7. OBJEKTORIENTIERUNG II

Inhalt

- Mehr zu Vererbung und Subtyping
  - Abstrakte Klassen
  - Finale Klassen und Methoden
  - Mehrfachvererbung und das Diamond-Problem
- Nachtrag: Ad-hoc Polymorphismus
- Spezielle Mitgliedsfunktionen
  - Copy-Konstruktor und -Zuweisungsoperator
  - Move-Konstruktor und -Zuweisungsoperator
- Mehr zu `const`
  - `const`-Funktionen
  - `const`-Korrektheit

Abstrakte Klassen

- Manchmal macht es **keinen** Sinn, dass von einer Klasse Objekte erstellt werden können
- **Beispiel**: Schauen wir uns nochmals die Klasse `Shape` an

```
class Shape {
 public:
  virtual ~Shape() = default;
  virtual double area();
};
```

  - `Shape` schreibt vor, dass alle Figuren eine Methode `area` haben
  - Die Methode `Shape::area` lässt sich aber nicht sinnvoll definieren
  - Objekte des Typs `Shape` machen auch keinen Sinn
- In solchen Fällen benutzt man sogenannte **abstrakte Klassen** („abstract class“)
  - Abstrakte Klassen können **alles**, was konkrete Klassen auch können
  - Zusätzlich deklarieren sie **rein virtuelle** („pure virtual“) Methoden

Abstrakte Klassen

- In unserem Beispiel bietet es sich an, die Methode `Shape::area` als **rein virtuell** zu deklarieren

```
class Shape {
 public:
  virtual ~Shape() = default;
  virtual double area() = 0;
};
```

  - Das `= 0` hinter dem Methodennamen bedeutet, dass die Methode zwar deklariert ist, aber nicht definiert werden muss
  - Deshalb können wir die (sinnlose) Definition von `Shape::area` in der Quelltextdatei `Shape.cpp` nun löschen

- Konkrete Subklassen einer abstrakten Klasse müssen dann **alle** rein virtuellen Methoden überschreiben und definieren

Abstrakte Klassen

- Auch ein Destruktor kann als **rein virtuell** deklariert werden

```
class Shape {
 public:
  virtual ~Shape() = 0;
};
```

- Dann **muss** allerdings eine Definition angegeben werden, z.B.

```
Shape::~Shape() = default;
```

- Ist das nicht etwas **unlogisch**? Nein!
  - Konstruktoren und Destruktoren sind **spezielle** Funktionen, die automatisch entlang der Vererbungshierarchie aufgerufen werden
  - Dieser Automatismus funktioniert nicht mit undefinierten Destruktoren
- **GTK** Mit rein virtuellen Destruktoren kann man abstrakte Klassen deklarieren, die sonst **keine** rein virtuellen Methoden haben

Finale Klassen und Methoden

- Man kann auch **verbieten**, dass weitere Subklassen einer Klasse erstellt werden, indem man die Klasse als `final` deklariert

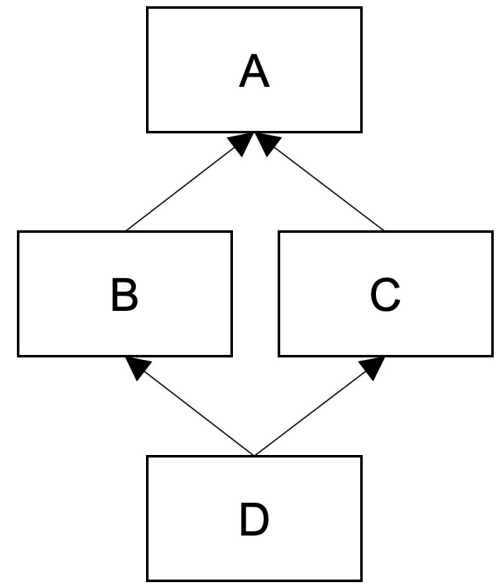
```
class Square final : public Rectangle {
 public:
  Square(XYPoint upleft, int length);
  ~Square() override = default;
};
```

- Mit `final` kann man ebenfalls **verhindern**, dass einzelne Methoden überschrieben werden können

```
class Rectangle : public Shape {
 public:
  double area() final;
};
```

Mehrfachvererbung und das Diamond-Problem

- Mehrfachvererbung kann zu **mehrdeutigen** Situationen führen
- **Das Diamond-Problem**
  - Die Klassen `B` und `C` erben von Klasse `A`
  - Beide Subklassen **überschreiben** dabei die virtuelle Methode `A::m`
  - Die Klasse `D` erbt sowohl von Klasse `B` **und** Klasse `C`
  - Welche Definition der Methode `m` erbt Klasse `D`, wenn sie die Methode **nicht** selber überschreibt?



