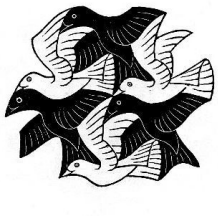
ZUSAMMENFASSUNG

PARADIGMEN DER SOFTWAREENTWICKLUNG

SOMMERSEMESTER 2025



Niklas Conrad
Fakultät Informatik
Bachelor of Science Inforamtik
September 2025

NIKLAS CONRAD

Paradigmen der Softwareentwicklung

FAKULTÄT INFORMATIK:
https://www.hs-schmalkalden.de/hochschule/fakultaeten/fakultaet-informatik

INHALTSVERZEICHNIS

1 EINFÜHRUNG

1.1 ABSTRAKTION

ABSTRAKTION ist das gezielte Unterdrücken oder Verbergen bestimmter Details eines Prozesses oder Artefakts, um andere Aspekte, Strukturen oder Eigenschaften klarer hervorzuheben.
Eigenschaften einer Abstraktion sind Prinzipien des:

- Verbergens oder Weglassens – ***Komplexitätsreduktion***
- Generalisierens – ***Wiederverwendbarkeit***
- Konzepts - Ïdee versus Realität- ***Flexibilität***

BEISPIELE

1. Die Landkarte ist nicht das Gelände
   - Eine Karte vereinfacht die Realität: Anstatt jeden Baum, jeden Stein oder jede Blume darzustellen, zeigt sie nur die wesentlichen Informationen, die man braucht.
   - Karten folgen allgemeinen Konventionen, zum Beispiel Linien mit Höhenangaben zur Darstellung der Topografie.
   - Eine Karte ist eine abstrakte Repräsentation eines Landabschnitts – ein Modell, das sich von Karte zu Karte unterscheiden kann.
2. Waschmaschine
   - „Die Waschmaschine starten“ ist ein abstrakter Vorgang. Die eigentliche Realität sind die vielen mechanischen Prozesse im Inneren der Maschine.
   - Es ist wesentlich einfacher, einen Knopf zu drücken, als manuell festzulegen, wann welche Ventile öffnen oder schließen oder wie sich die Trommel bewegen soll.
   - Auch wenn sich Waschmaschinen in ihrer Funktionsweise leicht unterscheiden (z.B. im Hinblick auf Energieeffizienz), gibt es fast immer die abstrahierte, vereinheitlichte Funktion „Start“. Diese ist eine Generalisierung über verschiedene Modelle und Marken hinweg.

Eine Anwendung ist eine Abstraktion einer Abstraktion einer Abstraktion...

- Benutzeroberfläche
- Hochsprache (z.B. JavaScript)
- Low-Level-Sprache (z.B. C)
- Maschinensprache
- Rechnerarchitektur (Register, Speicher, Recheneinheit usw.)
- Schaltungselemente (Logikgatter)
- Transistoren
- Festlörperphysik
- ...

1.2 ABSTRAKTIONSEBENEN

Abstraktionsebenen beziehen sich auf die verschiedenen Schichten oder Stufen, auf denen ein Computersystem betrachtet oder entwickelt werden kann. Jede Ebene verbirgt die Komplexität der darunterliegenden und bietet eine einfachere Schnittstelle für die darüberliegende. Damit wir, als Menschen, komplexe Softwaresysteme entwickeln können müssen wir Probleme analysieren, verstehen und kommunizieren können. Abstraktionen bieten die möglichkeit Probleme zu strukturieren.

Programmiersprachen bieten eine Reihe von Abstraktionsebenen an:

KONZEPTE beschreiben eine grundlegende Idee oder ein gedankliches Modell, das Möglichkeiten aufzeigt, wie Abstraktionen erstellt oder strukturiert werden können.
Konzepte beantworten also die Frage:

> *„**Welche** grundlegenden Möglichkeiten zur Abstraktion oder Strukturierung gibt es?“*

Typische Konzepte in der Informatik sind: *Objekt, Klasse, Schnittstelle, Vererbung, Typ, Nachricht*

PARADIGMEN definieren ein umfassendes Denkmodell oder eine allgemeine Sichtweise, wie Software strukturiert wird, um Ziele der Wartbarkeit, Erweiterbarkeit, Flexibilität, Robustheit, etc zu erreichen. Sie beantworten die Frage:

*„**Wie** kombiniere ich Konzepte und Abstraktionen zu einem stimmigen, einheitlichen Gesamtbild?“*

Typische Programmierparadigmen sind u.a.:

OBJEKTORIENTIERTES PARADIGMA (OOP) Kombination von Konzepten wie Objekten, Klassen, Nachrichten, Kapselung, Vererbung, Polymorphie, Late Binding, um komplexe Software modular, verständlich und wartbar zu gestalten.

FUNKTIONALES PARADIGMA Kombination von Konzepten wie reinen Funktionen, Zustandslosigkeit, Immutabilität, höherwertigen Funktionen, um fehlerarme und gut testbare Software zu erzeugen.

PROZEDURALES PARADIGMA Kombination von Konzepten wie Prozeduren, Modulen, und strukturierten Anweisungen, um eine schrittweise Abfolge von Operationen klar darzustellen.

DEKLARATIVES PARADIGMA Kombination von Abstraktionen, um zu beschreiben, was getan werden soll, anstatt explizit anzugeben, wie es umgesetzt wird (z.B. SQL oder logische Programmierung).

PRINZIPIEN beschreiben eher eine allgemeine Richtlinie oder eine grundlegende Regel, nach der die Konzepte praktisch angewendet werden. Prinzipien geben Orientierung für konkrete Entscheidungen, sie beantworten also die Frage:

*„**Wie** setze ich Konzepte und Paradigmen sinnvoll ein, unter welchen Voraussetzungen, und warum?“*

Typische Prinzipien in der Informatik sind:

- Information Hiding (Informationsversteckung)
- Separation of Concerns (Trennung der Belange)
- Single Responsibility Principle (Einzelverantwortung)
- DRY (Don’t Repeat Yourself)
- Open/Closed Principle (OCP)

2 GRUNDLAGEN VON C++

2.1 VARIABLEN, INITIALISIERUNG UND ZUSTAND

2.1.1 Definition von Variablen

In C++ bezeichnet man einen benannten Speicherbereich mit einem definierten Datentyp als **Variable**. Eine Variable ist ein Objekt, dem ein bestimmter **Zustand** (Wert) zugewiesen werden kann. Die Definition einer Variable legt den Speicherbereich an, jedoch ist der Zustand zunächst undefiniert, falls keine Initialisierung erfolgt.

```
int x; // Variable vom Typ int, undefinierter Zustand
```

Solche Variablen sollten *nicht verwendet werden*, bevor ein expliziter Wert zugewiesen wurde.

2.1.2 Initialisierungsarten

C++ erlaubt verschiedene Initialisierungsformen:

- **Copy-Initialisierung:** ruft den Copy-Konstruktor auf.
- **List-Initialisierung (Uniform Initialization):** ist strenger -> vermeidet implizite Konvertierungen.
- **Zero-Initialisierung:** Leere Initialisierung mit {}, die zu einem sicheren Standardwert führt.

Beispiele:

```
int a = 1;      // Copy-Initialisierung
int b(1);       // ebenfalls Copy-Initialisierung
int c {1};      // List-Initialisierung
int d {};       // Zero-Initialisierung -> d = 0
```

2.2 POINTER UND REFERENZEN IN C++

Pointer (Zeiger)

Ein **Pointer** ist eine Variable, die die *Adresse* eines anderen Objekts speichert. In C++ wird ein Zeiger mit dem Symbol * deklariert:

```
char* p;  // Pointer auf ein char
```

Ein Pointer kann auf ein Element eines Arrays zeigen:

```
char* p = &v[3];  // p zeigt auf das 4. Element von v
char x = *p;      // x erhaelt den Wert des Elements, auf das p zeigt
```

Der Präfix * bedeutet „Inhalt von“ (Dereferenzierung), das Präfix & bedeutet „Adresse von“. Pointer erlauben es, Datenstrukturen direkt zu manipulieren oder Funktionen effizient mit großen Datenmengen zu arbeiten zu lassen.

Referenzen

Eine **Referenz** ist ein Alias für eine existierende Variable. Sie wird mit dem Suffix & deklariert, z. B.:

```
int a = 42;
int& r = a;  // r ist eine Referenz auf a
```

Referenzen müssen beim Anlegen initialisiert werden und können später nicht auf andere Objekte zeigen. Anders als bei Zeigern ist keine Dereferenzierung nötig – der Zugriff erfolgt wie bei der Ursprungsvariablen.

Referenzen werden oft zur Übergabe von Argumenten in Funktionen verwendet, um Kopien zu vermeiden:

```
void sort(std::vector<double>& v); // Referenz als Parameter
```

Möchte man eine Referenz verwenden, aber gleichzeitige Änderungen verhindern, verwendet man eine `const`-Referenz:

```
double sum(const std::vector<double>& v); // Nur lesender Zugriff
```

2.3 FUNKTIONSAUFRUFKONVENTIONEN

In C++ können Funktionsparameter auf drei Arten übergeben werden: **Call by Value**, **Call by Reference** und **Call by Pointer**. Die Wahl beeinflusst, ob die Funktion auf das Originalobjekt oder nur auf eine Kopie zugreift.

Call by Value

Bei der Übergabe *by value* wird eine Kopie des Arguments erstellt. Änderungen innerhalb der Funktion haben keinen Einfluss auf das Original.

```
void modify(int x) {
    x = 42;
}

int main() {
    int a = 10;
    modify(a);   // a bleibt 10
}
```

Call by Reference

Hier wird eine Referenz auf das Originalobjekt übergeben. Änderungen wirken sich direkt auf das Original aus.

```
void modify(int& x) {
    x = 42;
}

int main() {
    int a = 10;
    modify(a);   // a ist jetzt 42
}
```

Besonders nützlich, wenn große Objekte übergeben werden und keine Kopie erzeugt werden soll.

