1. Typen, Variablen und Konstanten

Fachbegriffe:

- **Datentypen:** Legen fest, welche Art von Daten eine Variable speichern kann (z. B. `int`, `float`, `double`, `char`).
- **Variablen:** Speicherorte, die mit einem Namen versehen sind und einen bestimmten Datentyp haben.
- **Konstanten:** Werte, die sich nach ihrer Initialisierung nicht mehr ändern (z. B. mit `const`).

Warum verwendet man sie?

- Datentypen bestimmen, wie viel Speicherplatz reserviert wird und welche Operationen erlaubt sind.
- Variablen ermöglichen die Wiederverwendung von Speicher und Daten.
- Konstanten schützen vor versehentlicher Änderung und machen den Code lesbarer.

Was passiert, wenn man sie nicht verwendet?

- Ohne Variablen oder Konstanten könnte man keine Werte speichern.
- Ohne den richtigen Datentyp könnten Programme fehlerhaft sein oder abstürzen.

Beispiel:

```
const double PI = 3.14159; // Konstante
int alter = 25; // Variable
float preis = 19.99; // Gleitkommazahl
```

2. Konvertierungen

Fachbegriffe:

- **Implizite Konvertierung:** Automatische Anpassung eines Datentyps an einen anderen (z. B. `int` zu `float`).
- **Explizite Konvertierung (Casting):** Manuelle Umwandlung mit Cast-Operatoren (`(float)x` oder `static_cast<float>(x)`).

Warum verwendet man sie?

- Um Daten zwischen verschiedenen Typen zu verwenden.
- Vermeidung von Kompatibilitätsproblemen zwischen Datentypen.

Was passiert, wenn man sie nicht verwendet?

- Ohne Konvertierungen kann es zu Typkonflikten, Datenverlust oder Laufzeitfehlern kommen.

Beispiel:

```
int a = 10;
float b = static_cast<float>(a) / 3; // Explizite
Konvertierung
```

3. Gültigkeit und Sichtbarkeit

Fachbegriffe:

- **Gültigkeitsbereich:** Bereich im Code, in dem eine Variable verfügbar ist.
- **Sichtbarkeit:** Zugriffsmöglichkeit auf Variablen (z. B. `public`, `private`).

Warum verwendet man sie?

- Verhindert ungewollte Änderungen von Daten außerhalb des vorgesehenen Bereichs.
- Fördert sauberen und sicheren Code.

Was passiert, wenn man sie nicht verwendet?

- Ohne definierte Gültigkeit können Variablen unkontrolliert überschrieben werden.
- Ohne Sichtbarkeitskontrolle wird der Code anfällig für Fehler.

Beispiel:

```
void beispiel() {
    int lokal = 5; // Lokale Variable, nur innerhalb der
Funktion sichtbar
}
```

4. Strukturen

Fachbegriffe:

- **Struktur (`struct`):** Gruppierung von mehreren Variablen zu einer Einheit.
- **Mitglieder:** Einzelne Variablen in einer Struktur.

Warum verwendet man sie?

- Bündelt zusammenhängende Daten in einer Einheit.
- Erleichtert das Verwalten und den Zugriff auf komplexe Datensätze.

Was passiert, wenn man sie nicht verwendet?

- Ohne Strukturen müsste man viele Einzelvariablen verwenden, was den Code unübersichtlich macht.

Beispiel:

```
struct Person {
    string name;
    int alter;
};
```

5. Ein- und Ausgabe

Fachbegriffe:

- **Streams:** Datenströme, z. B. `cin` (Eingabe) und `cout` (Ausgabe).
- **Operatoren (<<, >>):** Ermöglichen das Lesen und Schreiben von Daten.

Warum verwendet man sie?

- Um Daten mit Benutzern oder anderen Programmen auszutauschen.

Was passiert, wenn man sie nicht verwendet?

- Ohne Ein- und Ausgabe gäbe es keine Möglichkeit, mit dem Benutzer zu interagieren.

Beispiel:

```
cout << "Bitte geben Sie Ihr Alter ein: ";
cin >> alter;
```

6. Arrays

Fachbegriffe:

- **Array:** Sammlung von Elementen gleichen Datentyps, gespeichert in zusammenhängendem Speicher.
- **Index:** Position eines Elements im Array (beginnend bei 0).

Warum verwendet man sie?

- Erleichtert das Arbeiten mit mehreren gleichartigen Daten.

Was passiert, wenn man sie nicht verwendet?

- Man müsste jede Variable einzeln definieren und verwalten, was unpraktisch ist.

Beispiel:

```
int zahlen[5] = {1, 2, 3, 4, 5};
```

7. Strings

Fachbegriffe:

- **String:** Zeichenkette, die in C++ durch die `string`-Klasse repräsentiert wird.

Warum verwendet man sie?

- Um Textdaten einfach und effizient zu verwalten.

Was passiert, wenn man sie nicht verwendet?

- Ohne Strings müsste man Arrays von `char` verwenden, was fehleranfälliger ist.

Beispiel:

```
string begruessung = "Hallo, Welt!“;
```

8. Kontrollstrukturen

Fachbegriffe:

- **Bedingungen:** `if`, `else`, `else if` - Kontrollieren die Ausführung basierend auf einer Bedingung.
- **Schleifen:** `for`, `while`, `do-while` - Wiederholen Codeblöcke.
- **Switch-Case:** Auswahlstruktur für mehrere Bedingungen.

Warum verwendet man sie?

- Um Abläufe im Programm zu steuern.
- Wiederholungen und Entscheidungen effizient umzusetzen.

Was passiert, wenn man sie nicht verwendet?

- Der Code würde linear und unflexibel, Wiederholungen müssten mehrfach geschrieben werden.

