6. OBJEKTORIENTIERUNG I

Inhalt

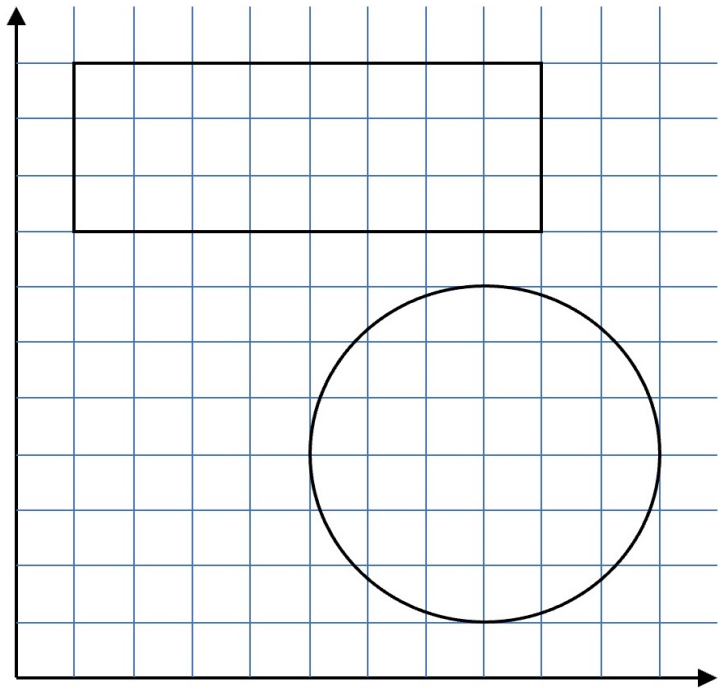
- Objektorientierte Programmierung
  - Wiederverwendung von Quellcode
  - Was sind Klassen und Objekte?
- Klassen
  - Membervariablen und Methoden
  - Datenkapselung: `public`, `private` und Friend-Klassen
  - Konstruktoren und Destruktoren
- Vererbung
  - Wiedervendung von Quellcode
  - Subtyping: `public` vs. `protected` und `private` Vererbung
  - Dynamisches Dispatching: virtuelle Methoden

Objektorientierte Programmierung

- **Ziel**: Softwareentwicklung soll modularer, wiederverwendbar und einfacher zu verstehen und zu pflegen gemacht werden
- Die **objektorientierte Programmierung** führt neue Konzepte ein
  - **Klassen** geben uns die Möglichkeit, eigene Datentypen (im Sinne von gültigem Wertebereich und erlaubten Operationen) zu definieren
  - **Datenkapselung** erlaubt uns, die (interne) Implementierung einer Klasse hinter einer Schnittstelle, die den Zugriff regelt, zu verstecken
  - **Vererbung** ermöglicht uns, Quellcode wiederzuverwenden, ohne ihn kopieren zu müssen
  - **Subtyp-Polymorphismus** befähigt uns, generisch zu programmieren, d.h., wir können auf Werten operieren, ohne diese im Detail zu kennen
- GTK Objektorientierte Programmierung ist deshalb auf einer **höheren Abstraktionsebene** angesiedelt als die imperative Programmierung die wir bis jetzt gemacht haben

Warum brauchen wir das?

- Schreibe ein C++-Programm, das
  - geometrische Figuren (Kreis, Dreieck, Rechteck, Polygon etc.) anlegt,
  - diese in einem `std::vector<Shape> shapes` speichert und
  - von allen die Fläche berechnet, indem es über `shapes` iteriert.
- **Annahmen**
  - Diskreter, zweidimensionaler Raum
  - Wir beschränken uns erstmal auf **Kreise** und **Rechtecke**
  - Jede geometrische Figur ist durch **zwei** Punkte in der Ebene beschrieben
    - **Kreis**: Zentrum und Punkt auf dem Kreis
    - **Rechteck**: Eckpunkte (oben links) und (unten rechts)
- Wir verwenden nur **Sprachkonzepte**, die wir bis jetzt kennengelernt haben