Call by Pointer

Ein Zeiger auf das Original wird übergeben. In der Funktion muss dereferenziert werden. Zusätzliche Prüfung auf `nullptr` ist oft notwendig.

```
void modify(int* x) {
    if (x)
        *x = 42;
}

int main() {
    int a = 10;
    modify(&a);  // a ist jetzt 42
}
```

Diese Variante erlaubt explizit das Weiterreichen von `nullptr`, falls optionaler Zugriff gewünscht ist.

Vergleich

- **By Value:** sicher, aber ineffizient für große Objekte; keine Seiteneffekte.
- **By Reference:** direkt, effizient, lesbar; kein Nullwert möglich.
- **By Pointer:** flexibel, explizit; aber fehleranfälliger (Nullprüfung, Syntax).

2.4 SPEICHERUNG VON OBJEKTEN: STACK VS. HEAP

2.4.1 Grundlagen der Speicherbereiche

C++ unterscheidet bei der Speicherung von Objekten zwischen zwei primären Speicherbereichen:

- **Stack:** schneller, automatisch verwalteter Speicherbereich für lokale Objekte.
- **Heap:** dynamisch verwalteter Speicherbereich, der explizit mit `new` oder `std::make_unique` alloziert werden muss.

2.4.2 Primitive Datentypen auf dem Stack

Primitive Datentypen wie `int`, `float` oder `char` werden typischerweise auf dem **Stack** gespeichert, wenn sie lokal in Funktionen oder Blöcken deklariert werden.

```
void func() {
    int x {42};      // x liegt auf dem Stack
}
```

Diese Speicherbereiche werden automatisch freigegeben, wenn der Gültigkeitsbereich verlassen wird.

2.4.3 Objekte auf dem Stack

Auch komplexere Objekte (z.B. Klasseninstanzen) können auf dem Stack angelegt werden:

```
std::string name {"Alice"}; // liegt komplett auf dem Stack (interner Speicher im Heap)
```

Achtung: Viele Standardklassen wie `std::string` verwalten **internen Speicher** auf dem Heap, auch wenn das Objekt selbst auf dem Stack liegt.

2.4.4 Objekte auf dem Heap

Der Heap erlaubt dynamische Speicherallokation während der Laufzeit. Dazu muss Speicher manuell angefordert und ggf. wieder freigegeben werden:

```
int* ptr = new int {42};            // Speicher auf dem Heap
std::unique_ptr<int> smart = std::make_unique<int>(42); // sicherere Alternative
```

Ohne `smart pointer` besteht die Gefahr von Speicherlecks.

2.4.5 Unterschiede: Stack vs. Heap

Kriterium	**Stack**	**Heap**
Speicherverwaltung	automatisch	manuell (bzw. via Smart Pointer)
Lebensdauer	begrenzt auf Scope	über Funktionsgrenzen hinweg
Geschwindigkeit	sehr schnell	langsamer (Allokation aufwändiger)
Speichergröße	begrenzt (z.B. 1MB)	groß (abhängig vom System)
Gefahr	Stack Overflow	Memory Leak
Typischer Einsatz	lokale Variablen, Parameter	dynamische Datenstrukturen

Tabelle 1: Vergleich Stack vs. Heap

2.4.6 Wann wird was wo gespeichert?

- **Lokal deklarierte primitive Variablen** -> Stack
- **Funktionsparameter**-> Stack
- **Rückgabewerte per Referenz/Pointer** -> abhängig von Aufrufer
- **Mit `new` oder `std::make_unique` erzeugte Objekte** -> Heap
- **Elemente in `std::vector`, `std::string`** -> intern im Heap

3 OBJEKTORIENTIERTES PARADIGMA

1. Objekte kommunizieren durch das Senden und Empfangen von Nachrichten (welche aus Objekten bestehen)
2. Objekte haben ihren eigenen Speicher
3. Jedes Objekt ist die Instanz einer Klasse
4. Die Klasse beinhaltet das Verhalten aller ihrer Instanzen

Auszug aus der Definition nach Alan Kay, 1993

3.1 KLASSEN, OBJEKTE UND MEMBERFUNKTIONEN

Eine **Klasse** in C++ ist ein benutzerdefinierter Typ, der Daten (Membervariablen) und Funktionen (Memberfunktionen) zusammenfasst. Aus einer Klasse lassen sich Objekte (auch Instanzen genannt) erzeugen, die auf diesem Bauplan basieren.

Definition einer Klasse

Klassen bestehen aus einer Kopfzeile mit dem Schlüsselwort `class` und dem Klassennamen, einem Rumpf in geschweiften Klammern und einem abschließenden Semikolon. Die Daten und Funktionen werden innerhalb der Klasse deklariert:

```
class Example {
    int x;
    void do_something();
};
```

Objekterzeugung und Lebenszyklus

Ein Objekt ist eine konkrete Instanz einer Klasse. Es wird erzeugt, verwendet und schließlich automatisch oder manuell zerstört. Die Verwaltung dieser Phasen erfolgt durch spezielle Funktionen:

- **Konstruktor:** gleichnamige Funktion zur Initialisierung
- **Destruktor:** Funktion zur Aufräumlogik, beginnt mit ~

```
class Simple {
public:
    Simple();        // Konstruktor
    ~Simple();       // Destruktor
};
```

Der Konstruktor wird beim Anlegen des Objekts automatisch aufgerufen, der Destruktor beim Verlassen des Gültigkeitsbereichs (z. B. bei Stack-Objekten).

Memberfunktionen

Funktionen innerhalb einer Klasse heißen **Memberfunktionen**. Sie können auf die Daten des Objekts zugreifen und typischerweise über das Punkt-Operator (`.`) verwendet werden.

```
class Counter {
    int value;
public:
    void reset() { value = 0; }
    void increment() { ++value; }
    int get() const { return value; }
};
```

Der Aufruf erfolgt z.B. so:

```
Counter c;
c.increment();
std::cout << c.get();
```

Beispiel: Eigener Vektor-Typ

Das folgende Beispiel zeigt eine einfache Klasse mit Konstruktor, Zugriffsfunktion und Datenhaltung über dynamisch allozierten Speicher:

```
class Vector {
public:
    Vector(int s) : elem{new double[s]}, sz{s} {}
    double& operator[](int i) { return elem[i]; }
    int size() const { return sz; }

private:
    double* elem;
    int sz;
};
```

Der Speicher für die Elemente wird mit new zur Laufzeit auf dem freien Speicher (Heap) angelegt. Die Daten selbst sind `private`, der Zugriff erfolgt kontrolliert über Memberfunktionen.

Hinweise zur Gestaltung

- Konstruktoren dürfen überladen werden.
- Der Destruktor hat keine Parameter und wird nicht manuell aufgerufen.
- Datenzugriff sollte möglichst über Funktionen erfolgen (siehe nächster Abschnitt zur Kapselung).

3.2 KAPSELUNG, VERERBUNG UND POLYMORPHISMUS

Kapselung

Kapselung bedeutet das Zusammenfassen von Daten und Funktionen innerhalb eines Typs, sodass interne Implementierungsdetails verborgen bleiben. Der Zugriff erfolgt ausschließlich über eine klar definierte Schnittstelle (*Interface*). Dies schützt den internen Zustand und reduziert Abhängigkeiten.

- Nur definierte Memberfunktionen können auf `private` Daten zugreifen.
- Änderungen an der Implementierung erfordern keine Änderungen bei Nutzern der Klasse.

Dieses Prinzip ist zentral für **abstrakte Datentypen (ADT)** – ein Konzept, bei dem nur die Signaturen der Funktionen sichtbar sind, nicht aber deren interne Arbeitsweise.

Vererbung

Vererbung erlaubt es, eine neue Klasse (Subklasse) auf Basis einer bestehenden Klasse (Basisklasse) zu definieren. Gemeinsame Funktionalität wird wiederverwendet, zusätzliche spezialisierte Eigenschaften können ergänzt werden.

```
class Animal {
public:
    void eat();
};

class Dog : public Animal {
public:
    void bark();
};
```

Ein `Dog` erbt alle `public` und `protected` Member von `Animal` und kann sie direkt verwenden.

Polymorphismus

Polymorphismus erlaubt es, Objekte unterschiedlicher Klassen über einen gemeinsamen Basistyp anzusprechen – meist über `virtual` Funktionen.

```
class Shape {
public:
    virtual void draw();
};

class Circle : public Shape {
public:
    void draw() override;
};

void render(Shape& s) {
    s.draw();  // Aufruf haengt vom konkreten Objekttyp ab
}
```

Der tatsächliche Funktionsaufruf wird zur Laufzeit entschieden (*dynamisches Binden*). Voraussetzung ist ein Zeiger oder eine Referenz auf den Basistyp sowie eine `virtual`-Deklaration.

class vs. struct

Sowohl `class` als auch `struct` definieren benutzerdefinierte Typen mit identischer Funktionalität. Der Unterschied liegt in der **Zugriffsregel**:

- `class`: Mitglieder sind `private` standardmäßig
- `struct`: Mitglieder sind `public` standardmäßig

Zugriffsmodifikatoren

- **public**
  Mitglieder sind von überall sichtbar - auch von außerhalb der Klasse. Dies bildet die öffentliche Schnittstelle (Interface) eines Typs.
- **private**
  Zugriff ist nur innerhalb der Klasse selbst erlaubt. Ideal für interne Zustände, die nicht direkt verändert werden sollen.
- **protected**
  Wie `private`, jedoch mit dem Unterschied, dass abgeleitete Klassen (`class Derived : public Base`) auf diese Member zugreifen dürfen.