Beispiel:

```
if (x > 0) {
    cout << "Positiv";
} else {
    cout << "Negativ";
}
```

9. Objektorientierte Programmierung

Fachbegriffe:

- **OOP:** Programmieransatz, der Objekte und Klassen verwendet.
- **Modellierung:** Abbildung realer Probleme auf Objekte.

Warum verwendet man sie?

- Fördert Wiederverwendbarkeit, Stabilität und Nachvollziehbarkeit des Codes.
- Reduziert die Kluft zwischen Modellierung und Implementierung.

Was passiert, wenn man sie nicht verwendet?

- Der Code wird unübersichtlich und schwer erweiterbar.

Beispiel:

```
class Fahrzeug {
    string modell;
    int baujahr;
};
```

10. Objekte kapseln Daten und Funktionen

Fachbegriffe:

- **Kapselung:** Versteckt interne Details eines Objekts (private, protected).
- **Zugriffsmethoden:** Ermöglichen den Zugriff auf private Daten (Getter, Setter).

Warum verwendet man sie?

- Schützt Daten vor ungewollten Änderungen.
- Fördert sauberen und sicheren Code.

Was passiert, wenn man sie nicht verwendet?

- Daten könnten beliebig verändert werden, was Fehler begünstigt.

Beispiel:

```
class Konto {
private:
    double saldo;

public:
    void setSaldo(double betrag) { saldo = betrag; }
    double getSaldo() { return saldo; }
};
```

11. Nachvollziehbarkeit der Implementierung

Fachbegriffe:

- **Code-Lesbarkeit:** Verständlicher Aufbau von Klassen und Methoden.
- **Dokumentation:** Kommentierte Implementierung.

Warum verwendet man sie?

- Erleichtert das Verstehen und Warten des Codes.
- Hilft Teams, effizient zusammenzuarbeiten.

Was passiert, wenn man sie nicht verwendet?

- Der Code wird schwer zu verstehen und zu pflegen.

Beispiel:

```
// Klasse zur Verwaltung von Studenten
class Student {
    // Attribute und Methoden ...
};
```

12. Programmstabilität

Fachbegriffe:

- **Fehlervermeidung:** Kontrolle von Eingaben und Programmlogik.
- **Fehlerbehandlung:** Umgang mit unerwarteten Situationen.

Warum verwendet man sie?

- Reduziert Abstürze und unvorhergesehenes Verhalten.
- Sichert Programmausführung bei fehlerhaften Eingaben.

Was passiert, wenn man sie nicht verwendet?

- Das Programm wird anfälliger für Fehler und Abstürze.

Beispiel:

```
if (eingabe > 0) {
    // Weiterverarbeitung
} else {
    cout << "Ungültige Eingabe";
}
```

13. Wiederverwendbarkeit

Fachbegriffe:

- **Modularer Code:** Aufteilung in unabhängige, wiederverwendbare Module.
- **Klassen und Funktionen:** Ermöglichen Mehrfachverwendung von Code.

Warum verwendet man sie?

- Spart Zeit und reduziert redundanten Code.
- Erleichtert die Wartung und Erweiterung.

Was passiert, wenn man sie nicht verwendet?

- Es entsteht viel redundanter Code, der schwer zu warten ist.

Beispiel:

```
void druckeName(string name) {
    cout << "Name: " << name;
}
```

14. Gleiche Objekte durch Klassen beschrieben

Fachbegriffe:

- **Klasse:** Bauplan für Objekte.
- **Objekt:** Instanz einer Klasse.

Warum verwendet man sie?

- Einheitliche Beschreibung von ähnlichen Entitäten.
- Erleichtert das Erstellen und Verwalten von Objekten.

Was passiert, wenn man sie nicht verwendet?

- Jede Instanz müsste individuell definiert werden, was ineffizient ist.

Beispiel:

```
class Fahrzeug {
    string typ;
    int baujahr;
};
Fahrzeug auto1, auto2;
```

15. Vererbung

Fachbegriffe:

- **Basisklasse:** Gemeinsame Eigenschaften und Methoden.
- **Abgeleitete Klasse:** Erbt von einer Basisklasse.

Warum verwendet man sie?

- Fördert Wiederverwendbarkeit und erweitert Funktionalität.

Was passiert, wenn man sie nicht verwendet?

- Gemeinsamkeiten müssten in jeder Klasse separat implementiert werden.

Beispiel:

```
class Fahrzeug {
protected:
    string typ;
};

class Auto : public Fahrzeug {
    int türen;
};
```

16. Klassenelemente (Daten-Attribute und Methoden)

Fachbegriffe:

- **Attribute:** Variablen innerhalb einer Klasse.
- **Methoden:** Funktionen innerhalb einer Klasse.

Warum verwendet man sie?

- Bündeln zusammengehörige Daten und Funktionen.
- Erleichtert die Arbeit mit Objekten.

Was passiert, wenn man sie nicht verwendet?

- Daten und Funktionen wären nicht logisch verbunden.

Beispiel:

```
class Fahrzeug {
private:
    string typ;
public:
    void setTyp(string t) { typ = t; }
};
```

17. Der this-Zeiger repräsentiert ein Objekt

Fachbegriffe:

- **this-Zeiger:** Zeiger, der auf das aktuelle Objekt einer Klasse verweist.
- **Selbstreferenz:** Zugriff auf das eigene Objekt innerhalb der Klasse.

Warum verwendet man ihn?

- Um auf Mitglieder eines Objekts zuzugreifen, besonders bei Namenskonflikten.
- Wird häufig für Methodenkaskaden oder Rückgabewerte verwendet.

Was passiert, wenn man ihn nicht verwendet?

- Ohne `this` können Namenskonflikte entstehen, und die Implementierung bestimmter Logiken wird schwieriger.