Manöverkritik

- Unser Programm erfüllt die Spezifikation und funktioniert korrekt!
- Wo liegt hier also das Problem?
  - Schlechte **Wiederverwendbarkeit**: Dritte müssen die (interne) Implementierung (z.B. die Semantik der beiden Punkte) kennen
  - Schlechte **Modularität**: Notwendige Änderungen beschränken sich nicht auf wenige, isolierte Stellen im Quelltext
  - Schlechte **Erweiterbarkeit**: Das Programm kann nur erweitert werden, wenn es im Quelltext vorliegt
- **Szenario**: Was müssten wir tun, um Dreiecke einzuführen?
  - `Type` muss um `triangle` ergänzt werden
  - `Shape.points` muss nun (für alle Figuren) drei Punkte speichern
  - `area` muss um einen weiteren Fall erweitert werden
- Mit objektorientierter Programmierung geht das besser!

Klassen

- Eine Klasse bündelt Daten und die dazugehörigen Operationen
  - **Mitgliedsvariablen** („member variables“)
  - **Mitgliedsfunktionen** („member functions“) / **Methoden** („methods“)
- Wir können unsere `Shape` von vorhin, wie folgt, umbauen

```
class Shape {
  Type type_;
  XYPoint points_[2];
  double area();
};
```

- Die **Deklaration** einer Klasse geschieht in der Header-Datei
  - Meistens heißt die Header-Datei gleich wie die Klasse: `Shape.h`
  - Das Semikolon (`;`) am Ende der Deklaration muss da stehen!
- **GTK** Der Unterstrich (_) ist eine übliche Konvention, um Mitgliedsvariablen von anderen Variablen zu unterscheiden

Klassen

- Die **Definition** der Klasse erfolgt in einer Quelltextdatei

```
#include "Shape.h"

double Shape::area() {
  switch (type_) { ... }
}
```

- **Beachte!**
  - Der Klassenname ist Teil des Funktionsnamen und **muss** bei der Definition angegeben werden
  - Methoden können auf Mitgliedsvariablen direkt zugreifen

- Zum Vergleich nochmals die „alte“ Version der Funktion

```
double area(const Shape* shape) {
  switch (shape->type) { ... }
}
```

Klassen

- Die Werte einer Klasse nennt man **Objekte** oder **Instanzen**
- Instanzen einer Klasse erzeugt man folgendermaßen

```
Shape circle;
```

  - Ganz „normale“ Variablendeklaration: links der Typ, rechts der Name
  - Auch hier wird bereits Speicherplatz für ein `Shape`-Objekt reserviert und `circle` ist der Name dieses Stück Speichers

- Analog zu Strukturen erfolgt der Zugriff auf Mitgliedsvariablen und Methoden über den Mitgliedszugriff-Operator `.`

```
circle.type_ = Type::circle;
std::cout << circle.area() << std::endl;
```

Datenkapselung

- Die Mitgliedsvariablen und Methoden einer Klasse sind standardmäßig **nicht** von außen zugreifbar
  - Der Programmcode auf der letzten Folie funktioniert so (noch) nicht
  - GTK Das ist der in Woche 4 erwähnte, einzige Unterschied zwischen Klassen und Strukturen, in denen standardmäßig alles zugreifbar ist
- **Zugriffsmodifikatoren** kontrollieren die Berechtigungen

```
class Shape {
 private:
  Type type_;
  XYPoint points_[2];
 public:
  area();
};
```

  - `public` gewährt Zugriff **von überall** (über `.`)
  - `private` gewährt Zugriff **nur aus der Klasse** (d.h. alle Instanzen)
  - `protected` besprechen wir später in dieser Vorlesung