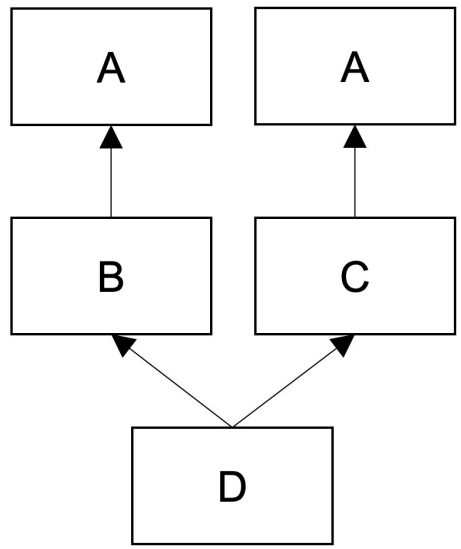
- Der Methodenaufruf `d.m();`, bei dem `d` eine Instanz von `D` ist, führt zu einer Fehlermeldung des Compilers

```
error: request for member ‘m’ is ambiguous
```

- Es gibt **zwei** Möglichkeiten, diese Mehrdeutigkeit zu beheben
  - Eine der beiden Methoden **explizit** aufrufen, z.B. mit `d.B::m();`
  - Die Methode in der Klasse `D` **überschreiben** und **definieren**

Mehrfachvererbung und das Diamond-Problem

- Selbst wenn die Klassen `B` und `C` die Methode `A::m` **nicht** überschreiben, ist der Aufruf `d.m();` mehrdeutig
- Kling merkwürdig, ist aber so
  - Eine Instanz von `D` besteht aus dem Teil für `D` selbst und **rekursiv** aus Teilen für die Superklassen von `D`
  - Bei der Auflösung der Superklassen wird der **statische** Typ der Klassen `B` und `C` betrachtet
  - Die Instanz der Klasse `D` hat also **zwei** Teile für die Klasse `A`, da sie die Superklassen von `B` **und** `C` ist



- **Virtueller Vererbung** („virtual inheritance“) stellt sicher, dass nur ein Teil für die gemeinsame Superklasse erstellt wird

```
class A { void m(); };
class B : public virtual A { ... };
class C : public virtual B { ... };
class D : public B, public C { ... };
```

Nachtrag: Ad-hoc Polymorphismus

- In C++ können (alle Typen von) Funktionen **überladen** werden

```
void add(int lhs, int rhs, int* res);
void add(double lhs, double rhs, double* res);
```

  - Gleicher Name und Rückgabetyp
  - Andere Argumenttypen

- Bei überladenen Funktionen wählt der Compiler die passende Implementierung aufgrund der Typen der übergebenen Werte

```
int i = 5;
double d;
add(i, 2, &i);      // Die 1. Funktion wird aufgerufen
add(2.0, i, &d);    // Die 2. Funktion wird aufgerufen
```

- GTK Diese „Vielgestaltigkeit“ von Funktionen nennt man **Ad-hoc Polymorphismus**
- GTK Ad-hoc Polymorphismus und Objektorientierung sind **orthogonale** Konzepte

Nachtrag: Ad-hoc Polymorphismus

- Mögliche Probleme und Fehler bei überladenen Funktionen
  - Funktionen, die sich **nur** im Rückgabewert unterschieden

```
float m();
double m();    // Compiler-Fehler!
```

  - Mehrdeutige Situationen

```
void m(int x, double y) { ... }
void m(double x, int x) { ... }

m(1, 1);       // Beide Varianten passen gleich gut!
```

- **Aber**: Überladene Funktionen können sich durch verschiedene „Const-heit“ („constness“) unterscheiden (siehe später)

Spezielle Mitgliedsfunktionen

- Jede Klasse hat **sechs** Mitgliedsfunktionen, die implizit definiert sind, auch wenn man sie nicht explizit implementiert
  - **Defaultkonstruktor**
  - **Defaultdestruktor**
  - **Copy-Konstruktor**
  - **Copy-Zuweisungsoperator**
  - **Move-Konstruktor**
  - **Move-Zuweisungsoperator**
- Den Defaultkonstruktor und -destruktor haben wir bereits in der letzte Woche kennengelernt
- Nun schauen wir uns noch die anderen vier Mitgliedsfunktionen an

