INF2 – Prüfungs-Zusammenfassung (C / C++)

C: Dateien · Präprozessor & Module · Speicherverwaltung · Listen | C++: Kapselung · Vererbung · Polymorphismus · Konsolidierung · Standard Library

1. C – Dateien lesen/schreiben

Textdatei vs. Binärdatei

- Textdatei: Zeichencodes (ASCII/UTF-8), mit Texteditor lesbar. C nutzt im Wesentlichen ASCII (keine Umlaute). Zeilenende: LF (Linux/Mac) bzw. CR-LF (Windows). Bsp.: "12345\n" → Bytes 49 50 51 52 53 [13] 10.
- Binärdatei: rohe Bytes (raw data), keine Zeichen-Interpretation. Programm muss Aufbau (Records) kennen. Record = zusammengehörende Bytefolge (z.B. 4 Bytes = ein int).

Pfadangaben

Typ	Beispiel
Absolut	Linux/Mac: /home/daten/datei.txt (ab Wurzel /) Windows: c:\\home\\daten\\datei.txt (Laufwerk, \\ als Trenner)
Relativ	datei.txt · ./datei.txt (aktuelles Verz. .) · ../daten/datei.txt (übergeordnet ..)

Wichtige Dateioperationen (stdio.h)

Funktion	Bedeutung
fopen / fclose	Datei öffnen / schliessen. fopen gibt FILE* zurück, im Fehlerfall NULL → immer prüfen!
fprintf / fscanf	formatierte Aus-/Eingabe (wie printf/scanf, aber mit FILE*)
fputc / fgetc	einzelnes Zeichen schreiben/lesen
fputs / fgets	String schreiben/lesen (fgets liest inkl. Zeilenende, sicherer als gets)
fflush	Pufferinhalt sofort in Datei schreiben
remove	Datei löschen

Beispiel: Datei öffnen & schreiben (C)

```
#include <stdio.h>
int main(void) {
    FILE *fp;
    char filename[] = "test.txt";
    fp = fopen(filename, "w");      // "w"=schreiben, "r"=lesen, "a"=anhängen
    if (fp == NULL) {               // IMMER auf NULL prüfen!
        printf("Error opening file %s\n", filename);
        return 1;
    }
    fprintf(fp, "Hello World\n");
    fclose(fp);                     // nicht vergessen!
}
```

String Tokenizer (string.h)

```
char str[] = "one,two,three";
char *token = strtok(str, ",");   // 1. Aufruf: String + Trennzeichen
while (token != NULL) {
    printf("token: %s\n", token);
    token = strtok(NULL, ",");    // weitere Aufrufe: NULL übergeben
}
```

⚠ strtok verändert den Original-String (ersetzt Trenner durch \0) – nicht auf const char anwenden.*

Standard-Streams umleiten (Kommandozeile)

Befehl	Bedeutung
programm > datei	Standardausgabe in Datei schreiben (überschreiben)
programm >> datei	an Datei anhängen
programm < datei	Standardeingabe aus Datei lesen
programm 2> datei	nur Fehlerausgabe in Datei
programm 2> /dev/null	Fehlerausgabe ignorieren (Linux); Windows: 2> NUL

2. C – Präprozessor & Modulkonzept

Drei Aufgaben des Präprozessors: 1) Dateien einfügen (`#include`), 2) Makros ersetzen (`#define`), 3) bedingte Übersetzung (`#ifdef / #ifndef / #endif`).

Include-Guard (Header-Datei) – Standardmuster

```
// doit.h
#ifndef DOIT_H
#define DOIT_H

#define MAX 1000
int doit(int b);          /* nur Deklaration */

#endif
```

```
// main.c                    // doit.c
#include "doit.h"            #include "doit.h"
int main(void) {             int doit(int b) {    /* Definition */
  return MAX + doit(10);       return MAX + b;
}                             }
```

⚠ #include "datei.h" sucht zuerst lokal (eigenes Projekt); #include <datei.h> sucht in System-Verzeichnissen (Standard-Library). Include-Guards verhindern Mehrfach-Einbindung (sonst doppelte Definitionen → Compilerfehler).

#define – Achtung bei Makros

- Makros sind reine Textersetzung, keine Funktionen → keine Typprüfung.
- Klammern setzen! #define prod(a,b) a*(b) → prod(x+2,y-1) wird zu x+2*(y-1), NICHT (x+2)*(y-1) — klassische Prüfungsfalle.
- Magic Numbers vermeiden: #define MAX_ANZAHL 100 statt Zahl direkt im Code. Achtung: Werte wie 100.00 in einer for-Schleife mit int-Zähler können zu Typ-/Vergleichsproblemen führen (Fliesskomma vs. int).