Datenkapselung

- **Datenkapselung** („encapsulation“) bzw. **Information Hiding** ist ein zentrales Prinzip objektorientierter Programmierung
  - Nur so viel Information und Zugriff wie nötig nach außen preisgeben
  - So viel wie möglich von der Implementierung „verstecken“
- **Mitgliedsvariablen** sind in der Regel immer `private`
  - Zugriff erfolgt über sogenannte Setter- und Getter-Methoden
  - **Vorteil**: Lese- und Schreibzugriff kann unabhängig voneinander kontrolliert werden (z.B. `public` Getter, kein Setter)
  - **Vorteil**: Klasse kann den internen Zustand konsistent verändern, falls z.B. Abhängigkeiten zwischen Mitgliedsvariablen bestehen
- **Methoden** können `private` oder `public` sein
  - Methoden, die Teil der Schnittstelle einer Klasse sind, sollten `public` sein, z.B. `Shape::area`, `List::insert` oder `Chess::start`
  - Interne oder Hilfsmethoden sollten `private` sein, z.B. `Chess::init`, `Account::update_balance` etc.

Datenkapselung

- Gewisse **externe** Funktionen oder Klassen brauchen manchmal Zugriff auf den **internen** Zustand unserer Klasse haben
  - Eine **Friend-Funktion** wird über den Namen, die Argumenttypen und den Rückgabetyp identifiziert
  - Eine **Friend-Klasse** wird über den Namen der Klasse identifiziert
- Das kann z.B. zum Testen ganz hilfreich sein

```
class Shape {
  friend class ShapeTest;
};
```

  - **Jede** Instanz einer Klasse mit dem Namen `ShapeTest` hat nun Zugriff auf den internen Zustand einer Instanz von `Shape`
  - Friend-Funktionen regeln der Zugriff etwas feingranularer

Klassenvariablen

- Neben Mitgliedsvariablen kann eine Klasse auch sogenannte **Klassenvariablen** deklarieren
  - Mitgliedsvariablen können für jede Instanz (jedes Mitglied) der Klasse **unterschiedliche** Werte annehmen
  - Klassenvariablen haben für alle Instanzen den **gleichen** Wert
- Klassenvariablen werden auch in der Header-Datei deklariert

```
class Shape {
 private:
  static const XYPoint ORIGIN;
};
```

- Und sie werden auch in der Quelltextdatei definiert

```
const XYPoint Shape::ORIGIN = { 0, 0 };
```

- GTK Natürlich kann man auch Klassenvariablen haben, die **nicht** `const` sind, z.B. zum Zählen aller Instanzen einer Klasse

Konstruktor

- Bevor man ein Objekt (sinnvoll) benutzen kann, muss man es typischerweise **initialisieren**
  - Da die Mitgliedsvariablen gekapselt sind, geht das nicht mehr direkt
  - Wir könnten eine extra Methode `init` dafür schreiben
  - Das ist fehleranfällig, da man vergessen könnte, `init` aufzurufen
- Dafür gibt es sogenannte **Konstruktoren** („constructors“), die bei der Definition eines Objektes garantiert aufgerufen werden

```
Shape circle(Type::circle, {8, 4}, {11, 4});
```

- Ein Konstruktor ist eine spezielle Funktion, die den gleichen Namen wie die Klasse hat
  - Eine Klasse kann **mehrere** Konstruktoren mit unterschiedlichen Argumenten haben
  - Der **passende** Konstruktor wird vom Compiler aufgrund der übergebenen Argumente ausgewählt

Konstruktor

- Die **Deklaration** eines Konstruktors erfolgt in der Header-Datei

```
class Shape {
 public:
  Shape(Type type, XYPoint p1, XYPoint p2);
};
```

  - Der Konstruktor heißt **genau gleich** wie die Klasse
  - Konstruktoren haben **keinen** Rückgabetyp
  - Bei genau einem Argument `explicit` davor schreiben (Woche 7)

- Die **Definiton** (Implementierung) erfolgt in der Quelltextdatei

```
Shape::Shape(Type type, XYPoint p1, XYPoint p2) {
  type_ = type;
  points_[0] = p1;
  points_[1] = p2;
}
```