Copy-Konstruktor

- Wie alle anderen Konstruktoren heißt der **Copy-Konstruktor** wie die Klasse und hat keinen Rückgabewert
- Er nimmt **genau ein** Argument, das eine konstante Referenz auf eine Instanz der Klasse ist

```
class Circle : public Shape {
 public:
  Circle(const Circle& c);  // Copy-Konstruktor
};
```

- Wird der Copy-Konstruktor nicht explizit definiert, gibt es einen **impliziten** Copy-Konstruktor, der folgendes macht
  - Kopiert alle Mitgliedsvariablen der übergebenen Instanz
  - Für jede Mitgliedsvariable, die keinen Basistyp hat, wird rekursiv der Copy-Konstruktor des betreffenden Typs aufgerufen

Copy-Konstruktor

- Prinzipiell kann ein Objekt auf **zwei Arten** kopiert werden
  - **Flache Kopie** („shallow copy“): Das kopierte Objekt enthält nur Verweise auf die Mitgliedsvariablen des ursprünglichen Objekts
  - **Tiefe Kopie** („deep copy“): Alle Mitgliedsvariablen des ursprünglichen Objekts werden ihrerseits tief kopiert
- **Achtung**: Die Semantik des Copy-Konstruktor verlangt aber, dass eine tiefe Kopie gemacht wird!
- GTK Insbesondere bei Objekten mit Zeigern auf dynamischen Speicher muss sichergestellt werden, dass tief kopiert wird
  - Der implizite Copy-Konstruktor kopiert nur die Adressen der Zeiger
  - Kann zu falschem Verhalten führen, z.B. bei einer verketteten Liste
  - Kann eine Speicherschutzverletzung verursachen („double free“)

Copy-Konstruktor

- Der Copy-Konstruktor wird immer dann aufgerufen, wenn es das Objekt, in das kopiert wird, vorher **noch nicht** gab

```
Circle c1({8, 4}, 3);  // "Normaler" Konstruktor

Circle c2(c1);         // Copy-Konstruktor von c2
                       // mit Argument c1
Circle c3 = c1;        // Copy-Konstruktor von c3
                       // mit Argument c1
c2 = c3;               // Nicht der Copy-Konstruktor!
```

Copy-Zuweisungsoperator

- Wenn man in ein Objekt kopieren will, das es vorher schon gab, braucht man den **Copy-Zuweisungsoperator**

```
class Circle : public Shape {
 public:
  Circle& operator=(const Circle& c);
};
```

- Die Implementierung des Copy-Zuweisungsoperator ist typischerweise ähnlich zu der des Copy-Konstruktors
- **GTK** Auf die folgenden Punkte muss man aber zusätzlich achten
  - Möglicherweise muss Speicher, den das Objekt (`this`) bereits reserviert hatte, freigegeben werden
  - Die Zuweisung muss auch funktionieren, wenn ein Objekt sich selbst zugewiesen oder wenn in Serie zugewiesen wird
  - Jede Implementierung von `operator=` muss eine Referenz auf das Objekt selbst (`*this`) zurückgeben

„Rule of Three“

- Wenn man **eine** der folgenden drei speziellen Mitgliedsfunktionen definiert, sollte man **alle drei** implementieren
  - **Destruktor**
  - **Copy-Konstruktor**
  - **Copy-Zuweisungsoperator**
- Diese sogenannte **„Rule of Three“** oder **„The Law of the Big Three“** ist folgendermaßen begründet
  - Vestößt man gegen die Regel, passen die implizit definierten Funktionen ziemlich sicher nicht zu den explizit definierten Funktionen
  - Wenn zum Beispiel der Copy-Konstruktor Speicher für eine tiefe Kopie reserviert, wird dieser vom impliziten Destruktor **nicht** freigegeben

Expliziter Konstruktor

- Bei „normalen“ Konstruktoren mit nur einem Argument empfiehlt es sich, `explicit` davor zu schreiben (Woche 6)
- Der Grund dafür ist die **automatische Typkonversion** in C++
  - Nehmen wir an, dass wir eine Klasse mit folgendem Konstruktor haben