Beispiel:

```
class Konto {
    double saldo;
public:
    Konto& setSaldo(double saldo) {
        this->saldo = saldo; // this löst den Konflikt
        return *this;
    }
};
```

18. Die Klasse ist ein eigener Gültigkeitsbereich

Fachbegriffe:

- **Gültigkeitsbereich:** Innerhalb der Klasse sind Attribute und Methoden nur in ihrem Kontext sichtbar.
- **Encapsulation (Kapselung):** Einschränkung des Zugriffs auf Daten.

Warum verwendet man sie?

- Verhindert Konflikte mit globalen oder anderen lokalen Variablen.
- Macht den Code modularer und sicherer.

Was passiert, wenn man es nicht berücksichtigt?

- Globale Namenskonflikte können auftreten, und der Code wird weniger robust.

Beispiel:

```
class Fahrzeug {
    int baujahr; // Nur innerhalb der Klasse sichtbar
};
```

19. Klassenelemente können geschützt werden

Fachbegriffe:

- **private, protected, public:** Zugriffsmodifikatoren für Klassenelemente.
- **Datenkapselung:** Schützt interne Daten vor direktem Zugriff.

Warum verwendet man sie?

- Um die Integrität der Daten sicherzustellen.
- Verhindert, dass Außenstehende ungewollt Änderungen vornehmen.

Was passiert, wenn man sie nicht verwendet?

- Interne Daten könnten manipuliert oder fehlerhaft verwendet werden.

Beispiel:

```
class Konto {
private:
    double saldo; // Geschützt vor direktem Zugriff
};
```

20. Klassen können mit `class` oder `struct` definiert werden

Fachbegriffe:

- **class:** Standardmäßig private Zugriffsrechte.
- **struct:** Standardmäßig öffentliche Zugriffsrechte.

Warum verwendet man sie?

- Unterschiedliche Zugriffsstandards bieten Flexibilität bei der Modellierung.

Was passiert, wenn man sie nicht verwendet?

- Unterschiedliche Anwendungsfälle könnten schwerer abgebildet werden.

Beispiel:

```
class Fahrzeug {
    string modell; // Private
};

struct Person {
    string name; // Public
};
```

21. Implementierung von Methoden innerhalb oder außerhalb der Klasse

Fachbegriffe:

- **Inline-Methoden:** Methoden, die innerhalb der Klasse definiert sind.
- **Externe Implementierung:** Definition außerhalb der Klasse mit Scope-Operator (`::`).

Warum verwendet man sie?

- Flexibilität bei der Strukturierung des Codes.
- Externe Implementierung hält Klassendefinitionen übersichtlich.

Was passiert, wenn man es nicht verwendet?

- Der Code könnte unübersichtlich und schwer wartbar werden.

Beispiel:

```
class Fahrzeug {
    void starten(); // Deklaration
};

void Fahrzeug::starten() { // Externe Implementierung
    cout << "Motor gestartet.";
}
```

22. Der Scope-Operator und der this-Zeiger zur Namensauflösung

Fachbegriffe:

- **Scope-Operator `::`:** Verknüpft eine Methode oder Variable mit einer Klasse.
- **this-Zeiger:** Verwendet bei Namenskonflikten innerhalb eines Objekts.

Warum verwendet man sie?

- Um Namenskonflikte zu lösen.
- Erleichtert die Zuordnung von Methoden und Variablen.

Was passiert, wenn man sie nicht verwendet?

- Namenskonflikte könnten zu Fehlern oder unvorhersehbarem Verhalten führen.

Beispiel:

```
class Fahrzeug {
    static int anzahl;
public:
    static void setAnzahl(int a) { Fahrzeug::anzahl = a; } //
Scope-Operator
};
```

23. Standardkonstruktor

Fachbegriffe:

- **Standardkonstruktor:** Konstruktor ohne Parameter.
- **Automatisch generiert:** Wird erstellt, wenn kein anderer Konstruktor definiert wurde.

Warum verwendet man ihn?

- Initialisiert Objekte mit Standardwerten.
- Ermöglicht die Erstellung von Objekten ohne zusätzliche Parameter.

Was passiert, wenn man ihn nicht hat?

- Objekte könnten ohne gültige Initialisierung existieren.

Beispiel:

```
class Fahrzeug {
public:
    Fahrzeug() { cout << "Standardkonstruktor aufgerufen."; }
};
```

24. Konstruktoren mit Parametern

Fachbegriffe:

- **Parametrisierter Konstruktor:** Konstruktor, der Werte zur Initialisierung akzeptiert.

Warum verwendet man sie?

- Um Objekte direkt mit spezifischen Werten zu erstellen.

Was passiert, wenn man sie nicht verwendet?

- Jedes Objekt müsste nachträglich initialisiert werden.

Beispiel:

```
class Fahrzeug {
public:
    Fahrzeug(string typ) { cout << "Fahrzeugtyp: " << typ; }
};
```

25. Initialisierungsliste

Fachbegriffe:

- **Initialisierungsliste:** Direktes Initialisieren von Attributen im Konstruktor.
- **Effizienz:** Vermeidet doppelte Initialisierung.

Warum verwendet man sie?

- Reduziert Overhead und verbessert die Lesbarkeit.
- Unverzichtbar für `const` oder Referenz-Attribute.

Was passiert, wenn man sie nicht verwendet?

- Attribute werden möglicherweise zweimal initialisiert, was ineffizient ist.

Beispiel:

```
class Fahrzeug {
    const int baujahr;
public:
    Fahrzeug(int jahr) : baujahr(jahr) {} //
Initialisierungsliste
};
```

26. Kopierkonstruktor

Fachbegriffe:

- **Kopierkonstruktor:** Konstruktor, der ein neues Objekt als Kopie eines bestehenden Objekts erstellt.
- **Standardkopierkonstruktor:** Automatisch generierter Konstruktor, der eine flache Kopie erstellt.

Warum verwendet man ihn?

- Um ein neues Objekt exakt zu kopieren.
- Nützlich bei Klassen mit dynamischem Speicher (tiefe Kopie).