Konstruktor

- Mitgliedsvariablen können im Konstruktor auch mit sogenannten **Initialisierungslisten** („initializer lists“) initialisiert werden

```
Shape::Shape(Type type, XYPoint p1, XYPoint p2) :
    type_{type}, points_{p1, p2} {
  // weitere Initialisierung
}
```

- GTK In der Regel ist diese Variante vorzuziehen
  - Mitgliedsvariablen in der **gleichen** Reihenfolge (von links nach rechts) initialisieren, wie sie (von oben nach unten) deklariert sind
  - **Alle** Mitgliedsvariablen initialisieren
- **Vorteile**
  - Initialisierungslisten sind typischerweise schneller als Zuweisungen
  - Bei Nicht-Basistypen wird der passende Konstruktor aufgerufen

Konstruktor

- Der sogenannte **Defaultkonstruktor** ist der Konstruktor ohne Argumente
  - Den Defaultkonstruktor gibt es, ohne dass man ihn **explizit** definiert
  - Er initialisiert jede Mitgliedsvariable entsprechend ihrer Definition in der Header-Datei
- Den impliziten Defaultkonstruktor kann man auf zwei Arten loswerden
  - Überschreiben mit einem expliziten Konstruktor **ohne** Argumente
  - Löschen, indem man einen Konstruktor **mit** Argumenten schreibt
- Manchmal braucht man den impliziten Defaultkonstruktor (ohne Argumente) **sowie** explizite Konstruktoren (mit Argumenten)

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

Destruktor

- Der **Destruktor** („destructor“) wird aufgerufen, wenn ein Objekt freigegeben wird
  - Der Sichtbarkeitsbereich des Objekts endet (Woche 2)
  - Ein Zeiger auf das Objekt wird mit `delete` gelöscht (Woche 4)
- Auch der Destruktor wird in der Header-Datei deklariert

```
class Shape {
 public:
  ~Shape();
};
```

und in der Quelltextdatei definiert

```
Shape::~Shape() { ... }
```

  - Der Destruktor mit einer Tilde (~) und heißt gleich wie die Klasse
  - Es gibt **nur einen** Destruktor pro Klasse
  - Destruktoren können **keine** Argumente haben

Destruktor

- Wenn man keinen expliziten Destruktor schreibt, gibt es immer einen impliziten **Defaultdestruktor** (wie Konstruktoren)
  - Für Mitgliedsvariablen von Basistypen tut der Defaultdestruktor nichts
  - Für alle anderen Mitgliedsvariablen ruft er rekursiv deren Destruktor (in umgekehrter Reihenfolge ihrer Deklaration in der Header-Datei) auf
- Ein explizit definierter Destruktor macht das auch, nachdem er den Code des Destruktors ausgeführt hat
- Auch den impliziten Defaultdestruktor kann man explizit definieren

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

- **GTK** Die Aufgabe des Destruktors ist, alles zu löschen, was im Konstruktor (oder während der Lebenszeit des Objekts) angelegt wurde

Programmieren!

- Wir schreiben das Programm `basic-shapes.cpp`, wie folgt, um
  - `shape.h` deklariert die Klasse `Shape`
  - `shape.cpp` definiert (implementiert) die Klasse `Shape`
  - `main.cpp` erzeugt geometrische Figuren und berechnet deren Fläche

Manöverkritik

- Durch **Datenkapselung** haben wir unseren Quellcode weniger fehleranfällig gemacht
  - Die Implementierung (`type_` und `points_`) einer `Shape` ist versteckt
  - Mit einer `Shape` kann nur über eine genau definierte Schnittstelle (Konstruktor und Methode `area`) interagiert werden
- Die **Erweiterbarkeit** unseres Codes ist immer noch ein Problem!
- Wenn wir **Dreiecke** unterstützen möchten, haben wir im Vergleich zur ersten Version nichts gewonnen!
  - `Type` muss um `triangle` ergänzt werden
  - `Shape.points_` muss nun (für alle Figuren) drei Punkte speichern
  - `Shape::area` muss um einen weiteren Fall erweitert werden
- Mit dem Konzept der **Vererbung** können wir unseren Code so strukturieren, dass solche Erweiterungen einfacher möglich sind