    ```
    Person(int id);  // "normaler" ctor ohne explicit
    ```

  - Zudem definieren wir diese Funktion

    ```
    void process(Person person);
    ```

  - Nun rufen wir die Funktion (aus Versehen) so auf

    ```
    process(42);
    ```

  - Ohne `explicit` kompiliert dieses Beispiel, da der Aufruf von `process` die `42` in `Person(42)` konvertiert
- Mit `explicit` können wir solche **impliziten Aufrufe** des Konstruktors verbieten

Zwischenfazit

- Copy-Konstruktor und Copy-Zuweisungsoperator

```
class Circle : public Shape {
 public:
  Circle(const Circle& rhs);
  Circle& operator=(const Circle& rhs);
}
```

- In Zuweisungen der Form `lhs = rhs` ist die Instanz (`this`) der Klasse die linke und der Parameter `rhs` die rechte Seite
- **Teuer** für Klassen mit dynamischem Speicher, da `new` für jedes „Teilobjekt“ aufgerufen werden muss
- Das ursprüngliche Objekt bleibt **unverändert** (`rhs` ist `const` in beiden Signaturen)

Move-Operationen

- Häufig „kopiert“ man, braucht aber eigentlich das ursprüngliche Objekt hinterher **nicht** mehr, z.B. bei `swap`

```
Circle c1;
Circle c2;
Circle tmp = c1;      // c1 wird gleich ueberschrieben
c1 = c2;              // c2 wird gleich ueberschrieben
c2 = tmp;             // tmp wird nicht mehr benutzt
```

- In diesem Fall könnte man den Speicher, den das Objekt auf der rechten Seite dynamisch reserviert hat, direkt **weiterbenutzen**
- Dafür gibt es in C++ die **Move-Operationen**

Move-Operationen

- Move-Konstruktor und Move-Zuweisungsoperator

```
class Circle : public Shape {
 public:
  Circle(Circle&& rhs);
  Circle& operator=(Circle&& rhs);
}
```

- `Circle&&` ist eine sogenannte **rvalue-Referenz**, d.h., in etwa „Referenz auf ein Objekt, aus dem man verschieben darf“
- Das Objekt `rhs` muss nach der Operation noch gültig sein, darf aber einen **anderen Wert** als vorher haben oder **leer** sein
- Für Objekte mit dynamischem Speicher kann man Verschieben häufig **effizienter** implementieren als Kopieren

Move-Operationen

- Move-Operationen werden in **zwei** Fällen aufgerufen
  - Bei temporären Objekten ohne Namen

```
Circle c1;
c1 = Circle({0, 0}, 3);                    // Move-Zuweisung
```

  - Bei expliziter Verwendung von `std::move`

```
Circle c2(std::move(c1));                  // Move-Konstruktor
```

- GTK Die Funktion `std::move` verschiebt überhaupt gar nichts
  - Sie zeigt nur an, dass ein Objekt mittels Move-Konstruktor oder -Zuweisungsoperator verschoben werden **kann**
  - Das Gleiche könnte man auch mit der expliziten **Typkonvertierung** `static_cast<Circle&&>(c1)` erreichen
  - Ist die rechte Seite als `const` deklariert, ergibt `std::move` eine Kopie

```
const Circle c3({3, 3}, 2);
Circle c4 = std::move(c3);                 // Copy-Zuweisung
```

„Rule of Five“

- **„Rule of Three“**
  - Wenn man Destruktor, Copy-Konstruktor oder -Zuweisungsoperator implementiert, sollte man immer alle drei implementieren
  - Der Grund dafür ist **konsistentes Verhalten** der drei Operationen
- **„Rule of Five“**
  - Wenn man die Operationen der Rule of Three implementiert, sollte man auch immer den Move-Konstruktor und -Zuweisungsoperator implementieren
  - Der Grund dafür ist **bessere Performance**
- **„Rule of Zero“**
  - Klassen, die keinen dynamischen Speicher manuell verwalten (mit `new` und `delete`), sollten keine der fünf Operationen implementieren
  - Der Grund dafür ist **weniger fehleranfälliger Quellcode**

Löschen von Copy- und Move-Operationen

- Wenn ein Objekt nicht kopiert oder verschoben werden soll, muss man die entsprechenden (impliziten) Operationen **löschen**

```
class DoNotCopy {
 private:
  DoNotCopy(const DoNotCopy& rhs) = delete;
  DoNotCopy& operator=(const DoNotCopy& rhs) = delete;
};
```

- **GTK** Nun verstehen wir auch, wie der `std::unique_ptr` verhindern kann, dass man ihn kopiert, anstatt ihn zu verschieben