Modulkonzept – Best Practices

- Jedes Modul: eigene .h (Deklarationen/Schnittstelle) + .c (Implementierung).
- Struct-Typen, die in mehreren Modulen gebraucht werden, gehören in eine gemeinsame .h-Datei mit Include-Guard.
- Trennung von Schnittstelle (.h) und Implementierung (.c) ermöglicht getrenntes Compilieren und Wiederverwendung.

3. C – Dynamische Speicherverwaltung

Speicherorganisation

Bereich	Inhalt
Code	Programmcode (read-only)
Daten	globale & static Variablen
Heap	dynamische Allokation (malloc/free) – wächst nach oben
Stack	lokale Variablen, Funktionsparameter – wächst nach unten

malloc / free / calloc / realloc

Funktion	Bedeutung
malloc(n)	alloziert n Bytes, NICHT initialisiert (Müll-Werte)
calloc(n, size)	alloziert n*size Bytes, mit 0 initialisiert
realloc(ptr, n)	vergrössert/verkleinert vorhandenen Block
free(ptr)	gibt Speicher frei – Pflicht, sonst Memory Leak!

Beispiel: dynamisches Array

```
int *iArray;
int n = 0;
scanf("%d", &n);
iArray = malloc(n * sizeof(int));      // immer sizeof verwenden!

iArray[95] = 12;                        // Klammernotation
printf("%d\n", *(iArray + 95));        // Zeigernotation – äquivalent
...
free(iArray);                           // Speicher freigeben
```

Typische Fehler

Begriff	Beschreibung
Memory Leak	Speicher alloziert, aber nie mit free() freigegeben → Speicher bleibt blockiert
Dangling Pointer	Zeiger zeigt auf bereits freigegebenen/ungültigen Speicher (z.B. Adresse einer lokalen Stack-Variable nach Funktionsende)
Doppeltes free()	undefiniertes Verhalten, Programmabsturz möglich

⚠ Lokale (Stack-)Variablen werden nach Funktionsende ungültig — niemals deren Adresse zurückgeben/weiterverwenden (klassische Prüfungsfrage: Rückgabewert ist dann unvorhersehbar/Müll).

4. C – Verkettete Listen

- Jedes Listenelement (struct) enthält Daten + Zeiger auf das nächste Element (next). Letztes Element: next == NULL (Listenende).

Struktur eines Listenelements

```
typedef struct ListElem_s {
    char name[16];
    char first[16];
    int  rank;
    struct ListElem_s *next;     // Zeiger auf nächstes Element
} ListElem;
```

Liste mit Ankerelement (Dummy-Head)

- Problem ohne Anker: Einfügen am Listenanfang muss als Spezialfall behandelt werden (list selbst muss verändert werden).
- Lösung: zusätzliches Dummy-Element als Kopf ("anchor") ohne eigene Daten → Einfügen/Löschen ist überall gleich (list zeigt immer auf den Anker, nie NULL).

Queue (FIFO) – typische Anwendung

- Warteschlange: First In, First Out.
- enqueue(element) – Element an einem Ende einfügen.
- element = dequeue() – Element am anderen Ende entnehmen.

⚠ Stack (LIFO) zum Vergleich: push() fügt oben ein, pop() entnimmt zuletzt eingefügtes Element (siehe Übungsaufgaben).

Typische Operationen / Fallstricke (Prüfung)

- Traversieren: while (curr->next != NULL) { curr = curr->next; } → curr zeigt am Ende auf letztes Element.
- Letztes Element entfernen: man braucht den Vorgänger des letzten Elements (p->next->next == NULL prüfen, dann p->next = NULL und altes letztes Element free()-en).
- Sonderfälle beachten: leere Liste (first == NULL) und Liste mit nur einem Element (first->next == NULL) müssen separat geprüft werden.
- free() erst NACH dem letzten Zugriff auf das Element aufrufen, sonst Dangling Pointer / Use-after-free.

5. C++ – Allgemeine Syntax-Grundlagen (Basics)

Da du in C bereits sicher bist: Diese Tabelle zeigt die wichtigsten Syntax-Unterschiede C → C++ im Überblick.