Vererbung

- Das Konzept der **Vererbung** („inheritance“) dient in objektorientierten Sprachen dazu, Quellcode wiederzuverwenden
  - Dazu leitet man **Subklassen** von **Superklassen** ab
  - Eine Klasse ohne Superklassen heißt **Basisklasse** („base class“)
  - Eine Subklasse **erbt** dabei Zustandsinformation (Mitgliedsvariablen) und Funktionalität (Methoden) von der Superklasse
  - Die Subklasse kann weitere, eigene Mitgliedsvariablen und Methoden definieren oder diejenigen der Superklasse **überschreiben**
- GTK C++ unterstützt die folgenden Vererbungsarten
  - **Mehrfachvererbung** („multiple inheritance“): eine Subklasse kann von mehreren Superklassen abgeleitet sein
  - **Multilevel-Vererbung** („multi-level inheritance“): eine Subklasse kann selbst wieder Superklasse für eine andere Subklasse sein
  - **Hierarchische Vererbung** („hierarchical inheritance“): von einer Superklasse können mehrere Subklassen abgeleitet sein

Vererbung

- Vererbung wird in C++ folgendermaßen deklariert (Header-Datei)

```
class Circle : Shape {
 public:
  Circle(XYPoint center, int radius);
  double area();
};
```

  - `Circle` ist nun eine Subklasse von `Shape`
  - `Circle` definiert einen neuen, verständlicheren Konstruktor
  - `Circle` überschreibt die Methode `area`

- Den Konstruktor können wir (in der Quelltextdatei) definieren, indem wir den **Konstruktor der Superklasse** `Shape` aufrufen

```
Circle::Circle(XYPoint center, int radius) :
    Shape(Type::circle, center,
          { center.x, center.y + radius }) {
}
```

Vererbung

- Beim Implementieren von `Circle::area` gibt es ein Problem!

```
double Circle::area() {
  return M_PI * (pow(points_[0].x - points_[1].x, 2) +
                 pow(points_[0].y - points_[1].y, 2));
}
```

- Die Mitgliedsvariable `points_` ist in `Circle` nicht zugreifbar, da sie in `Shape` als `private` deklariert wurde
- Wir haben zwei Möglichkeiten, um dieses Problem zu lösen
  - Wir könnten die Mitgliedsvariable in `Shape` als `protected` deklarieren, damit sie auch in Subklassen sichtbar ist
  - Wir könnten eigene (`private`) Mitgliedsvariablen für `Circle` deklarieren (und diejenigen in `Shape` entfernen)
- Die zweite Lösung ist besser
  - `protected` ist fast genauso gefährlich wie `public`
  - Jede Figur weiß selbst am besten, welche Informationen sie braucht

Vererbung

- Nach dieser **Refaktorisierung** („refactoring“) sehen unsere zwei Klassen nun so aus

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

class Circle : Shape {
 private:
  XYPoint center_;
  int radius_;
 public:
  Circle(XYPoint center, int radius);
  double area();
};
```

- Klasse `Rectangle` kann analog deklariert und definiert werden

Zugriffmodi für Vererbung

- Mist, nun kompiliert unser Hauptprogramm nicht mehr!