3.3 FUNKTIONS- UND OPERATORÜBERLADUNG

Funktionsüberladung

In C++ können mehrere Funktionen denselben Namen haben, solange sie sich in der Anzahl oder im Typ der Parameter unterscheiden. Dieses Konzept heißt **Funktionsüberladung**.

```
void print(int i);
void print(double d);
void print(const std::string& s);

print(42);          // ruft print(int) auf
print(3.14);        // ruft print(double) auf
print("Hello");     // ruft print(std::string) auf
```

Die Auswahl erfolgt zur Compile-Zeit anhand der Argumente. Überladene Funktionen sollten semantisch dasselbe leisten, aber für verschiedene Typen.
Ist keine eindeutige Auswahl möglich, führt dies zu einem `ambiguous call`:

```
void f(int, double);
void f(double, int);

f(0, 0);  // Fehler: Mehrdeutig
```

Auch Memberfunktionen einer Klasse können überladen werden.

Operatorüberladung

C++ erlaubt es, **Operatoren** wie `+`, `<`, `[]`, `()` etc. für eigene Typen zu überladen, sodass diese wie eingebaute Typen verwendet werden können.

```
class Vector {
public:
    double& operator[](int i) { return elem[i]; }
    Vector operator+(const Vector& other);
private:
    double* elem;
    int sz;
};
```

Durch Überladen von `operator[]` kann ein Objekt wie ein Array verwendet werden. Die Addition zweier Objekte wird mit `operator+` definiert.

Operatorfunktionen können als Member oder als freie Funktionen geschrieben werden. Für symmetrische Operatoren wie `==` oder `+` empfiehlt sich häufig die freie Variante mit Freundschaft:

```
class Point {
    int x, y;
public:
    friend Point operator+(const Point& a, const Point& b);
};
```

4 FUNKTIONALES PARADIGMA

4.1 FUNKTIONALE PROGRAMMIERUNG IN C++

Funktionale Programmierung ist ein Programmierstil, bei dem Funktionen als zentrale Bausteine betrachtet werden. In C++ sind funktionale Konzepte durch moderne Sprachfeatures wie Lambdas, Funktionsobjekte und Templates nutzbar.

Lambda-Ausdrücke

Ein **Lambda-Ausdruck** ist eine anonyme Funktion, die direkt im Code definiert wird. Die Syntax ist kompakt und ermöglicht Funktionen „on the fly“:

```
auto greater_than = [](int a, int b) { return a > b; };
std::cout << greater_than(4, 2); // Ausgabe: 1
```

Lambdas sind besonders nützlich für lokale Operationen, z.B. als Parameter für `std::sort`, `std::for_each` oder benutzerdefinierte Funktionen.

Closures und Capture Lists

Lambdas können auf Variablen aus ihrem Kontext zugreifen. Dies erfolgt über die **Capture List** in eckigen Klammern:

```
int threshold = 5;
auto is_small = [threshold](int x) { return x < threshold; };
```

- `[=]` -> alle benötigten Variablen per Wert kopieren
- `[&]` -> alle per Referenz
- `[x]` -> nur x per Wert
- `[&x]` -> nur x per Referenz

Ein Lambda mit erfassten Werten ist eine sogenannte **Closure** – ein Funktionsobjekt mit internem Zustand.

Direkter Lambda-Aufruf

Lambdas können direkt definiert und sofort aufgerufen werden - nützlich für einmalige Berechnungen:

```
int result = [](int x, int y) { return x + y; }(3, 4);
```

Generische Lambdas

Seit C++14 kann der Parametertyp eines Lambdas mit `auto` generisch gehalten werden:

```
auto print = [](const auto& x) { std::cout << x << '\n'; };
print("Hello"); print(42);
```

Higher-Order Functions

Eine **Higher-Order Function** ist eine Funktion, die andere Funktionen als Argument nimmt oder zurückgibt.

```
void for_all(const std::vector<int>& v, auto op) {
    for (auto x : v) op(x);
}
for_all({1, 2, 3}, [](int x) { std::cout << x << '\n'; });
```

Nested Functions und Closures im Kontext

C++ erlaubt keine echten verschachtelten Funktionen wie z.B. Python. Aber durch Lambdas mit Captures lässt sich ein ähnliches Verhalten nachbilden:

```
void outer() {
    int counter = 0;
    auto inner = [&]() { ++counter; };
    inner(); inner();
    std::cout << counter; // Ausgabe: 2
}
```

Currying (teilweise Anwendung)

Currying bedeutet, eine Funktion mit mehreren Argumenten in eine Kette von Funktionen mit jeweils einem Argument umzuwandeln:

```
auto add = [](int a) {
    return [=](int b) { return a + b; };
};
auto add5 = add(5);
std::cout << add5(3); // Ausgabe: 8
```

C++ unterstützt dieses Muster durch Lambdas elegant, obwohl es aus der funktionalen Programmierung stammt.

5 GENERISCHE PROGRAMMIERUNG

Generische Programmierung ermöglicht es, Algorithmen und Datentypen unabhängig von konkreten Typen zu formulieren. In C++ wird dies primär durch `template`-Mechanismen realisiert. Der Compiler generiert daraus zur Compile-Zeit typenspezifischen Code (Instanziierung), wodurch generische Programme effizient und typsicher bleiben. Generische Programmierung fördert Wiederverwendbarkeit, Abstraktion und Typsicherheit – zentrale Prinzipien moderner Softwareentwicklung.

5.1 TEMPLATES

Generische Programmierung ermöglicht es, Funktionen und Datentypen so zu definieren, dass sie mit beliebigen Typen arbeiten können. In C++ wird dieses Paradigma primär durch **Templates** realisiert – sowohl für Typen als auch für Werte.

Templates für Typen

Ein `template` erlaubt es, einen Platzhalter für einen Typ zu definieren:

```
template<typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

Dieser Code erzeugt eine Funktion `max`, die mit beliebigen Typen funktioniert, solange der Operator `>` verfügbar ist. Der Compiler generiert bei Bedarf eine passende Version – dieser Vorgang heißt *Template Instanziierung*.

Template-Klassen

Auch ganze Klassen können generisch gestaltet werden. Im Beispiel wird ein einfacher Vektor-Typ für beliebige Elementtypen definiert:

```
template<typename T>
class Vector {
    T* elem;
    int sz;
public:
    explicit Vector(int s);
    ~Vector() { delete[] elem; }

    T& operator[](int i) { return elem[i]; }
    const T& operator[](int i) const { return elem[i]; }

    int size() const { return sz; }
};
```

Eine mögliche Konstruktor-Definition außerhalb der Klasse:

```
template<typename T>
Vector<T>::Vector(int s) {
    if (s < 0) throw std::length_error("Negative size");
    elem = new T[s];
    sz = s;
}
```

Verwendung:

```
Vector<int> vi(10);        // Vector von int
Vector<std::string> vs(5); // Vector von Strings
```

Constrained Templates (C++20)

Templates können auf bestimmte Typ-Eigenschaften eingeschränkt werden. Diese Einschränkungen nennt man **Constraints**, und sie werden mit Hilfe von `Concepts` formuliert:

```
template<typename T>
concept Element = requires(T a) {
    a == a;
    a < a;
};

template<Element T>
class Vector { /* ... */ };
```

So wird zur Compile-Zeit sichergestellt, dass nur gültige Typen verwendet werden.

Wert-Templates (non-type arguments)

Templates können auch **Wertparameter** aufnehmen – z.B. zur Definition von statisch dimensionierten Arrays:

```
template<typename T, int N>
struct Buffer {
    T data[N];
    int size() const { return N; }
};
```

Verwendung:

```
Buffer<char, 128> global_buf;
Buffer<int, 32> local_buf;
```

Template Argument Deduction

Templates können die benötigten Typen oft automatisch aus den Konstruktorargumenten ableiten:

```
template<typename T>
class Pair {
    T a, b;
public:
    Pair(T x, T y) : a(x), b(y) {}
};

Pair p1{3, 4};  // deduziert T = int
```

Bei uneindeutigen Typen hilft eine **Deduction Guide**, z. B.:

```
template<typename Iter>
Vector2(Iter, Iter) -> Vector2<typename Iter::value_type>;
```

Funktions-Templates

Funktionen können wie Klassen generisch gestaltet werden:

```
template<typename Sequence, typename Value>
Value sum(const Sequence& s, Value init) {
    for (auto x : s)
        init += x;
```

```
    return init;
}
```

Verwendung:

```
std::vector<int> vi = {1, 2, 3};
double result = sum(vi, 0.0);  // ergibt 6.0
```

Funktionsobjekte (Funktoren)

Ein **Funktionsobjekt** ist ein Objekt mit einer `operator()`-Definition. Es verhält sich wie eine Funktion, kann aber internen Zustand enthalten:

```
template<typename T>
class Less_than {
    const T val;
public:
    Less_than(const T& v) : val(v) {}
    bool operator()(const T& x) const { return x < val; }
};
```

Verwendung mit einem Algorithmus:

```
Less_than<int> less42(42);
bool b = less42(40); // true

std::count_if(v.begin(), v.end(), Less_than<int>(10));
```

Funktionsobjekte eignen sich besonders als Parameter für generische Algorithmen, da sie effizienter und flexibler als einfache Funktionszeiger sind.

Vergleich mit Lambdas

Lambdas und Funktionsobjekte erfüllen denselben Zweck – sie erzeugen „Callable"-Objekte. Lambdas sind jedoch anonymer und kürzer:

```
std::count_if(v.begin(), v.end(), [](int x) { return x < 10; });
```

Funktionsobjekte lohnen sich bei komplexem Zustand oder Wiederverwendung. Lambdas sind ideal für Inline-Verwendung.

5.2 RULE OF THREE, FIVE UND ZERO

In C++ steuert der Umgang mit Ressourcen (z.B. dynamischer Speicher, Datei-Handles, Zeiger) maßgeblich die Notwendigkeit zur Definition spezieller Funktionen in einer Klasse. Die Regeln „of Three", „of Five" und „of Zero" helfen dabei, Ressourcen korrekt zu verwalten – gemäß dem Prinzip: *„Wer Ressourcen besitzt, muss sich darum kümmern."*

Was ist die Rule of Three?

Wenn eine Klasse eine der folgenden drei Memberfunktionen definiert, dann sollte sie auch die beiden anderen definieren:

- **Destruktor** (`~Class()`)
- **Copy Constructor** (`Class(const Class&)`)
- **Copy Assignment Operator** (`operator=(const Class&)`)

Diese Regel gilt, wenn eine Klasse Ressourcen besitzt, die tief kopiert oder gelöscht werden müssen (z.B. mit new allozierter Speicher).