Bereich	C vs. C++
Header	C: stdio.h, string.h … | C++: <iostream>, <string>, <vector> (ohne .h); C-Header mit c-Präfix: <cstdio>, <cstring>
I/O	C: printf/scanf | C++: std::cout << x; std::cin >> x; (Operatoren << und >>, typsicher)
Strings	C: char[] + string.h | C++: std::string (dynamisch, kein 0-Terminator nötig, + zum Verketten)
Null-Zeiger	C: NULL (#define ... (void*)0) | C++: nullptr (typsicher) – NULL in C++ vermeiden
Bool	C: int (0/1) bzw. C99 bool | C++: bool mit true/false (1/0 kompatibel)
Namen-Konflikte	C: Prefix-Konvention z.B. Foo_bar() | C++: namespace Foo { void bar(); }
Typumwandlung	C: (int)x | C++: static_cast<int>(x) (typsicherer, 4 spezialisierte Casts)
Konstanten	C: #define N 100 (magic, typlos) | C++: constexpr size_t N = 100; (typsicher, compile-time)
Arrays/Listen	C: malloc + struct + Zeiger | C++: std::vector<T>, std::array<T,N> (Standard-Container)
struct/class	C: nur Daten, alles public | C++: struct/class mit Methoden, this-Pointer, public/private
Speicher	C: malloc/free (kein Konstruktor/Destruktor) | C++: new/delete (rufen Konstruktor/Destruktor auf)
Fehlerbehandlung	C: Rückgabewert prüfen (z.B. NULL, -1) | C++: throw/try/catch (Exceptions)

auto – Typ-Inferenz

```
constexpr auto n = 42;            // n ist automatisch int
auto it = data.begin();           // Compiler leitet Iterator-Typ ab
for (const auto& item : data) { ... }   // Range-based for-Loop (idiomatisch)
```

6. C++ – Kapselung (OOP Teil I)

Klassen / Daten + Methoden zusammen

- struct-Typen können in C++ neben Daten auch Funktionen (Methoden) enthalten → Kapselung.
- Innerhalb von Methoden: Zugriff auf Member via `this` (Pointer auf Instanz) – wird vom Compiler implizit angenommen, meist weggelassen.

public / private / class vs. struct

```
class Point {
public:
    Point(double x, double y): x_{x}, y_{y} { }
    Point() = default;              // Compiler generiert Default-Konstruktor
    double getX() const { return x_; }   // const: ändert Objekt nicht
    double getY() const { return y_; }
private:
    double x_{0.0};                 // In-Class Initialisierung (Default-Wert)
    double y_{0.0};
};
```

⚠ class: Member standardmässig private. struct: Member standardmässig public (Kompatibilität zu C). Idiomatisches C++: Member-Variablen private, Zugriff über public Getter/Setter.

Konstruktor & Destruktor

Konzept	Bedeutung
Zweck	Sicherstellen, dass jedes Objekt einen definierten Anfangszustand hat (C: Stack-Variablen haben keinen!)
Syntax Konstruktor	ClassName(Parameter) : InitialisiererListe { FunctionBody }
Syntax Destruktor	~ClassName() { FunctionBody } – wird automatisch am Ende der Lebenszeit aufgerufen
Reihenfolge Init-Liste	Member werden in Deklarations-Reihenfolge initialisiert (nicht in Reihenfolge der Init-Liste!)
RAII	Resource-Acquisition-Is-Initialization: Ressourcen werden im Konstruktor besorgt, im Destruktor automatisch freigegeben

Beispiel Konstruktor-Overloading

```
struct Point {
    Point()                     : x_{0.0}, y_{0.0} { }   // Default
    Point(double x, double y)   : x_{x},   y_{y}   { }   // parametrisiert
    double x_, y_;
};
Point p1;           // ruft Point()      -> x_=0.0, y_=0.0
Point p2{};         // idiomatisch: ruft ebenfalls Point()
Point p3{4.0, 1.0}; // ruft Point(double,double) -> x_=4.0, y_=1.0
```

Referenzen (&) – Objekt-Alias

```
int val = 123;
int &ref = val;   // ref ist Alias auf val (MUSS initialisiert werden)
ref = 17;          // val wird direkt verändert
```

- Referenz: alternativer Name für existierendes Objekt, kann später nicht auf anderes Objekt umgehängt werden.
- Typischer Einsatz: Funktionsparameter (vermeidet Kopie, kein Pointer-Handling nötig).