Was passiert, wenn man ihn nicht verwendet?

- Der Standardkopierkonstruktor kann unerwünschte flache Kopien erstellen, was zu Speicherproblemen führt.

Beispiel:

```
class Fahrzeug {
    string typ;
public:
    Fahrzeug(const Fahrzeug& f) { typ = f.typ; } //
Kopierkonstruktor
};
```

27. Konvertierkonstruktor

Fachbegriffe:

- **Konvertierkonstruktor:** Konstruktor, der einen Wert eines anderen Typs in ein Objekt umwandelt.

Warum verwendet man ihn?

- Ermöglicht Typkonvertierungen und vereinfacht die Erstellung von Objekten.

Was passiert, wenn man ihn nicht verwendet?

- Konvertierungen wären weniger flexibel, und der Compiler müsste andere Wege finden.

Beispiel:

```
class Fahrzeug {
    string typ;
public:
    Fahrzeug(string t) { typ = t; } // Konvertierkonstruktor
};
```

28. Konstruktoraufruf

Fachbegriffe:

- **Konstruktoraufruf:** Automatische Initialisierung beim Erstellen eines Objekts.
- **Mehrere Konstruktoren:** Unterschiedliche Konstruktoren für verschiedene Initialisierungsarten.

Warum verwendet man ihn?

- Um Objekte direkt bei der Erstellung zu initialisieren.

Was passiert, wenn man ihn nicht verwendet?

- Objekte wären uninitialisiert, was zu undefiniertem Verhalten führen könnte.

Beispiel:

```
Fahrzeug auto1("SUV"); // Konstruktor mit Parameter
aufgerufen
```

29. Destruktoren

Fachbegriffe:

- **Destruktor:** Spezielle Methode, die aufgerufen wird, wenn ein Objekt zerstört wird.
- **Speicherfreigabe:** Wird häufig verwendet, um dynamisch zugewiesenen Speicher freizugeben.

Warum verwendet man ihn?

- Um Ressourcen zu bereinigen (z. B. Speicher, Dateien).
- Verhindert Speicherlecks.

Was passiert, wenn man ihn nicht verwendet?

- Ressourcen könnten nicht ordnungsgemäß freigegeben werden.

Beispiel:

```
class Fahrzeug {
public:
    ~Fahrzeug() { cout << "Destruktor aufgerufen."; }
};
```

30. Standardmethoden und -operatoren

Fachbegriffe:

- **Standardmethoden:** Automatisch generierte Methoden (z. B. Standardkonstruktor, Kopierkonstruktor).
- **Operatoren:** Vorbereitete Funktionen, die mit Operatoren wie `+`, `–` arbeiten.

Warum verwendet man sie?

- Sie erleichtern grundlegende Operationen und sparen Entwicklungszeit.

Was passiert, wenn man sie nicht verwendet?

- Der Entwickler müsste alle Methoden und Operatoren manuell implementieren.

Beispiel:

```
Fahrzeug auto1, auto2;
auto2 = auto1; // Standardzuweisung
```

31. Statische Klassenelemente

Fachbegriffe:

- **Statische Elemente:** Gehören zur Klasse, nicht zu einzelnen Objekten.
- **Zugehörigkeit:** Werden für alle Objekte der Klasse geteilt.

Warum verwendet man sie?

- Für globale Eigenschaften oder Zähler, die von allen Objekten geteilt werden.

Was passiert, wenn man sie nicht verwendet?

- Es könnten redundante Kopien von gemeinsamen Daten entstehen.

Beispiel:

```
class Fahrzeug {
    static int anzahl;
};
int Fahrzeug::anzahl = 0; // Initialisierung
```

32. Klassenvariablen müssen außerhalb der Klasse initialisiert werden

Fachbegriffe:

- **Statische Variablen:** Müssen außerhalb der Klasse definiert und initialisiert werden.

Warum verwendet man sie?

- Weil sie von allen Objekten geteilt werden und nicht Teil einer einzelnen Instanz sind.

Was passiert, wenn man sie nicht initialisiert?

- Es kommt zu Kompilierfehlern.

Beispiel:

```
int Fahrzeug::anzahl = 0; // Statische Variable initialisiert
```

33. Konstante Datenelemente in Objekten

Fachbegriffe:

- **Konstante Attribute:** Können nur einmal im Konstruktor initialisiert werden.

Warum verwendet man sie?

- Um sicherzustellen, dass sich ein Attribut nach der Initialisierung nicht mehr ändert.

Was passiert, wenn man sie nicht verwendet?

- Es könnten ungewollte Änderungen auftreten.

Beispiel:

```
class Fahrzeug {
    const int baujahr;
public:
    Fahrzeug(int jahr) : baujahr(jahr) {} // Initialisierung
};
```

34. Konstante Instanzfunktionen

Fachbegriffe:

- **Konstante Methoden:** Können keine Änderungen an den Attributen eines Objekts vornehmen.

Warum verwendet man sie?

- Um sicherzustellen, dass der Zustand eines Objekts in bestimmten Methoden unverändert bleibt.

Was passiert, wenn man sie nicht verwendet?

- Ungewollte Änderungen am Objekt könnten passieren.

Beispiel:

```
class Fahrzeug {
    int baujahr;
public:
    int getBaujahr() const { return baujahr; } // Konstante
Methode
};
```

35. Freundfunktionen und Freundklassen

Fachbegriffe:

- **Freundfunktion:** Funktion, die Zugriff auf private und geschützte Elemente einer Klasse hat.
- **Freundklasse:** Klasse, die auf private Elemente einer anderen Klasse zugreifen kann.

Warum verwendet man sie?

- Um spezielle Zugriffsmöglichkeiten zu gewähren, ohne die gesamte Klasse öffentlich zu machen.

Was passiert, wenn man sie nicht verwendet?

- Zugriff auf private Daten müsste durch umständliche Methoden erfolgen.