```
class Buffer {
    int* data;
    int size;

public:
    Buffer(int s) : data{new int[s]}, size{s} {}
    ~Buffer() { delete[] data; }

    Buffer(const Buffer& other)                // Copy Constructor
      : data{new int[other.size]}, size{other.size} {
        std::copy(other.data, other.data + size, data);
    }

    Buffer& operator=(const Buffer& other) {   // Copy Assignment
        if (this != &other) {
            delete[] data;
            data = new int[other.size];
            size = other.size;
            std::copy(other.data, other.data + size, data);
        }
        return *this;
    }
};
```

Was ist die Rule of Five?

- **Move Constructor** (`Class(Class&&)`)
- **Move Assignment Operator** (`operator=(Class&&)`)

Wenn eine Klasse Kopieroperationen und Destruktor braucht, sollte sie auch überlegen, die Move-Varianten zu definieren – insbesondere, wenn Ressourcen effizient verschiebbar sind.

```
Buffer(Buffer&& other) noexcept              // Move Constructor
  : data{other.data}, size{other.size} {
    other.data = nullptr;  // Besitzuebertragung
    other.size = 0;
}

Buffer& operator=(Buffer&& other) noexcept { // Move Assignment
    if (this != &other) {
        delete[] data;
        data = other.data;
        size = other.size;
        other.data = nullptr;
        other.size = 0;
    }
    return *this;
}
```

→ Die **Rule of Five** gilt, wenn Ressourcen manuell verwaltet werden und die Klasse eine effiziente Übergabe ermöglichen soll (z.B. Rückgabe aus Funktionen).

Was ist die Rule of Zero?

Ideal ist, wenn eine Klasse gar keine dieser Funktionen selbst definieren muss – das nennt man **Rule of Zero**.
Sie gilt, wenn die Klasse ausschließlich aus anderen Klassen besteht, die sich selbst korrekt verwalten (z.B. `std::vector`, `std::string`, `unique_ptr`).

```
class DataHolder {
    std::vector<int> data;
    std::string name;
    // keine eigenen Spezialfunktionen notwendig
};
```

→ Die Rule of Zero ist **robust, wartbar und sicher**, weil sie auf RAII basiert und keine manuelle Speicherverwaltung enthält.

Wann sollte welche Regel beachtet werden?

- Verwende **Rule of Zero**, wenn du nur Standardtypen oder moderne Smart-Pointer nutzt.
- Wende die **Rule of Three** an, wenn deine Klasse selbst Ressourcen verwaltet (`new`, `malloc`, Dateihandles).
- Nutze die **Rule of Five**, wenn du zusätzlich Performance durch **Move-Semantik** optimieren willst.

Warum ist das wichtig?

Ohne diese Regeln kann es leicht zu **Ressourcenlecks**, **doppeltem Freigeben**, **undefiniertem Verhalten** oder ineffizientem Code kommen. Insbesondere bei dynamischer Speicherverwaltung ist sauberes Kopieren und Löschen unerlässlich.

5.3 RULE OF ZERO, THREE UND FIVE

Warum gibt es diese Regeln überhaupt?

In C++ werden bestimmte Funktionen von Klassen automatisch generiert, wenn sie nicht explizit angegeben sind. Diese sogenannten *Special Member Functions* betreffen das Kopieren, Verschieben und Zerstören von Objekten.
Sie sind zentral für die **Ressourcenverwaltung** (z.B. Speicher, Dateien, Handles) und müssen korrekt implementiert sein, damit Objekte sicher und effizient verwendet werden können.

Die fünf Special Member Functions

Eine Klasse kann automatisch (oder manuell) die folgenden Memberfunktionen besitzen:

1. Destruktor: `~T()`
2. Copy Constructor: `T(const T&)`
3. Copy Assignment Operator: `operator=(const T&)`
4. Move Constructor: `T(T&&)`
5. Move Assignment Operator: `operator=(T&&)`

Die Regeln **of Zero, Three und Five** beschreiben, welche dieser Funktionen definiert werden sollen – abhängig davon, wie die Klasse mit Ressourcen umgeht.

Rule of Zero

WAS IST DAS? Eine Klasse benötigt keine eigene Definition der fünf Memberfunktionen – alles wird automatisch korrekt generiert.

WANN GILT DAS? Wenn eine Klasse nur aus anderen gut verwalteten Objekten besteht (`std::vector`, `std::string`, `unique_ptr` etc.) und selbst keine Ressourcen (z.B. über new) verwaltet.

```
class Person {
    std::string name;
    int age;
    std::vector<std::string> hobbies;
    // keine eigene Ressourcenkontrolle notwendig
};
```

Fehlerhaftes Beispiel (ohne Rule of 3/5):

```
class Broken {
    int* data; // selbst verwalteter Speicher
    // kein Destruktor, kein Copy/Move -> Speicherleck oder Double-Free
};
```

Rule of Three

RICHTIGES BEISPIEL:

WAS IST DAS? Wenn du eine dieser drei Funktionen brauchst, solltest du alle drei implementieren:

- Destruktor
- Copy Constructor
- Copy Assignment Operator

WARUM? Weil sie alle mit **Kopieren und Freigeben von Ressourcen** zu tun haben. Wenn du z.B. Speicher per new anforderst, musst du sicherstellen, dass beim Kopieren und Zerstören alles korrekt behandelt wird.

Richtiges Beispiel:

```
class Buffer {
    int* data;
    int size;
public:
    Buffer(int s) : data{new int[s]}, size{s} {}
    ~Buffer() { delete[] data; }

    Buffer(const Buffer& other)
      : data{new int[other.size]}, size{other.size} {
        std::copy(other.data, other.data + size, data);
    }

    Buffer& operator=(const Buffer& other) {
        if (this != &other) {
            delete[] data;
            data = new int[other.size];
            size = other.size;
            std::copy(other.data, other.data + size, data);
        }
        return *this;
    }
};
```

Fehlerhaftes Beispiel (fehlende Kopierlogik):

```
class Buggy {
    int* data;
public:
    Buggy(int s) { data = new int[s]; }
    ~Buggy() { delete[] data; }
    // Kein Copy Constructor -> flache Kopie
    // -> Doppeltes delete bei Zerstoerung
};
```

Rule of Five

WAS IST DAS? Zusätzlich zur Rule of Three solltest du ab C++11 auch diese beiden Funktionen bereitstellen, wenn deine Klasse Ressourcen besitzt:

- Move Constructor
- Move Assignment Operator

WARUM? Um das unnötige Kopieren großer Datenmengen zu vermeiden – statt zu kopieren, kannst du den Besitz des Speichers übertragen.

Richtiges Beispiel mit Move-Semantik:

```
Buffer(Buffer&& other) noexcept
  : data{other.data}, size{other.size} {
    other.data = nullptr; // ownership uebertragen
}

Buffer& operator=(Buffer&& other) noexcept {
    if (this != &other) {
        delete[] data;
        data = other.data;
        size = other.size;
        other.data = nullptr;
    }
    return *this;
}
```

FEHLERFALL OHNE MOVE: Wenn eine Klasse mit großen Ressourcen (z.B. `std::vector` oder dynamischem Speicher) keine Move-Operationen unterstützt, wird beim Rückgabewert einer Funktion unnötig kopiert – mit hohem Zeitaufwand.

Was bedeuten diese Regeln wirklich?

Die Begriffe *Rule of Zero, Three, Five* sind keine „Regeln" im rechtlichen Sinn, sondern Erfahrungswerte aus der Praxis von Systemprogrammierung mit C++. Sie sind ein Hinweis darauf, **wann man bestimmte Memberfunktionen selbst schreiben sollte** – und vor allem, **wann man es besser bleiben lässt**.
Die zugrundeliegende Idee ist einfach:

> *Wenn du in einer Klasse manuell in den Lebenszyklus von Objekten eingreifst – also Konstruktion, Zerstörung, Kopieren oder Verschieben selbst steuerst –, dann musst du dir über **alle fünf** dieser Operationen Gedanken machen.*

Denn: Wenn du eine davon definierst (z.B. einen Destruktor), schaltet der Compiler das automatische Erzeugen der anderen möglicherweise ab. Es liegt dann an dir, die übrigen korrekt bereitzustellen – oder bewusst zu verbieten.

Man könnte es auch so formulieren (frei nach einem flapsigen, aber treffenden Kommentar eines Kommilitonen):

> *„Wenn deine Klasse schon komplex genug ist, um einen eigenen Konstruktor oder Destruktor zu brauchen, dann ist sie auch komplex genug, um **alle fünf Spezialfunktionen** zu brauchen. Sonst benutzt später jemand deinen Code und erwartet, dass alles funktioniert.“*

Der Sinn der Regeln ist also nicht, dogmatisch alle fünf Funktionen zu schreiben – sondern zu erkennen: **Wenn du einen Eingriff vornimmst, trägst du Verantwortung für den ganzen Lebenszyklus.** Wer das ignoriert, riskiert undefiniertes Verhalten, Speicherlecks, doppelte Freigaben oder ineffiziente Kopien – oft stillschweigend und schwer auffindbar.
Darum helfen die Regeln dabei, Klassen wartbar, korrekt und effizient zu halten – **je weniger du selbst regeln musst, desto besser.**

5.3.1 Aggregation

Aggregation ist eine Form der Objektbeziehung, bei der ein Objekt ein anderes enthält – jedoch *ohne Besitz* daran zu übernehmen. Die Lebensdauer des enthaltenen Objekts liegt dabei nicht in der Verantwortung der aggregierenden Klasse.
Im folgenden Beispiel besitzt ein `Department` einen Verweis (`const Teacher&`), aber es ist nicht für dessen Lebensdauer verantwortlich:

```
class Teacher {
    std::string m_name{};
public:
    Teacher(std::string_view name) : m_name{ name } {}
    const std::string& getName() const { return m_name; }
};

class Department {
    const Teacher& m_teacher; // Aggregation: kein Besitz
public:
    Department(const Teacher& teacher) : m_teacher{ teacher } {}
};

int main() {
    Teacher bob{ "Bob" };        // bob lebt ausserhalb der Abteilung
    {
        Department department{ bob };  // benutzt bob, besitzt ihn aber nicht
    } // department wird geloescht, bob bleibt bestehen

    std::cout << bob.getName() << " still exists!\n"; // Gueltig
    return 0;
}
```

Die Klasse `Department` speichert eine Referenz auf ein `Teacher`-Objekt, das außerhalb ihrer selbst erzeugt wurde. Dadurch ist sie nicht für dessen Existenz verantwortlich – sie nutzt es nur.
Alltagssprache: *Eine Aggregation ist wie ein Klassenzimmer, das einen Lehrer nutzt – aber der Lehrer gehört nicht dem Klassenzimmer. Er existiert unabhängig davon.*
Wichtig: Da Referenzen nicht „null“ sein können, muss beim Anlegen der Aggregation sichergestellt werden, dass das referenzierte Objekt tatsächlich existiert – und auch *länger lebt* als die aggregierende Klasse.

Type Traits in C++

Type Traits sind Hilfsmittel der C++-Standardbibliothek, um während der Compile-Zeit Informationen über Typen zu ermitteln oder Bedingungen zu formulieren.

Alltagssprache:
Type Traits sind wie eine Checkliste für Typen: Du fragst den Compiler „Ist dieser Typ ein Integer?“ und bekommst die Antwort direkt zur Compile-Zeit. So kannst du deinen Code anpassen, bevor er überhaupt läuft.
Verwendung finden sie z.B. bei:

- *SFINAE* (Substitution Failure Is Not An Error)
- *template constraints*
- *Type-Dispatching* zur Spezialisierung generischer Templates

Beispiel: `std::is_integral`
Ermittelt zur Compile-Zeit, ob ein Typ ein ganzzahliger Typ ist:

```
#include <type_traits>
#include <iostream>

template<typename T>
void check_integral() {
    if constexpr (std::is_integral_v<T>) {
        std::cout << "Integral type\n";
    } else {
        std::cout << "Not integral\n";
    }
}

int main() {
    check_integral<int>();     // Integral type
    check_integral<float>();   // Not integral
}
```

Beispiel: `std::is_same`
Vergleicht zwei Typen zur Compile-Zeit:

```
static_assert(std::is_same_v<int, int>);        // OK
static_assert(!std::is_same_v<int, double>);    // OK
```

Nützlich z.B. zur Vermeidung fehlerhafter Spezialisierungen.

6 METAPROGRAMMIERUNG

6.1 COMPILE-TIME EVALUATION & METAPROGRAMMING CONSTRUCTS

6.1.1 const, constexpr, consteval

Diese drei Schlüsselwörter steuern, wann und ob ein Ausdruck zur Compile-Zeit ausgewertet wird.

`const`
Deklariert, dass ein Wert nach der Initialisierung nicht mehr verändert werden darf. `const` kann sowohl zur Compile-Zeit als auch zur Laufzeit verwendet werden.

```
const int x = 5;            // fester Wert
const int y = compute();    // erlaubt, auch wenn compute()
                            // nur zur Laufzeit berechenbar ist
```

`const` schützt nur vor Veränderung – garantiert aber keine Compile-Time-Auswertung.

`constexpr`
Markiert eine Variable oder Funktion, deren Wert **möglichst** bereits zur Compile-Zeit berechnet werden soll. Wenn die Umstände es erlauben (z.B. keine Laufzeit-Abhängigkeiten), wird der Ausdruck sofort zur Compile-Zeit ausgewertet.

```
constexpr int square(int x) { return x * x; }

constexpr int value = square(4);      // OK, Compile-Time-Auswertung
int runtime = square(std::rand());    // OK, aber zur Laufzeit
```

`Constexpr`-Funktionen dürfen auch mehr als nur einen Return-Ausdruck enthalten (z.B. Schleifen, Bedingungen).

`consteval`
Erzwingt, dass eine Funktion **ausschließlich** zur Compile-Zeit aufgerufen werden darf. Jeder Aufruf muss vollständig auf statischen Werten basieren.

```
consteval int fixed() { return 42; }

constexpr int ok = fixed();     // OK
int fail = fixed();             // Fehler! Nicht zur Laufzeit erlaubt
```

Im Gegensatz zu `constexpr` ist `consteval` strikt – es erlaubt keine Laufzeitnutzung, selbst wenn keine dynamischen Abhängigkeiten vorhanden sind.

`static_assert`
`static_assert` ist eine Anweisung an den Compiler, eine Bedingung **zur Compile-Zeit** zu prüfen. Falls die Bedingung `false` ergibt, bricht die Kompilierung mit einer Fehlermeldung ab.
Die Syntax ist einfach:

```
static_assert(condition [, "Fehlermeldung"]);
```

Wird oft verwendet in generischen Templates, bei Type Traits oder überall dort, wo bestimmte Annahmen zur Compile-Zeit garantiert werden müssen.

Beispiel: Type Trait Prüfung

```
static_assert(std::is_integral_v<int>);         // OK
static_assert(! std::is_same_v<int, double>);   // OK: int und double sind verschieden
```

Ein typischer Denkfehler wäre es, anzunehmen, dass folgender Ausdruck falsch ist:

```
// Das scheint wie ein Fehler auszusehen ...
static_assert(! std::is_same_v<int, double>);
// ... ist aber korrekt, weil int != double -> Bedingung ist true
```

Zur Abgrenzung: `assert` zur Laufzeit
Während `static_assert` zur Compile-Zeit wirkt, wird die normale `assert()` aus `<cassert>` zur Laufzeit geprüft. Sie wird häufig in Debug-Builds verwendet und kann in Release-Builds deaktiviert sein:

```
#include <cassert>

int main() {
    int x = 5;
    assert(x > 0); // Laufzeitpruefung
}
```

6.1.2 auto und decltype – Typableitung

C++ erlaubt es, den Typ einer Variable automatisch vom Initialisierungswert abzuleiten. Die Schlüsselwörter `auto` und `decltype` bilden dafür die Grundlage.

`auto` leitet den Typ einer Variable zur Compile-Zeit direkt aus dem Initialisierungs-Ausdruck ab:

```
auto b = true;       // bool
auto ch = 'x';       // char
auto i = 123;        // int
auto d = 1.2;        // double
auto z = sqrt(2.0); // z ist double, da sqrt(double) -> double
```

Vorteile von `auto`:

- Spart redundante oder lange Typdeklarationen
- Unterstützt generische Programmierung, bei der Typen nicht explizit bekannt sind
- Erhöht Lesbarkeit in Kombination mit komplexen Ausdrücken (z.B. Iteratoren, Lambdas)
- Reduziert Copy-Paste-Fehler bei mehrfach vorkommenden Typen

Typisches Beispiel:

```
std::vector<std::string> names = { "Alice", "Bob" };

for (auto& name : names) {
    std::cout << name << '\n';
}
```

„Almost always auto“
Die Regel *„Almost always auto“* besagt, dass man `auto` verwenden sollte, *außer* der Typ trägt semantische Bedeutung. Beispiel: `float` vs. `double`, oder `int32_t` in Low-Level-Programmierung.