Operator-Überladung

- Globale Funktion: ReturnTyp operator<<(LhsTyp& lhs, const RhsTyp& rhs) — nötig wenn linker Operand kein eigener Typ ist (z.B. ostream << MyClass).
- Methode: Point& operator*=(int rhs) { ...; return *this; } — Aufruf ist an lhs=Objekt gebunden (p1 *= 3, aber 3 *= p1 geht nicht).

Rule-of-3 / Rule-of-0

- Rule-of-3: Braucht eine Klasse einen eigenen Destruktor (z.B. Ressourcen-Cleanup), braucht sie i.d.R. auch Kopierkonstruktor & Kopier-Zuweisung.
- Rule-of-0: Bevorzugt – keine der speziellen Methoden selbst schreiben; Compiler generiert sie korrekt, wenn nur Member wie std::vector/std::string verwendet werden.
- Wert-Typ: kann konstruiert/kopiert/zugewiesen werden (Compiler generiert automatisch). Nicht-Wert-Typ: Kopie/Move explizit mit = delete verboten (verhindert ungewollte Kopien).

7. C++ – Konsolidierung (new/delete, Smart Pointer, I/O, Templates)

new / delete statt malloc / free

Operator	Bedeutung
new T	alloziert + ruft passenden Konstruktor auf
new T[n]	alloziert Array + ruft Default-Konstruktor auf jedem Element
delete p	ruft Destruktor auf + gibt Speicher frei
delete[] p	ruft Destruktor auf JEDEM Element + gibt Speicher frei

⚠ new[] MUSS mit delete[] freigegeben werden, sonst undefiniertes Verhalten. C- und C++-Speicherverwaltung NICHT mischen (z.B. malloc + delete).

Smart Pointer – std::unique_ptr

- RAII-Wrapper: besitzt ein Objekt exklusiv, löscht es automatisch am Ende des Gültigkeitsbereichs (Scope) – kein manuelles delete nötig, kein Leak möglich.

```
#include <memory>
void f() {
    std::unique_ptr<Point> p{new Point{3.0, 4.0}};
    if (p->getX() > 0.0) { ... }
} // ~unique_ptr ruft hier automatisch delete auf

std::unique_ptr<Point[]> arr{new Point[2]};  // -> ruft automatisch delete[] auf
```

Datei-I/O mit fstream (statt FILE*)

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    ifstream inFile;  ofstream outFile;
    inFile.open("in.txt");
    outFile.open("out.txt");        // ios::app für Anhängen-Modus
    if (inFile.is_open() && outFile.is_open()) {
        string line;
        while (getline(inFile, line)) outFile << line << endl;
    }
    outFile.close();  inFile.close();
}
```

⚠ ifstream = nur lesen, ofstream = nur schreiben, fstream = beides. printf/scanf gilt in C++ als 'nicht idiomatisch'.

Exceptions: throw / try / catch

```
int divide(int a, int b) {
    if (b == 0) throw std::runtime_error("Division by 0!");
    return a / b;
}
try {
    int result = divide(10, 0);
}
catch (const std::exception& e) {     // runtime_error IS-A exception
    std::cout << "Error: " << e.what() << "\n";
}
```

- std::exception ist Basisklasse; std::runtime_error, std::logic_error etc. erben davon. Mehrere catch-Blöcke möglich – spezifischste zuerst, der erste passende wird ausgeführt. catch (...) fängt alles.
- Stack-Unwinding: zwischen throw und passendem catch werden für alle FERTIG konstruierten Objekte automatisch die Destruktoren aufgerufen (Hauptgrund für RAII).

Templates & Standard Library Basics

```
template<typename ArgT>
void printValue(const ArgT& v) { std::cout << v; }   // Code nur 1x geschrieben
printValue(17);     // Compiler generiert printValue<int>
printValue(3.14);   // Compiler generiert printValue<double>
```

Typ	Bedeutung
std::string	dynamischer String, kein 0-Terminator nötig; #include <string>
std::vector<T>	dynamisches Array; push_back(), Index-Zugriff [i], insert(); #include <vector>

8. C++ – Vererbung (OOP Teil II)

Grundsyntax & Begriffe

```
class Vehicle {                         // Basisklasse
public:
    double getSpeed() const { ... }
};
class Car: public Vehicle {             // abgeleitete Klasse
public:
    int getDoors() const { ... }
};
Car car{};
car.getDoors();   // eigene Methode
car.getSpeed();   // geerbte Methode
```

Begriff	Bedeutung
Basis-/Super-/Eltern-Klasse	Klasse, von der vererbt wird (Base/Parent Class)
Abgeleitete/Sub-/Kind-Klasse	Klasse, die erbt (Derived/Child Class)

Zugriffsrechte: public / protected / private

Modifier	Sichtbarkeit
public	überall zugreifbar (auch ausserhalb der Klasse)
protected	nur in dieser Klasse UND in abgeleiteten Klassen zugreifbar
private	nur in dieser Klasse selbst zugreifbar (NICHT in abgeleiteten Klassen!)