Beispiel:

```
class Fahrzeug {
    friend void zeigeTyp(const Fahrzeug& f); //
Freundfunktion
};
```

36. Operatorüberladung

Fachbegriffe:

- **Operatorüberladung:** Ermöglicht es, Operatoren für benutzerdefinierte Klassen anzupassen.
- **Methoden und Funktionen:** Können für Operatoren geschrieben werden.

Warum verwendet man sie?

- Um benutzerdefinierte Objekte intuitiv zu manipulieren.

Was passiert, wenn man sie nicht verwendet?

- Standardoperatoren könnten nicht mit benutzerdefinierten Klassen arbeiten.

Beispiel:

```
class Fahrzeug {
    int anzahl;
public:
    Fahrzeug operator+(const Fahrzeug& f) {
        Fahrzeug temp;
        temp.anzahl = this->anzahl + f.anzahl;
        return temp;
    }
};
```

37. Operatoren als Methoden oder Funktionen

Fachbegriffe:

- **Operator als Methode:** Wird in der Klasse definiert und der erste Operand ist das Objekt (`this`).
- **Operator als Funktion:** Wird außerhalb der Klasse definiert und alle Operanden werden als Parameter übergeben.

Warum verwendet man sie?

- Um flexibler zu sein: Methoden eignen sich gut für Klassenmitglieder, Funktionen für unabhängige Operatoren.

Was passiert, wenn man sie nicht verwendet?

- Standardoperatoren könnten nicht an benutzerdefinierte Objekte angepasst werden.

Beispiel:

```
class Fahrzeug {
    int baujahr;
public:
    bool operator>(const Fahrzeug& f) const { return baujahr
> f.baujahr; } // Methode
};

bool operator==(const Fahrzeug& f1, const Fahrzeug& f2)
{ return f1.baujahr == f2.baujahr; } // Funktion
```

38. Vererbung und Sichtbarkeit

Fachbegriffe:

- **public, protected, private Vererbung:** Kontrolliert, wie die Basisklassenmitglieder in der abgeleiteten Klasse sichtbar sind.
- **Basisklasse und abgeleitete Klasse:** Die abgeleitete Klasse erbt von der Basisklasse.

Warum verwendet man sie?

- Um bestehende Funktionalitäten wiederzuverwenden und zu erweitern.
- Sichtbarkeit sichert, dass sensible Daten nicht ungewollt vererbt werden.

Was passiert, wenn man sie nicht verwendet?

- Man müsste wiederholt ähnliche Klassen schreiben.

Beispiel:

class Fahrzeug {

```
protected:
    string typ; };

class Auto : public Fahrzeug {
    int türen;};
```

39. Mehrfache Vererbung

Fachbegriffe:

- **Mehrfache Vererbung:** Eine Klasse erbt von mehr als einer Basisklasse.
- **Diamantproblem:** Konflikt bei mehrfacher Vererbung derselben Basisklasse.

Warum verwendet man sie?

- Ermöglicht das Kombinieren von Funktionalitäten mehrerer Klassen.

Was passiert, wenn man sie nicht verwendet?

- Bestimmte Designanforderungen könnten schwerer umsetzbar sein.

Beispiel:

```
class Fahrzeug {
    string typ;
};

class Motor {
    int ps;
};

class Auto : public Fahrzeug, public Motor {};
```

40. Objekte identifizieren

Fachbegriffe:

- **Objekt-ID:** Kann durch Attribute oder Pointer realisiert werden.
- **Vergleich:** Identifikation durch spezifische Werte oder Adressen.

Warum verwendet man sie?

- Um individuelle Objekte innerhalb einer Datenstruktur oder im Speicher zu unterscheiden.

Was passiert, wenn man sie nicht verwendet?

- Objekte könnten verwechselt werden, was zu falschen Ergebnissen führt.

Beispiel:

```
Fahrzeug f1, f2;
if (&f1 == &f2) { cout << "Gleiche Objekte"; } // Vergleich
durch Adresse
```

41. Modularisierung

Fachbegriffe:

- **Module:** Unterteilung des Codes in kleinere, unabhängige Einheiten (z. B. Dateien).
- **Kohäsion:** Jedes Modul hat eine spezifische Aufgabe.

Warum verwendet man sie?

- Erleichtert Wartung und Wiederverwendung.
- Reduziert Abhängigkeiten zwischen Codeabschnitten.

Was passiert, wenn man sie nicht verwendet?

- Der Code wird unübersichtlich und schwer wartbar.

Beispiel:

```
// Fahrzeug.h
class Fahrzeug {
    string typ;
};
```

42. Templates

Fachbegriffe:

- **Template:** Schablone für Funktionen oder Klassen, die mit beliebigen Datentypen arbeitet.
- **Typparameter:** Platzhalter für Datentypen.

Warum verwendet man sie?

- Um allgemeinen Code für verschiedene Datentypen zu schreiben.
- Verhindert redundante Implementationen.

Was passiert, wenn man sie nicht verwendet?

- Man müsste für jeden Datentyp eigene Implementationen schreiben.

Beispiel:

```
template <typename T>
T add(T a, T b) { return a + b; }
```

43. Defaultwerte

Fachbegriffe:

- **Standardwerte:** Vordefinierte Werte für Funktionsparameter.
- **Effizienz:** Reduziert die Schreibarbeit bei häufigen Funktionsaufrufen.

Warum verwendet man sie?

- Um die Funktionalität flexibler zu machen.

Was passiert, wenn man sie nicht verwendet?

- Der Funktionsaufruf würde immer explizit alle Argumente benötigen.

Beispiel:

```
void begruessung(string name = "Gast") { cout << "Hallo, " << name; }
```

44. Container

Fachbegriffe:

- **Container:** Datenstrukturen zur Verwaltung von Objekten (z. B. `vector`, `list`).
- **STL:** Standard Template Library mit vorgefertigten Containern.

