „über“ den aktuellen Wert von `b1` kopiert. Die in `b1` durchaus und auch weiterhin vorhandenen zusätzlichen Attribute (`approx_x`) werden nicht verändert und werden dadurch inkonsistent mit denen aus `A` (`x`).

violated_invariant_assignment.cpp

```cpp
#include <iostream>

using namespace std;

class A {
  protected:
    double x;

  public:
    A(double x_) : x(x_) {}
};

class B : public A {
  private:
    int approx_x;

  public:
    B(double x_) : A(x_), approx_x(x_) {}

    friend ostream& operator<<(ostream& stream, const B& b) {
      return stream << b.x << " =~ " << b.approx_x;
    }
};

int main() {
  B b1{3.0}, b2{7.0};
  A& ap = b1;
  ap = b2;

  cout << b1 << endl;
}
```

```
7 =~ 3
```

Rein virtuelle Methoden

Bei der Definition einer `virtual` Methode f innerhalb einer Klasse A kann statt eines Körpers `= 0` angegeben werden. Es ist dann nicht möglich Objekte zu initialisieren von A oder von A abgeleiteten Klassen, sofern diese f nicht überschatten. Werte vom Typ „Zeiger auf A“ bzw. „Referenz auf A“ bleiben jedoch möglich.

Man nennt f eine *rein virtuelle Methode* und A eine *abstract base class (ABC)*.

Beispiel (Rein virtuelle Methode). Wir vereinbaren die Methode `f` der Klasse `A` als rein virtuell.

pure_virtual_demo.cpp

```cpp
#include <iostream>

using namespace std;

class A { public:
  virtual char f() = 0;
};
class B : public A { public:
  virtual char f() { return 'B'; }
};

int main() {
  B b; cout << b.f() << endl;
}
```

```
B
```

Virtuelle Destruktoren

Die Auswahl des Destruktors bei Aufruf für ein Objekt, sowohl implizit wie auch explizit mit dem `delete`-Operator, erfolgt nach dem selben Verfahren wie der Aufruf einer Methode. Destruktoren können also, wie Methoden, `virtual` sein.

Beispiel (Vermeidung eines Speicherlecks durch virtuelle Destruktoren). Wir vereinbaren die Basisklasse `A` mit `virtual` Destruktor. Als Implementierung wählen wir die, sonst implizite, Standardimplementierung.

In der von `A` abgeleiteten Klasse `B` fügen wir ein Attribut `b` vom Typ „Zeiger auf `double`“ hinzu. Im Konstruktor von `B` wird Speicher für `b` zugewiesen und im Destruktor wieder freigegeben.

Im Hauptprogramm wird ein Objekt der Klasse `B` erzeugt, jedoch nur ein `unique_ptr<A>` darauf gespeichert. Zum Ende der Lebensdauer der Variable `ap` (d.h. am Ende des Hauptprogramms) wird automatisch der Destruktor der Klasse `unique_ptr` auf `ap` angewandt, insbesondere dadurch `delete` aufgerufen auf das Objekt, auf das `ap` zeigt. Da das Objekt auf das `ap` zeigt initialisiert wurde mit einem Konstruktor aus `B`, wird korrekt der `virtual` Destruktor der Klasse `B` gewählt, sodass der zugewiesene Speicher auch ordnungsgemäß wieder freigegeben wird.

virtual_destructor_demo.cpp

```cpp
#include <iostream>
#include <memory>

using namespace std;

class A {
  public:
    virtual ~A() = default;
```

```
    virtual void f() = 0;
};
class B : public A {
  private:
    double* b;

  public:
    static constexpr int l = 100;
    B() {
      b = new double[l];
      for (int i = 0; i < l; i++)
        b[i] = i;
    }
    ~B() { delete[] b; }

    void f() {
      for (int i = 0; i < l; i++)
        cout << b[i] << endl;
    }
};

int main() {
  unique_ptr<A> ap{new B{}};
  ap->f();
}
```

```
==…== Memcheck, a memory error detector
==…== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==…== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==…== Command: virtual_destructor_demo
==…==
==…==
==…== HEAP SUMMARY:
==…==     in use at exit: 0 bytes in 0 blocks
==…==   total heap usage: 4 allocs, 4 frees, 77,616 bytes allocated
==…==
==…== All heap blocks were freed -- no leaks are possible
==…==
==…== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```