```
class Vehicle {
public:    double getSpeed() const { return speed_; }
protected: void setSpeed(double s) { speed_ = s; }   // Basis + abgeleitete Klassen
private:   double speed_{0.0};                        // nur Vehicle selbst
};
class Car: public Vehicle {
public: void drive(double s) { setSpeed(s); }  // OK: protected ist nutzbar
};
```

Konstruktion / Destruktion der Basisklasse

- Konstruktor abgeleitete Klasse: 1) Konstruktor der Basisklasse, 2) Konstruktoren der Member, 3) eigener Funktionsblock.
- Destruktor: genau umgekehrte Reihenfolge: 1) eigener Funktionsblock, 2) Destruktoren der Member, 3) Destruktor der Basisklasse.

```
class Derived: public Base {
public:
    Derived(int x): Base{x} { ... }   // Basis-Konstruktor explizit in Init-Liste
};
```

Slicing – häufiger Prüfungsfehler

```
class Base    { int b_{0}; };
class Derived: public Base { int d_{0}; };
Derived d{};  Base b{};
b = d;   // erlaubt: d-Teil geht VERLOREN ("Slicing")
// d = b;  // NICHT erlaubt (Base ist nicht IS-A Derived)
```

- Derived IS-A Base → Zuweisung Derived an Base erlaubt, aber Daten der abgeleiteten Klasse gehen verloren (Slicing). Umgekehrt nicht möglich.
- Referenzen/Pointer auf Basisklasse können auf Derived-Objekt zeigen OHNE Slicing (Base& r = d; / Base* p = &d;) – aber ohne virtual ist nur die Basis-Schnittstelle sichtbar.

9. C++ – Polymorphismus (OOP Teil III)

virtual / override – Grundprinzip

- virtual in Basisklasse markiert eine Methode als überschreibbar. override in der abgeleiteten Klasse zeigt die Absicht klar an und lässt den Compiler prüfen, dass tatsächlich überschrieben wird (Tippfehler-Schutz).
- Statischer Typ = deklarierter Typ des Pointers/der Referenz (compile-time). Dynamischer Typ = tatsächlicher Typ des referenzierten Objekts (run-time).
- Nicht-virtuelle Methoden werden über den statischen Typ aufgerufen; virtuelle Methoden über den dynamischen Typ – DAS ist der Sinn von Polymorphismus.

Beispiel

```
class Base {
public:
    void printBase() const { cout << "Base\n"; }       // NICHT virtual
    virtual void print() const { cout << "----\n"; }    // virtual
    virtual ~Base() = default;     // WICHTIG: virtueller Destruktor!
};
class Derived: public Base {
public:
    void print() const override { cout << "Derived\n"; }
};
Base& r = derivedObjekt;
r.printBase();  // -> "Base"     (statischer Typ entscheidet)
```

```
r.print();      // -> "Derived"  (dynamischer Typ entscheidet!)
```

⚠ Virtueller Destruktor in der Basisklasse ist Pflicht, sobald über Basisklassen-Pointer gelöscht wird (delete basePtr;) — sonst wird nur ~Base() aufgerufen, nicht ~Derived() → Ressourcen-Leck/undefiniertes Verhalten!

Abstract Base Class (ABC) – pure-virtual

```
class SensorBase {                      // Abstract-Base-Class
public:
    virtual double measure() = 0;       // pure-virtual: KEINE Implementierung
    virtual ~SensorBase() = default;
};
// SensorBase s;  // FEHLER: kann nicht instanziiert werden!
class TemperatureSensor: public SensorBase {
public:
    double measure() override { ... }    // MUSS implementiert werden
};
```

- ABC = mindestens eine pure-virtual Methode (= 0). Kann nicht direkt instanziiert werden. Oft als Interface verwendet (alle Methoden pure-virtual, keine Daten).

static Member (Klassen-Member)

- static Member gehören zur Klasse, nicht zu einzelnen Instanzen — werden von allen Objekten geteilt.
- static Methoden haben KEINEN this-Pointer. static Daten-Member müssen im .cpp-File definiert werden (static wird dort nicht wiederholt).

Verschachtelte Typen