Warum verwendet man sie?

- Um Daten effizient zu speichern und zu verwalten.

Was passiert, wenn man sie nicht verwendet?

- Man müsste eigene Datenstrukturen entwickeln, was aufwendig ist.

Beispiel:

```
#include <vector>
vector<int> zahlen = {1, 2, 3};
```

45. Strings in C++

Fachbegriffe:

- **`std::string`:** Klasse für Zeichenketten.
- **Dynamische Speicherverwaltung:** Automatische Anpassung der String-Länge.

Warum verwendet man sie?

- Um Textdaten bequem zu speichern und zu bearbeiten.

Was passiert, wenn man sie nicht verwendet?

- Man müsste mit `char`-Arrays arbeiten, was fehleranfälliger ist.

Beispiel:

```
string text = "Hallo Welt";
```

46. String-Streams

Fachbegriffe:

- **String-Stream:** Klasse zur Verarbeitung von Strings als Stream (`stringstream`).
- **Umwandlung:** Einfache Konvertierung zwischen Datentypen und Strings.

Warum verwendet man sie?

- Um Strings flexibel zu verarbeiten und zu manipulieren.

Was passiert, wenn man sie nicht verwendet?

- Umständliche Konvertierungen und Manipulationen wären nötig.

Beispiel:

```
#include <sstream>
stringstream ss;
ss << 42;
string text = ss.str();
```

47. Virtuelle Methoden

Fachbegriffe:

- **Virtuell:** Methoden, die in abgeleiteten Klassen überschrieben werden können.
- **Polymorphie:** Laufzeitbindung von Methoden.

Warum verwendet man sie?

- Um eine flexible und erweiterbare Klassenhierarchie zu schaffen.

Was passiert, wenn man sie nicht verwendet?

- Der Code wäre weniger dynamisch und nicht polymorph.

Beispiel:

```
class Fahrzeug {
    virtual void fahren() { cout << "Fahrzeug fährt."; }
};
```

48. Standard-Typumwandlungen (Implizite Konvertierungen)

Fachbegriffe:

- **Implizite Konvertierung:** Automatische Anpassung eines Typs an einen kompatiblen Typ.
- **Beispiel:** `int` zu `float`.

Warum verwendet man sie?

- Erleichtert die Arbeit mit gemischten Datentypen.

Was passiert, wenn man sie nicht verwendet?

- Der Entwickler müsste explizite Konvertierungen angeben, was den Code umständlicher macht.

Beispiel:

```
int a = 5;
float b = a; // Implizite Konvertierung
```

49. Informationsverlust bei Konvertierungen

Fachbegriffe:

- **Informationsverlust:** Tritt auf, wenn ein Typ nicht alle Daten eines anderen Typs darstellen kann.
- **Beispiel:** `float` zu `int` (Dezimalstellen gehen verloren).

Warum ist das wichtig?

- Um mögliche Fehler und Ungenauigkeiten zu vermeiden.

Was passiert, wenn man es ignoriert?

- Datenverluste könnten zu falschen Ergebnissen führen.

Beispiel:

```
float x = 5.75;
int y = x; // Verlust der Dezimalstellen
```

50. Explizite Konvertierung in C++

Fachbegriffe:

- **`static_cast`:** Sichere Umwandlung zwischen verwandten Typen.
- **`dynamic_cast`:** Laufzeitprüfung bei Zeiger-Konvertierungen.
- **`reinterpret_cast`:** Bitweise Umwandlung ohne Typprüfung.
- **`const_cast`:** Entfernt oder fügt `const` hinzu.

Warum verwendet man sie?

- Um Konvertierungen sicherer und klarer zu machen.

Was passiert, wenn man sie nicht verwendet?

- Der Code wäre schwerer zu verstehen und potenziell unsicher.

Beispiel:

```
int a = 42;
float b = static_cast<float>(a); // Explizite Konvertierung
```

51. Dynamic Cast und seine Grenzen

Fachbegriffe:

- **`dynamic_cast`:** Funktioniert nur mit Zeigern oder Referenzen in polymorphen Klassenhierarchien.
- **Grenze:** Kann nicht mit Basistypen verwendet werden.

Warum verwendet man es?

- Um sicherzustellen, dass ein Objekt sicher in eine abgeleitete Klasse konvertiert wird.

Was passiert, wenn man es nicht verwendet?

- Es könnte zu undefiniertem Verhalten kommen.

Beispiel:

```
class Fahrzeug {
    virtual void fahren() {}
};

class Auto : public Fahrzeug {};

Fahrzeug* f = new Auto();
Auto* a = dynamic_cast<Auto*>(f); // Sicherer Cast
```

52. Abstrakte Klassen

Fachbegriffe:

- **Abstrakte Klasse:** Enthält mindestens eine rein virtuelle Methode (`= 0`).
- **Schnittstellen:** Dient als Grundlage für abgeleitete Klassen.

Warum verwendet man sie?

- Um ein gemeinsames Interface für verschiedene Klassen zu definieren.

Was passiert, wenn man sie nicht verwendet?

- Es wäre schwieriger, eine klare Struktur für Polymorphie zu schaffen.

Beispiel:

```
class Fahrzeug {
    virtual void fahren() = 0; // Reine virtuelle Methode
};
```

53. Ausnahmebehandlung in C++

Fachbegriffe:

- **`try`, `catch`, `throw`:** Mechanismen für das Abfangen von Ausnahmen.
- **`noexcept`:** Deklariert Funktionen als ausnahmesicher.

Warum verwendet man sie?

- Um Fehler strukturiert zu behandeln, ohne das Programm sofort abzubrechen.

Was passiert, wenn man sie nicht verwendet?

- Fehler würden das Programm unerwartet beenden.