```
std::vector<Shape*> shapes;
Circle circle({8, 4}, 3);
shapes.push_back(&circle);
```

- Der Compiler beschwert sich in der dritten Zeile
  - `error: ‘Shape’ is an inaccessible base of ‘Circle’`
  - Bis jetzt haben wir Vererbung ausschließlich dazu genutzt, den Code der Superklasse in der Subklasse wiederzuverwenden
  - Damit unser Code funktioniert, muss `Circle` auch ein **Subtyp** von `Shape` sein, d.h., die Superklasse muss zugreifbar sein
- In C++ gibt es drei **Zugriffsmodi** für Vererbung
  - **Private Vererbung** (Default) setzt alle `public` und `protected` Mitglieder der Superklasse in der Subklasse auf `private`
  - **Protected Vererbung** setzt alle `public` und `protected` Mitglieder der Superklasse in der Subklasse auf `protected`
  - **Public Vererbung** lässt alle Zugriffsmodifikatoren so, wie sie sind

Subtyp-Polymorphismus

- **Subtyping** ist eine Relation zwischen zwei Datentypen, die es ermöglicht, dass Objekte des Supertyps `T` durch Objekte des Subtyps `S` ersetzt werden können
- In C++ besteht diese Relation zwischen zwei Klassen (Typen), wenn die Vererbung als `public` deklariert wird

```
class Circle : public Shape { ... }
```

- Nochmals Mist, nun kompiliert unser Code zwar wieder, macht aber nicht mehr, was er soll
  - Offenbar wird die Methode `Shape::area` aufgerufen, wenn wir auf ein `Circle`-Objekt als Wert vom Typ `Shape` zugreifen
  - Dieses Verhalten ist in der Regel nicht, was wir wollen
- Ein weiteres Konzept objektorientierter Programmierung schafft hier Abhilfe: **dynamisches Dispatching** („dynamic dispatching“)

Dynamisches Dispatching

- Durch Subtyp-Polymorphismus („Vielgestaltigkeit“) kann ein Objekt **mehrere Typen** haben
- Im folgenden Beispiel hat `shape` **zwei** Typen

```
Circle circle({ 0, 0 }, 5};
Shape* shape = &circle;
```

  - Der **deklarierte** oder **statische** Typ von `shape` ist `Shape*`
  - Der **tatsächliche** oder **dynamische** Typ von `shape` ist `Circle*`

- GTK Der statische Typ eines Objekts ist zur **Kompilierungszeit** bekannt, der dynamische Typ erst zur **Laufzeit**
- **Dynamisches Dispatching** verwendet den dynamischen Typ statt des statischen Typs, um die aufgerufene Methode auszuwählen
- Damit der C++-Compiler diesen Mechanismus in unser Programm einbaut, müssen wir sogenannte **virtuelle Methoden** verwenden

Virtuelle Methoden

- Um in unserem Beispiel **virtuelle Methoden** zu nutzen, müssen wir nur die Klassendeklarationen (Header-Datei) anpassen

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

class Circle : public Shape {
  ...
  double area() override;
};
```

- Deklaration von virtuellen Methoden
  - In der **Basisklasse** mit `virtual` (ganz vorne)
  - In **abgeleiteten Klassen** mit `override` (ganz hinten)
- GTK Bessere Sicherheit mit `override`: falls es die Methode in der Basisklasse nicht (mehr) gibt, gibt es eine Fehlermeldung!

Virtuelle Methoden

- **GTK** Sobald eine Klasse virtuelle Methoden hat, sollte sie auch einen **virtuellen Destruktor** haben

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

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

  - Auf diese Weise werden **alle** Destruktoren in der Vererbungshierarchie eines Objekts ausgeführt, wenn es gelöscht wird
  - Andernfalls kann es hier zu Speicherlecks kommen

Programmieren!

- Um die Erweiterbarkeit unseres Programms auf den Prüfstand zu stellen, führen wir Dreiecke als neue Figur ein
  - Schreibe `triangle.h` mit den entsprechenden Deklarationen
  - Schreibe `triangle.cpp` mit den entsprechenden Definition
  - Die Fläche eines Dreiecks ist gegeben durch

$$\sqrt{s \cdot (s-a) \cdot (s-b) \cdot (s-c)}$$

    - a, b und c sind die drei Seiten des Dreiecks
    - $s = \frac{1}{2}(a+b+c)$ ist der halbe Umfang
  - Lege in `main.cpp` ein Dreieck an, füge es in den bestehenden `std::vector<Shape*>` ein und gib seine Fläche aus