Mehr zu `const`

- Das Schlüsselwort `const` haben wir bereits bei den Deklarationen von **Variablen** und **Funktionsargumenten** gesehen
  - Bei einer **Variablendeklaration** bedeutet `const`, dass man die Variable nach der initialen Zuweisung nicht mehr ändern darf

```
const double PI = 3.14159;
PI = 3.14159265358979323846;      // Compile-Fehler!
```

  - Vor einem **Funktionsargument** bedeutet `const` entsprechend, dass man diese lokale Variable nicht ändern darf

```
int sum(const int a, const int b) {
  a += b;                         // Compile-Fehler!
  return a;
}
```

Mehr zu `const`

- Wir haben `const` auch bei **Zeigern** und **Referenzen** gesehen
  - Bei **Zeigern** bedeutet `const`, dass man den Inhalt nicht verändern darf, die Adresse aber schon

```
const char* s = "PK_1";
s[3] = '2';                    // Compile-Fehler!
s = "PK_2";                     // Das funktioniert
```

  - Für den sehr seltenen Fall, dass man es umgekehrt haben möchte, schreibt man das `const` nach dem `*`

```
char s[5] = { 'P', 'K', '_', '1' , 0 };
char* const p = s;
p[3] = '2';                    // Das funktioniert nun
p = p + 1;                      // Compile-Fehler!
```

  - Es geht auch beides

```
const char* const s ="PK_1";
```

  - Bei **Referenzen** heißt `const`, dass man den Inhalt nicht ändern darf

```
void print(const std::string& text) { ... }
```

const-Methoden

- Bei einer **Methodendeklaration** bedeutet `const`, dass diese Methode keine Mitgliedsvariablen der Klasse verändern darf

```
class Shape {
 public:
  virtual double area() const = 0;
};
```

- Ändert die Methode trotzdem was, gibt es einen Compile-Fehler

```
error: assignment of member in read-only object
```

- Auf einer Instanz, die als `const` deklariert ist, können nur `const`-Methoden aufgerufen werden

```
const Circle circle({0, 0}, 3);
circle.area();      // Funktioniert, da const-Methode
```

const-Methoden

- Manchmal gibt es Methoden, die zwar einige **„Hilfsvariablen“** nicht aber den eigentlichen Zustand der Klasse ändern
- Wenn man diese „Hilfsvariablen“ als `mutable` („veränderbar“) deklariert, kann eine solche Methode trotzdem `const` sein

```
class Circle : public Shape {
 private:
  mutable int numCallsToArea_ = 0;
 public:
  double area() const override;
};

double Circle::area() const {
  numCallsToArea_++;
  ...
}
```

const-Methoden

- Überladene Funktionen können sich durch **verschiedene „Const-heit“**(„constness“) unterscheiden

```
class IntList {
 public:
  int& operator[](size_t i);                    // (1)
  const int& operator[](size_t i) const;        // (2)
};
```

- Der Compiler wählt die passende Variante der überladenen Methode anhand der „Const-heit“ des Objekts aus

```
IntList vlist(2);
const IntList clist(2);
vlist[0] = clist[0];   // links:  (1) -> int&,
                       // rechts: (2) -> const int&
clist[1] = vlist[1];   // Compile-Fehler!
                       // links:  (2) -> const int&
                       // rechts: (1) -> int&
```

`const`-Korrektheit

- Ein Programm ist **`const`-korrekt**, wenn ...
  - alle Variablen, die nach der initialen Zuweisung nicht mehr verändert werden *sollten*, als `const` deklariert sind,
  - alle Mitgliedsfunktionen, die eine Instanz der Klasse nicht verändern *sollten*, als `const` deklariert sind und
  - alle Rückgabewerte, die der aufrufende Code nicht verändern können *sollte*, als `const` deklariert sind.
- Ob das *sollte* in den Bedingungen gelten muss, ist dabei anhand des Sinn und Zwecks des Programms zu prüfen
- Ob die Kriterien für `const`-Korrektheit erfüllt sind, wird von Programmierer:innen nicht vom Compiler geprüft
- **Ausnahme**: Basistypen (`int`, `float`, `double` etc.) müssen in Funktionsargumenten nicht als `const` deklariert werden

Programmieren!

- Schreibe die verkettete Liste aus Woche 4 als Klasse um, die die folgenden zusätzlichen Operationen anbietet
  - Copy-Konstruktor und -Zuweisungsoperator
  - Move-Konstruktor und -Zuweisungsoperator
  - „Addition“ von zwei Listen (`+` und `+=`)