Beispiel:

```
try {
    throw runtime_error("Fehler aufgetreten");
} catch (const exception& e) {
    cout << e.what();
}
```

54. Reihenfolge in Ausnahme-Hierarchien

Fachbegriffe:

- **Handler-Reihenfolge:** Spezifischere Ausnahmen müssen vor allgemeinen behandelt werden.

Warum ist das wichtig?

- Sicherstellt, dass spezifische Fehler zuerst abgefangen werden.

Was passiert, wenn man es ignoriert?

- Allgemeine Handler könnten spezifische überschreiben.

Beispiel:

```
try {
    throw 5;
} catch (int e) {
    cout << "Integer Exception";
} catch (...) {
    cout << "Allgemeiner Fehler";
}
```

55. Ausnahmeobjekte weiterwerfen

Fachbegriffe:

- **Weiterwerfen:** Ein `throw` innerhalb eines `catch`-Blocks.

Warum verwendet man es?

- Um Fehlerbehandlung auf eine höhere Ebene zu verschieben.

Was passiert, wenn man es nicht verwendet?

- Fehler können nicht an andere Schichten delegiert werden.

Beispiel:

```
try {
    throw 42;
} catch (int e) {
    throw; // Weiterwerfen
}
```

56. Noexcept-Funktionen

Fachbegriffe:

- **`noexcept`:** Deklaration, dass eine Funktion keine Ausnahmen wirft.

Warum verwendet man sie?

- Optimiert den Code und garantiert Ausnahmesicherheit.

Was passiert, wenn man es nicht verwendet?

- Der Compiler kann keine Garantie über die Stabilität der Funktion geben.

Beispiel:

```
void sichereFunktion() noexcept {
    // Diese Funktion wirft keine Ausnahmen
}
```

57. Ausnahmebehandlung ohne OOP

Fachbegriffe:

- **Ausnahmen:** Funktioniert auch in prozeduralem Code ohne Klassen.

Warum ist das wichtig?

- Fehlerbehandlung unabhängig von OOP.

Was passiert, wenn man sie nicht verwendet?

- Prozeduraler Code müsste andere fehleranfälligere Methoden verwenden.

Beispiel:

```
try {
    throw "Fehler!";
} catch (const char* msg) {
    cout << msg;
}
```

58. IO-Streams

Fachbegriffe:

- **`cin`, `cout`:** Eingabe- und Ausgabestreams.
- **`cerr`:** Fehlerausgabe.

Warum verwendet man sie?

- Für einfache Ein- und Ausgabeoperationen.

Was passiert, wenn man sie nicht verwendet?

- Benutzerinteraktionen wären nicht möglich.

Beispiel:

```
cout << "Geben Sie eine Zahl ein: ";
cin >> zahl;
```

59. Stream-Formatierung

Fachbegriffe:

- **`std::setw`, `std::setprecision`:** Kontrollieren Ausgabeformat.

Warum verwendet man sie?

- Um Ausgaben übersichtlich und präzise zu formatieren.

Was passiert, wenn man sie nicht verwendet?

- Die Ausgabe könnte schwer lesbar sein.

Beispiel:

```
#include <iomanip>
cout << setw(10) << 42;
```

60. Virtuelle Methoden

Fachbegriffe:

- **Virtuelle Methoden:** Funktionen, die in Basisklassen definiert und in abgeleiteten Klassen überschrieben werden können.
- **Polymorphie:** Laufzeitbindung, bei der die korrekte Methode basierend auf dem Objekttyp aufgerufen wird.

Warum verwendet man sie?

- Ermöglicht flexibles Verhalten und erweiterbare Klassenhierarchien.

Was passiert, wenn man sie nicht verwendet?

- Es wird die Methode der Basisklasse aufgerufen, selbst wenn ein Objekt der abgeleiteten Klasse verwendet wird.

Beispiel:

```
class Fahrzeug {
public:
    virtual void fahren() { cout << "Fahrzeug fährt."; }
};

class Auto : public Fahrzeug {
public:
    void fahren() override { cout << "Auto fährt."; }
};
```

61. Typumwandlung mit Cast-Operatoren

Fachbegriffe:

- **`static_cast`:** Kompilierzeitüberprüfung, geeignet für verwandte Typen.
- **`dynamic_cast`:** Laufzeitüberprüfung, funktioniert nur mit Zeigern oder Referenzen.
- **`reinterpret_cast`:** Unsichere Umwandlung, z. B. zwischen nicht verwandten Zeigern.
- **`const_cast`:** Entfernt oder fügt `const` hinzu.

Warum verwendet man sie?

- Um sichere und kontrollierte Konvertierungen zu ermöglichen.

Was passiert, wenn man sie nicht verwendet?

- Der Code wird schwer lesbar, und es können unsichere Konvertierungen auftreten.

Beispiel:

```
int a = 10;
float b = static_cast<float>(a); // Sichere Typumwandlung
```

62. Dynamische Speicherverwaltung bei Strings

Fachbegriffe:

- **Dynamische Speicherverwaltung:** Strings passen ihre Größe dynamisch an.
- **Automatische Freigabe:** Speicher wird beim Löschen eines Strings freigegeben.

Warum verwendet man sie?

- Um flexible und speichereffiziente Textbearbeitung zu ermöglichen.

Was passiert, wenn man sie nicht verwendet?

- Der Entwickler müsste manuell Speicher reservieren und freigeben, was fehleranfällig ist.

Beispiel:

```
string text = "Hallo Welt";
text += "!";
```

63. String-Konvertierungen

Fachbegriffe:

- **C-Strings (`char*`):** Ältere String-Darstellung.
- **`std::string`:** Komfortable, moderne String-Klasse.

Warum verwendet man sie?

- Erleichtert den Übergang zwischen alter und neuer String-Implementierung.

Was passiert, wenn man sie nicht verwendet?

- Arbeiten mit gemischten Stringtypen wäre schwierig.