```
class LogBase {
public:
    enum Kind { kInfo, kWarning, kError, kFatal };   // Typ innerhalb Klasse
};
logger.log(LogBase::Kind::kInfo, "Text");   // Zugriff via Qualified Name
```

10. C++ – Standard Library (STL)

STL-Aufbau: Container, Iteratoren, Algorithmen

- Template-basiert: Container (Daten-Haltung), Iteratoren (Traversieren), Algorithmen (Zugriff/Manipulation), Funktions-Objekte.

Wichtigste Container

Container	Beschreibung
std::vector<T>	dynamisches Array, schnellster Zugriff per Index
std::array<T,N>	Array fixer Grösse (compile-time)
std::list<T>	doppelt verkettete Liste
std::map<K,V>	sortierte Key-Value-Paare, eindeutige Keys
std::set<T>	sortierte eindeutige Elemente
std::unordered_map<K,V>	Key-Value Paare, unsortiert (Hash-basiert, oft schneller)
std::stack<T>	LIFO-Adapter (push/pop)
std::queue<T>	FIFO-Adapter (push/pop an unterschiedlichen Enden)

Beispiel: Häufigkeiten zählen mit std::map

```
std::string samples[] = {"apple","banana","apple","orange","banana","apple"};
std::map<std::string, size_t> dict{};
for (const auto& s : samples) dict[s]++;     // dict[s] erstellt Eintrag falls nötig (Wert 0)
for (const auto& e : dict) std::cout << e.first << " --> " << e.second << "\n";
// Ausgabe (Key-sortiert): apple-->3  banana-->2  orange-->1
```

Iteratoren – generisches Muster

```
for (auto it = c.begin(); it != c.end(); ++it) {
    *it = ...;          // vector: *it liefert T&
    it->second = ...;   // map: *it liefert pair<const K,V>&, ->second = Value
}
```

- begin() zeigt auf erstes Element, end() zeigt HINTER das letzte Element (Abbruchkriterium it != c.end()).
- Idiomatisch: ++it statt it++ (Performance bei komplexen Iteratoren).

Wichtige Algorithmen (<algorithm>)

Funktion	Bedeutung
std::find(first,last,val)	findet erstes Element == val, gibt Iterator zurück (oder last, falls nicht gefunden)
std::sort(first,last)	sortiert Bereich aufsteigend
std::count / count_if	zählt Elemente (mit Bedingung)
std::for_each	führt Funktion auf jedem Element aus
std::distance(first,last)	Anzahl Elemente zwischen zwei Iteratoren

11. Prüfungs-Hinweise (aus alten Prüfungen/Minitests)

- Multiple-Choice oft im K-Prim-Format: alle Teilaussagen richtig = volle Punkte, ein Fehler = halbe Punkte, sonst 0 Punkte → sorgfältig JEDE Teilaussage einzeln prüfen!
- Häufige Fallen: Makro-Klammerung (#define ohne Klammern), Shift-Operatoren (<<, >>= sind In-Place), Stack-Variablen-Lebensdauer (Dangling Pointer), Slicing bei Vererbung, virtueller Destruktor vergessen, private vs. protected Vererbungszugriff.
- Code-Ergänzungsfragen: throw std::runtime_error("Text"); werfen, mit try { ... } catch (const std::exception& e) { ... } abfangen und e.what() ausgeben.
- Listen-Aufgaben: Sonderfälle (leere Liste / nur 1 Element) separat behandeln; Logikfehler vs. Syntaxfehler unterscheiden (z.B. *curr.next ist Syntaxfehler für curr->next; Dereferenzieren einer Kopie statt Pointer wäre Logikfehler).
- Bei Datei-Operationen: fopen-Rückgabewert (FILE* bzw. != 0, NICHT 0 im Erfolgsfall) immer auf NULL prüfen; Lese-Funktion für eine Zeile in C: fgets() (nicht readline/getline – das ist C++!).
- constexpr statt #define für typsichere Konstanten (idiomatisches C++); In-Class-Initialisierung für Default-Werte von Member-Variablen nutzen.