```
auto f = 3.14f;      // Achtung: f ist float, aber leicht zu übersehen!
double d = 3.14;     // explizit besser, wenn Praezision wichtig ist
```

`auto` ist besonders hilfreich, wenn der Typ schwer auszudrücken oder unwichtig ist – aber nicht in APIs, in denen Klarheit über den Typ Teil der Schnittstelle ist.

Grenzen von `auto`:
Nicht alle Initialisierungen führen zu den erwarteten Typen. Besonders problematisch ist die Initialisierung mit geschweiften Klammern:

```
auto x = {1};      // std::initializer_list<int>!
auto y{1};         // int (Uniform Initialization)
```

`decltype`
`decltype` gibt den Typ eines Ausdrucks zurück, ohne ihn auszuwerten. Damit kann man gezielt Typen ableiten:

```
int a = 5;
decltype(a) b = a * 2; // b ist int

decltype(sqrt(2.0)) root = 1.41; // root ist double
```

`decltype` ist auch nützlich für Rückgabewerte von Funktionen oder um bei Templates konsistent zu bleiben.

Fazit:

- `auto` reduziert Aufwand und erhöht Lesbarkeit bei komplexen Typen.
- `decltype` erlaubt gezielte Typbestimmung ohne Ausführung.
- Verwende `auto` fast immer – aber bewusst.

Alltagssprache:
Mit `auto` sagt man dem Compiler: „Du weißt es besser als ich – lies den Typ selbst raus." Mit `decltype` fragt man den Compiler: „Was ist das eigentlich für ein Typ?"

6.1.3 Variadische Templates und Fold Expressions

Variadic Templates erlauben es, eine Funktion oder Klasse mit beliebig vielen Argumenten beliebigen Typs zu definieren. Diese Argumente werden intern als sogenannter *Parameter Pack* behandelt.

Ein einfaches Beispiel:

```
template<typename T, typename... Args>
T sum(const T& first, const Args... args) {
    return first + sum(args...); // rekursives Entpacken
}
```

Hier wird `args...` rekursiv weitergegeben und entpackt, bis nur noch ein einzelnes Argument vorhanden ist.

Verpacken und Entpacken

- `typename... Args` definiert ein Parameter Pack (Verpacken)
- `Args...` entpackt die Argumente (Unpacking)
- `Args&&... args` erlaubt auch das perfekte Weiterleiten (`std::forward`)

Praktisches Beispiel mit Parameter Pack:

```
template<typename... Args>
void debug_log(const Args&... args) {
    (std::cout << ... << args) << '\n'; // Fold Expression
}

debug_log("Result: ", 42, ", valid: ", true);
```

Fold Expressions vereinfachen rekursive Variadic-Aufrufe. Sie ersetzen:

```
print(first);
print(rest...);
```

durch:

```
(std::cout << ... << args); // right fold
```

Right vs. Left Fold

```
template<typename... T>
int sum_right(T... v) {
    return (v + ... + 0); // right fold
}

template<typename... T>
int sum_left(T... v) {
    return (0 + ... + v); // left fold
}
```

Diese Folds sind insbesondere bei numerischer Reduktion nützlich (`sum`, `accumulate`).

Packen → Container:

```
template<typename Res, typename... Ts>
std::vector<Res> to_vector(Ts&&... ts) {
    std::vector<Res> res;
    (res.push_back(ts), ...); // Fold Expression
    return res;
}

auto v = to_vector<int>(1, 2, 3, 4);
```

Nutzung in realistischen Kontexten:

- Argumentweitergabe an Logging, Serialisierung, Transformation
- Implementierung generischer Operatoren, Konstruktoren
- Übergabe an Standard-Algorithmen oder Policies

Vorsicht:

- Variadic Templates können bei Fehlermeldungen schwer zu debuggen sein
- Typfehler in Fold Expressions sind manchmal schwer lesbar
- Template-Instantierungen können bei großen Packs teuer sein

7 DESIGN-PRINZIPIEN & SOFTWAREARCHITEKTUR

7.1 MODULARISIERUNG: TRENNUNG VON DEKLARATION & IMPLEMENTIERUNG

Ein zentrales Prinzip in der Softwareentwicklung ist die **Modularisierung**. Dabei wird ein Programm in klar abgegrenzte, logisch zusammengehörige Teile (*Module*) aufgeteilt. In C++ erfolgt dies klassischerweise durch die **Trennung von Deklaration und Implementierung**. Diese Trennung bildet die Grundlage für wartbare, verständliche und wiederverwendbare Programme.

Vorteile modularer Programme

- **Wiederverwendbarkeit**: Module lassen sich unabhängig in anderen Programmen nutzen.
- **Lesbarkeit & Wartbarkeit**: Durch kleinere Einheiten wird der Code verständlicher.
- **Kompilationszeit**: Nur geänderte Module müssen neu übersetzt werden.
- **Fehlerminimierung**: Klare Schnittstellen reduzieren Wechselwirkungen.
- **Teamarbeit**: Teams können parallel an unterschiedlichen Modulen arbeiten.

Trennung von Schnittstelle und Implementierung

Die Schnittstelle eines Moduls wird in einer `.h` (oder `.hpp`)-Datei deklariert. Die zugehörige Implementierung befindet sich in einer `.cpp`-Datei. Die Header-Datei beschreibt, *was* ein Modul bietet, die `.cpp`-Datei beschreibt, *wie* es funktioniert.

Beispiel: `Vector`-Klasse

```
class Vector {
public:
    Vector(int s);
    double& operator[](int i);
    int size() const;
private:
    double* elem;
    int sz;
};
```

```
#include "Vector.h"

Vector::Vector(int s)
    : elem(new double[s]), sz{s} {}

double& Vector::operator[](int i) {
    return elem[i];
}

int Vector::size() const {
    return sz;
}
```

```cpp
#include "Vector.h"
#include <iostream>

int main() {
    Vector v(3);
    v[0] = 1.1;
    v[1] = 2.2;
    v[2] = 3.3;

    for (int i = 0; i < v.size(); ++i)
        std::cout << v[i] << "\n";
}
```

Separate Compilation

Diese Trennung ermöglicht die **separate Kompilierung** einzelner Codeeinheiten. Jede `.cpp`-Datei kann unabhängig kompiliert werden, solange die zugehörigen Header verfügbar sind. Dies reduziert Kompilationszeiten und verbessert die Fehlerlokalisierung.

Module ab C++20

Seit C++20 kann die Schnittstelle eines Moduls durch das Schlüsselwort `module` definiert und mit `import` eingebunden werden. Dies ersetzt das klassische `#include`-Modell teilweise und verbessert sowohl die **Kompilationszeit** als auch die **Fehlervermeidung durch Duplikate oder Reihenfolgefehler**.

Listing 1: Moduldefinition mit C++20

```cpp
export module Vector;

export class Vector { public: Vector(int s); double& operator[](int i); int size() const; private:
double* elem; int sz; };

Vector::Vector(int s) : elem(new double[s]), sz{s} {}

double& Vector::operator[](int i) { return elem[i]; }

int Vector::size() const { return sz; }

export int size(const Vector& v) { return v.size(); }
```

```cpp
import Vector;
#include <cmath>

double sqrt_sum(Vector& v) {
    double sum = 0;
    for (int i = 0; i < v.size(); ++i)
        sum += std::sqrt(v[i]);
    return sum;
}
```

Vorteile von Modulen

- Module werden nur **einmal** kompiliert.
- `import` ist nicht transitiv: Importierte Symbole bleiben gekapselt.
- Reduziert Fehler durch Mehrfach-Includes oder falsche Reihenfolge.
- Besseres Tooling, bessere Wartbarkeit.

7.2 NAMESPACES & INCLUDE GUARDS

Die Organisation und Strukturierung von Code in größeren Projekten erfordert Mechanismen zur Vermeidung von Namenskonflikten und doppelten Definitionen. Zwei wichtige Konzepte in C++ sind hierbei **Namespaces** und **Include Guards**.

Namespaces

In C++ können gleichnamige Funktionen, Klassen oder Variablen in unterschiedlichen Namensräumen (*namespaces*) koexistieren. Dadurch lassen sich Konflikte vermeiden, z.B. wenn mehrere Bibliotheken gleiche Funktionsnamen verwenden.

```
namespace Math {
    double sqrt(double x) {
        // eigene Implementierung
        return x / 2.0; // Dummy
    }
}
```

Zugriff auf Elemente eines Namensraums:

- Vollständiger Zugriff: `Math::sqrt(4.0);`
- Importieren eines Namensraums: `using namespace Math;` (Achtung: Gefahr von Namenskonflikten!)
- Selektiver Import: `using Math::sqrt;`

Nested Namespaces

```
namespace graphics::render::shaders {
    void compile();
}
```

Include Guards

Beim Verwenden von Header-Dateien besteht die Gefahr, dass eine Datei mehrfach eingebunden wird. Dies kann zu doppelten Definitionen und Compilerfehlern führen. **Include Guards** verhindern dies durch bedingte Präprozessoranweisungen:

```
#ifndef VECTOR_H
#define VECTOR_H

class Vector {
  // ...
};

#endif // VECTOR_H
```

Alternative: `#pragma once`
Eine modernere, kompaktere Alternative ist:

```
#pragma once

class Vector {
  // ...
};
```

`#pragma once` ist einfacher zu schreiben und schwerer falsch zu verwenden, wird aber nicht in allen Compilern garantiert unterstützt. In der Praxis ist es jedoch weit verbreitet und zuverlässig.

7.3 RAII – RESOURCE ACQUISITION IS INITIALIZATION

Was ist RAII?

RAII steht für *Resource Acquisition Is Initialization*. Es ist ein zentrales Idiom in C++, das bedeutet: Eine Ressource (z. B. Speicher, Datei, Netzwerkverbindung, Mutex, Lock, ...) wird beim Erzeugen eines Objekts (im Konstruktor) erworben – und beim Zerstören des Objekts (im Destruktor) automatisch wieder freigegeben – **Kernprinzip:** *Objektlebensdauer = Ressourcenlebensdauer*

Wofür brauche ich RAII?

RAII garantiert, dass Ressourcen:

- korrekt freigegeben werden (`delete`, `close()`, `unlock()`, ...)
- auch im Fehlerfall oder bei `throw` automatisch freigegeben werden
- keine separaten Aufräumroutinen benötigen (kein explizites `try/finally`)

Das verhindert Leaks, Inkonsistenzen und vereinfacht Exception Handling erheblich.

Warum ist das besser als in C?

In C muss jede Ressource manuell verwaltet werden:

- `malloc()` → `free()`
- `fopen()` → `fclose()`
- Fehleranfälligkeit durch „vergessenes“ Freigeben (Memory Leak, File Handle Leak)

In C++ mit RAII übernimmt der Destruktor diese Aufgabe automatisch. Das macht den Code sicherer und robuster – besonders bei komplexen Kontrollflüssen.

Was wäre ohne RAII?

- Man müsste überall selbst an `delete` denken
- Fehler bei Exceptions führen leicht zu nicht freigegebenem Speicher
- Ressourcenverwaltung müsste dupliziert und getestet werden

Was ist besonders an RAII im Vergleich zu anderen Sprachen?

RAII ist direkt im Objektmodell von C++ verankert. Anders als in z.B. Java oder Python gibt es keine Garbage Collection – stattdessen sorgt der Destruktor für deterministische Freigabe.
Vorteile:

- keine Laufzeitkosten durch GC
- Vorhersehbarkeit (Ressourcen werden immer beim Scope-Ende freigegeben)
- effizient und sprachlich integriert (z.B. durch `std::unique_ptr`, `std::lock_guard`)

Wann sollte man RAII anwenden?

- Immer wenn eine Ressource **explizit** erworben und **verlässlich** freigegeben werden muss
- Typische Beispiele:
  - Speicher: `std::unique_ptr`, `std::vector`
  - Dateien: `std::ifstream`, `std::ofstream`
  - Locks: `std::lock_guard`, `std::scoped_lock`
- Auch bei eigenen Klassen, z. B. für Netzwerkverbindungen, Datenbank-Handles usw.

Alltagsvergleich (vereinfacht)

Stell dir RAII wie ein Leihbuch aus der Bibliothek vor:

- Du nimmst das Buch (Konstruktor)
- Wenn du den Raum verlässt (Scope-Ende), wird das Buch automatisch zurückgegeben (Destruktor)
- Du brauchst dich nicht aktiv darum kümmern, es passiert automatisch – auch wenn du f rüher gehen musst (z.B. bei einer `return`-Anweisung oder Exception)

Beispiel:

```
// Datei automatisch geöffnet und geschlossen
void readConfig() {
    std::ifstream file("config.txt"); // öffnet Datei
    std::string line;
    while (std::getline(file, line)) {
        std::cout << line << '\n';
    } // Datei wird hier automatisch geschlossen
}
```

```
// Locking mit RAII (keine manuelle Freigabe nötig)
std::mutex mtx;

void access() {
    std::lock_guard<std::mutex> guard(mtx); // sperrt Mutex
    // Zugriff auf geschützte Daten
} // Mutex wird beim Verlassen des Scopes automatisch freigegeben
```

Zusammenfassung

RAII ist eine der grundlegendsten Techniken im modernen C++. Es sorgt für:

- **Korrekte und sichere Ressourcenfreigabe**
- **Klare Objektverantwortung (Ownership)**
- **Einfacheres Fehler- und Exception-Handling**
- **Robusteren, wartbareren Code**

Ohne RAII wäre C++ deutlich fehleranfälliger – wie C. Mit RAII ist der C++-Entwickler entlastet und der Code strukturiert und sicher.

7.3.1 RAII und seine Verbindung zur Rule of Zero, Three und Five

RAII (*Resource Acquisition Is Initialization*) bildet das Rückgrat eines sicheren und automatisierten Ressourcenmanagements in C++. Die zentrale Idee: Ressourcen wie Speicher, Dateihandles oder Locks werden mit der Lebensdauer eines Objekts verknüpft – sie werden im Konstruktor erworben und im Destruktor freigegeben. Doch ob RAII *automatisch* und fehlerfrei wirkt, hängt stark vom gewählten Umgang mit Ressourcen ab. Genau hier kommen die sogenannten **Regeln der Objektsemantik** ins Spiel: die *Rule of Zero*, *Three* und *Five*.

RULE OF ZERO besagt, dass Klassen, die keine eigenen Ressourcen verwalten, auch keine benutzerdefinierten Konstruktoren, Destruktoren oder Zuweisungsoperatoren benötigen. Alle relevanten Funktionen können vom Compiler generiert werden. Dies ist der **ideale Zustand für RAII**, denn der Sprachmechanismus übernimmt das vollständige Ressourcenmanagement.

RULE OF THREE/FIVE greift, sobald eine Klasse eigene Ressourcen (z.B. rohen Speicher oder Dateideskriptoren) verwaltet und damit die Standardsemantik durch eigene Funktionen überschreibt. In solchen Fällen muss der Entwickler alle relevanten Spezialfunktionen selbst definieren: mindestens Kopierkonstruktor, Kopierzuweisung und Destruktor (Rule of Three), eventuell zusätzlich Move-Konstruktor und Move-Zuweisung (Rule of Five). **RAII ist hier weiterhin anwendbar**, jedoch ist die Verantwortung für korrektes Verhalten vollständig auf den Entwickler übergegangen. Fehler in der Implementierung führen häufig zu Speicherlecks, doppeltem Löschen oder undefiniertem Verhalten.
Zusammengefasst: RAII entfaltet sein volles Potenzial in Kombination mit der Rule of Zero. Sobald jedoch manuelle Kontrolle über Ressourcen notwendig wird, muss RAII bewusst und korrekt durch vollständige Implementierung der Objektsemantik unterstützt werden.

7.4 DAS PIMPL-IDIOM (POINTER TO IMPLEMENTATION)

Das **Pimpl-Idiom** (kurz für *Pointer to Implementation*) ist ein Strukturierungsprinzip in C++, mit dem Implementierungsdetails von Klassen vollständig verborgen werden. Ziel ist es, die **Kopplung zu reduzieren**, **Kompilationszeiten zu verbessern** und eine **klare Trennung von Schnittstelle und Implementierung** zu schaffen – insbesondere bei großen Codebasen und Bibliotheken.

Motivation

Bei der Verwendung von komplexen Datentypen oder häufig veränderten Membern in Klassen kann es zu Problemen führen, wenn diese direkt im Header sichtbar sind:

- Jede Änderung im Header führt zu **Neukompilation** aller abhängigen Dateien.
- Details der Implementierung werden für Nutzer der Klasse **unbeabsichtigt sichtbar**.

Das Pimpl-Idiom löst diese Probleme, indem die Implementierung in eine *separat deklarierte Struktur* ausgelagert wird, auf die nur ein Pointer im Header verweist.

Struktur und Umsetzung

Header-Datei: Nur die Schnittstelle ist sichtbar. Die Implementierung wird nur als unvollständiger Typ (forward declaration) bekannt gemacht.

```
#pragma once
#include <memory>

class MyClass {
public:
    MyClass();
    ~MyClass(); // Muss explizit definiert werden

    void doSomething();

private:
    struct Impl;                        // Vorwärtsdeklaration
    std::unique_ptr<Impl> pImpl;    // Einziger Member
};
```

Implementierungsdatei: Die tatsächlichen Details sind hier definiert und vollständig gekapselt.

```
#include "MyClass.hpp"
#include <iostream>

// Definition der Implementierung
struct MyClass::Impl {
    void doSomethingImpl() {
        std::cout << "Doing something hidden\n";
    }
};

MyClass::MyClass() : pImpl(std::make_unique<Impl>()) {}

MyClass::~MyClass() = default;

void MyClass::doSomething() {
    pImpl->doSomethingImpl();
}
```

Verwendung:

```
#include "MyClass.hpp"

int main() {
    MyClass obj;
    obj.doSomething(); // Nutzt versteckte Implementierung
}
```

Vorteile des Pimpl-Idioms

- **Minimale Abhängigkeiten im Header**: Nur Forward-Declarations, keine Includes nötig
- **Verbesserte Kompilationszeiten**: Änderungen an der Implementierung erfordern keine Neukompilation anderer Dateien
- **Echte Kapselung**: Alle privaten Details sind vollständig verborgen
- **Stabile ABI**: Binärschnittstellen ändern sich nicht bei interner Anpassung

Nachteile und Einschränkungen

- Zusätzlicher Indirektionsaufwand (Performance overhead durch Pointer)
- Keine Inlining-Möglichkeiten für versteckte Funktionen
- Erhöhter Aufwand bei Value-Semantik (`copy/move`-Semantik muss nachgezogen werden)

Anwendungsgebiete

- Große Bibliotheken und Frameworks
- Bibliotheksentwicklung mit stabiler ABI
- Klassen mit häufig wechselnder interner Struktur

Zusammenfassung

Das Pimpl-Idiom ist ein Werkzeug zur **sauberen Trennung von Schnittstelle und Implementierung**. Es erhöht die Wartbarkeit, schützt vor unnötiger Kopplung und verbessert Kompilierzeiten. In modernen C++-Projekten wird es vor allem dort eingesetzt, wo klare Modulschnittstellen und minimale Build-Abhängigkeiten erforderlich sind.

7.5 PIMPL-IDIOM: POINTER TO IMPLEMENTATION

7.5.1 Ausgangssituation: Direkte Datenhaltung im Header

Ein typisches Klassenlayout in C++ könnte wie folgt aussehen:

```
// Person.hpp
#ifndef PERSON_HPP
#define PERSON_HPP

#include <string>

class Person{
  std::string m_name;
  std::string m_strength;
  std::string m_speed;

public:
  Person(std::string s);
  ~Person(); std::string GetAttributes();
};
#endif
```

```
// Person.cpp
#include "Person.hpp"

Person::Person(std::string s) : m_name{s} {
  m_strength = "n/a";
  m_speed    = "n/a";
}

Person::~Person(){}

std::string Person::GetAttributes(){
  return m_name + ", " + m_strength + " " + m_speed;
}
```

Diese Implementierung ist zunächst korrekt, übersichtlich und funktional. Sie hat jedoch entscheidende Nachteile im Hinblick auf Modularität, Wartbarkeit und Binärkompatibilität (ABI-Kompatibilität).

7.5.2 Probleme bei Veränderung der Klasse

Angenommen, die Klasse soll erweitert werden:

```
private:
int age, height;
```

Diese scheinbar einfache Änderung hat weitreichende Konsequenzen:

- Jede Änderung am `private`-Teil erfordert einen `rebuild` aller Translation Units, die diesen Header inkludieren.
- Das `ABI` (*Application Binary Interface*) ändert sich: Objekte, die gegen eine ältere Version gelinkt sind, funktionieren nicht mehr ohne Neuübersetzung.
- Die Implementierung „leckt" durch das Interface – obwohl `m_name`, `m_speed`, `m_strength` interne Details sind, müssen sie im Header offengelegt werden.

7.5.3 Verbesserung: Einsatz des Pimpl-Idioms

Das Pimpl-Idiom löst diese Probleme durch eine strikte Trennung von Schnittstelle und Implementierung. Die Header-Datei zeigt nur einen Zeiger auf eine undurchsichtige Struktur:

```
// Person.hpp
#ifndef PERSON_HPP
#define PERSON_HPP

#include <string>

class Person{
  struct pImplPerson;
  pImplPerson* m_impl;

public:
  Person(std::string s);
  ~Person();
  std::string GetAttributes();
};

#endif
```

In der `.cpp`-Datei liegt nun die vollständige Implementierung:

```
// Person.cpp
#include "Person.hpp"

struct Person::pImplPerson {
  std::string m_name;
  std::string m_strength;
  std::string m_speed;
};

Person::Person(std::string s) : m_impl{new pImplPerson} {
  m_impl->m_name = s;
  m_impl->m_strength = "n/a";
  m_impl->m_speed = "n/a";
}

Person::~Person(){ delete m_impl; }
```

7.5.4 Vorteile der Pimpl-Technik

- **Stabilität der Binärschnittstelle (ABI)**: Die Klasse `Person` ist nun unabhängig von internen Details. Änderungen an `pImplPerson` erfordern keine Neuübersetzung anderer Module.
- **Bessere Kompilationszeiten**: Änderungen an Membervariablen oder Implementierungen führen nicht mehr zu einer Kaskade von Rebuilds.
- **Geringere Kopplung & stärkere Kapselung**: Der Nutzer der Klasse weiß nichts über die interne Datenstruktur – das Interface bleibt konstant.
- **Verbindung zu Designprinzipien**:
  - **RAII**: Die Ressource (`m_impl`) wird im Konstruktor angefordert und im Destruktor freigegeben – ganz im Sinne von RAII.
  - **Rule of Zero/Three/Five**: Sobald ein Rohzeiger verwendet wird, muss zumindest ein Destruktor geschrieben werden – **Rule of Three** greift. Alternativ: mit `std::unique_ptr` könnte man RAII-konform zur **Rule of Zero** zurückkehren.
  - **Trennung von Deklaration und Implementierung**: Die Pimpl-Technik fördert saubere Modularisierung.
  - **Namespaces und Include Guards**: Bleiben vollständig erhalten und unterstützen den gekapselten Charakter.

7.5.5 Fazit

Das Pimpl-Idiom ist ein bewährtes Mittel zur Verbesserung der Wartbarkeit, Erweiterbarkeit und Binärkompatibilität von C++-Code. Es erfordert einen gewissen Initialaufwand, zahlt sich jedoch besonders bei großen Projekten mit vielen Abhängigkeiten aus – und stellt somit ein echtes Werkzeug für saubere Softwarearchitektur dar.

Smart Pointer statt Rohzeiger: Empfehlung für Rule of Zero

In der zuvor gezeigten Pimpl-Implementierung wurde ein roher Zeiger verwendet:

```
pImplPerson* m_impl;
```

Dadurch ist der Entwickler selbst verantwortlich für das `new`/`delete`-Paar. Damit verletzt man die sogenannte **Rule of Zero** und muss mindestens einen eigenen Destruktor definieren. Falls Copy- oder Move-Operationen benötigt werden, muss zusätzlich auch `operator=` (Copy/Move) und Copy-/Move-Konstruktor definiert werden – man spricht dann von der **Rule of Three** oder sogar **Rule of Five**.

Durch Verwendung eines Smart Pointers wie `std::unique_ptr` lässt sich dieses Problem elegant vermeiden:

```
std::unique_ptr<pImplPerson> m_impl;
```

Damit wird der Destruktor automatisch generiert, die Ressourcenfreigabe sicher via RAII gehandhabt, und die Klasse bleibt **Rule-of-Zero**-konform.
Der Konstruktor wird nun so angepasst:

```
m_impl = std::make_unique<pImplPerson>();
```

Damit überträgt man die Verantwortung für die Speicherverwaltung an den Smart Pointer – und vermeidet nicht nur Speicherlecks, sondern auch unnötige manuelle Implementierung von Spezialfunktionen.

8 PRÜFUNGSFRAGEN

8.1 POINTER

Double-Linked-List

Implementieren Sie eine einfache Double-Linked-List, welche Knoten mit den Argumenten `next`,`prev` und `value` speichert und für die Zeiger zu Nachfolgenden und Vorherigen Knoten auch nur Pointer verwendet. Erstellen Sie dafür auch entsprechende Konstruktoren und einen Destruktor.
Welche Art der Konstruktor-Implementation (Rule of 5, 3 oder 0) wäre hier angebracht und warum?

Allgemeine C++-Konzepte und Sprachmechanismen

1. Erklären Sie den Unterschied zwischen einer Kopier-Konstruktion und einer Move-Konstruktion in C++. Wann wird welche Variante aufgerufen?
2. Was versteht man unter dem Begriff RAII im Kontext von C++? Nennen Sie ein Beispiel.
3. Was ist der Unterschied zwischen Stack- und Heap-Speicher in C++? Welche Auswirkungen hat dies auf das Lebenszeitmodell von Objekten?
4. Erklären Sie den Begriff „undefined behavior“ und nennen Sie zwei typische Ursachen in C++.
5. Was versteht man unter „Template-Instantiation“? Welche Rolle spielt dabei der Compiler?

Zeiger, Speicher und Referenzen

1. Erklären Sie die Pointer-Arithmetik in C++. Was passiert bei `ptr + 1`?
2. Geben Sie an, was bei `delete[]` und `delete` jeweils genau passiert. Wann muss welches verwendet werden?
3. Was ist der Unterschied zwischen einer Referenz und einem Zeiger in C++? Wann sollte welches verwendet werden?
4. Welche typischen Gefahren bestehen beim Einsatz von rohen Zeigern in C++?
5. Warum wurde `nullptr` eingeführt und was ist der Unterschied zu `NULL`?

Objektorientierte Konzepte und Speicherverwaltung

1. Beschreiben Sie die Rule of Three in C++. Welche Funktionen umfasst sie?
2. Was ist die Rule of Five und in welchen Fällen ist sie gegenüber der Rule of Three notwendig?
3. Was bedeutet die Rule of Zero und wann ist sie anwendbar?
4. Welche Aufgabe hat der Destruktor in einer Klasse? Wie verhält er sich bei Vererbung?
5. Wie verwaltet man Speicher korrekt in einer Klasse, wenn ein dynamisches Array als Member verwendet wird?

Templates, Metaprogrammierung und Generik

1. Was ist ein Template in C++? Welche Vorteile bietet es?
2. Erklären Sie den Unterschied zwischen Function Templates und Class Templates.
3. Was ist ein Variadic Template? Nennen Sie einen typischen Anwendungsfall.
4. Warum können Templates nicht in einer `.cpp`-Datei implementiert werden?
5. Wie funktionieren die Schlüsselwörter `typename` und `template` in Template-Kontexten?

Programmierparadigmen und Entwurfskonzepte

1. Nennen und beschreiben Sie die vier grundlegenden Programmierparadigmen.
2. Was ist funktionale Programmierung und wie wird sie in C++ ermöglicht?
3. Was versteht man unter Abstraktion in der objektorientierten Programmierung?
4. Was unterscheidet ein Konzept (Concept) in C++20 von einem herkömmlichen `enable_if`?
5. Wie wird das Prinzip „Separation of Concerns“ im Softwareentwurf umgesetzt?

Lambdas, Funktionen und moderne Sprachelemente

1. Erklären Sie die Bestandteile einer Lambda-Funktion in C++: `[]() {}`
2. Wie unterscheiden sich `std::function`, Funktionszeiger und Lambdas voneinander?
3. Welche Bedeutung hat die Capture-Liste `[]` in einer Lambda-Funktion?
4. Wie kann man eine Lambda-Funktion mit einem `auto`-Parameter schreiben? Was sind die Vorteile?
5. Wann ist es sinnvoll, `mutable` in einer Lambda-Funktion zu verwenden?

Praxisnahe Fragen – mittleres Niveau

1. Erstellen Sie eine einfache Klasse `SimpleVector`, die ein dynamisches Array kapselt. Welche Memberfunktionen benötigen Sie mindestens, um die Rule of Three zu erfüllen?
2. Welche Vorteile bietet `std::unique_ptr` gegenüber einem rohen Zeiger? In welchem Fall sollte man dennoch keinen `unique_ptr` verwenden?
3. Skizzieren Sie eine einfache `Node`-Struktur für eine einfach verkettete Liste. Worauf müssen Sie beim Löschen der Liste im Destruktor achten?
4. Worin unterscheiden sich Kopier- und Move-Konstruktor semantisch und bezüglich Ressourcenbesitz?
5. Wann entstehen flache und wann tiefe Kopien bei benutzerdefinierten Klassen?

LITERATUR

Alan Kay, The Early History of Smalltalk (programming language)

1993 *definition of "object oriented"*, https://de.wikipedia.org/wiki/Objektorientierte_Programmierung.

The Valuable Dev, Article on Fundamentals

2021 *What Are Abstractions in Software Engineering with Examples*, https://thevaluable.dev/abstraction-type-software-example/.