Beispiel:

```
string text = "Beispiel";
const char* cstr = text.c_str(); // Umwandlung in C-String
```

64. String-Streams zur Verarbeitung

Fachbegriffe:

- **`std::stringstream`:** Stream zur Manipulation von Strings.
- **Konvertierungen:** Erlaubt einfache Typ-String-Umwandlungen.

Warum verwendet man sie?

- Um Strings flexibel zu bearbeiten oder Daten in Text zu konvertieren.

Was passiert, wenn man sie nicht verwendet?

- Man müsste umständliche Methoden verwenden.

Beispiel:

```
#include <sstream>
stringstream ss;ss << 42;
string text = ss.str(); // Umwandlung in String
```

65. Prinzipien abstrakter Klassen

Fachbegriffe:

- **Reine virtuelle Methoden:** Erzwingen eine Implementierung in abgeleiteten Klassen.
- **Schnittstellen:** Ermöglichen Polymorphie und klare Struktur.

Warum verwendet man sie?

- Um abstrakte Basisklassen zu definieren, die keine eigenständigen Objekte sein sollen.

Was passiert, wenn man sie nicht verwendet?

- Es könnte schwierig sein, einheitliche Schnittstellen für Polymorphie zu schaffen.

Beispiel:

```
class Fahrzeug {
public:
    virtual void fahren() = 0; // Abstrakte Methode
};
```

66. Ausnahmebehandlung mit Hierarchien

Fachbegriffe:

- **Ausnahme-Hierarchie:** Klassenstruktur für spezifische Fehlerarten.
- **Polymorphe Behandlung:** Erlaubt das Fangen allgemeiner oder spezifischer Fehler.

Warum verwendet man sie?

- Um Fehler übersichtlich und differenziert zu behandeln.

Was passiert, wenn man sie nicht verwendet?

- Fehlerbehandlung wäre unstrukturiert.

Beispiel:

```
try {
    throw runtime_error("Fehler");
} catch (const runtime_error& e) {
    cout << "Runtime Fehler: " << e.what();
}
```

67. Ausnahmeobjekte weitergeben

Fachbegriffe:

- **Weiterwerfen von Ausnahmen:** Fehler können an den nächsten `catch`-Block weitergereicht werden.

Warum verwendet man es?

- Um Fehlerbehandlung zwischen verschiedenen Programmteilen zu delegieren.

Was passiert, wenn man es nicht verwendet?

- Fehler müssen lokal behandelt werden, was den Code komplizierter macht.

Beispiel:

```
try {
    throw runtime_error("Fehler");
} catch (const exception& e) {
    throw; // Weitergeben
}
```

68. Ausnahmebehandlung und `noexcept`

Fachbegriffe:

- **`noexcept`:** Kennzeichnet, dass eine Funktion keine Ausnahme werfen kann.

Warum verwendet man es?

- Verbessert die Stabilität und Effizienz des Codes.

Was passiert, wenn man es nicht verwendet?

- Der Compiler kann keine Optimierungen für Ausnahmesicherheit vornehmen.

Beispiel:

```
void funktion() noexcept {
    // Garantiert ausnahmesicher
}
```

69. Ausnahmebehandlung ohne Overhead

Fachbegriffe:

- **Effizienz der Ausnahmebehandlung:** Nur bei tatsächlichem Fehler wird Overhead erzeugt.

Warum ist das wichtig?

- Schnelle Programme trotz robuster Fehlerbehandlung.

Was passiert, wenn man sie nicht verwendet?

- Fehlerbehandlung könnte den Programmfluss beeinträchtigen.

Beispiel:

```
try {
    // Kein Overhead ohne Fehler
} catch (...) {
    // Ausnahmebehandlung
}
```

70. Container und Iteratoren

Fachbegriffe:

- **Container:** Datenstrukturen wie `vector`, `list`, `map`.
- **Iteratoren:** Ermöglichen Traversieren von Containern.

Warum verwendet man sie?

- Um Daten effizient zu speichern und zu durchsuchen.

Was passiert, wenn man sie nicht verwendet?

- Man müsste manuell Datenstrukturen und Traversierungslogik implementieren.

Beispiel:

```
#include <vector>
vector<int> v = {1, 2, 3};
for (auto it = v.begin(); it != v.end(); ++it) {
    cout << *it;
}
```

71. Modularisierung und Wiederverwendbarkeit

Fachbegriffe:

- **Module:** Unterteilung in überschaubare Dateien und Bibliotheken.
- **Wiederverwendung:** Förderung durch saubere Trennung und klare Schnittstellen.

Warum verwendet man sie?

- Erleichtert Wartung und Erweiterung.

Was passiert, wenn man sie nicht verwendet?

- Der Code wird unübersichtlich und schwerer wiederverwendbar.

Beispiel:

```
// Modul Fahrzeug.h
class Fahrzeug {
    string typ;
};
```

72. Templates und Typunabhängigkeit

Fachbegriffe:

- **Template:** Generiert Code für beliebige Datentypen.
- **Instanziierung:** Erstellung einer spezifischen Version des Templates.

Warum verwendet man sie?

- Um redundanten Code zu vermeiden und Flexibilität zu bieten.

Was passiert, wenn man sie nicht verwendet?

- Man müsste für jeden Typ eigene Funktionen oder Klassen schreiben.

Beispiel:

```
template <typename T>
T add(T a, T b) { return a + b; }
```

73. Vererbung und Polymorphie

Fachbegriffe:

- **Polymorphie:** Ermöglicht, dass abgeleitete Klassen Basisklassenmethoden überschreiben.
- **Vererbung:** Wiederverwendbarkeit von Code durch Erweiterung.

Warum verwendet man sie?

- Fördert sauberen, erweiterbaren Code.

Was passiert, wenn man sie nicht verwendet?

- Redundante Code-Duplikationen.

Beispiel:

```
class Fahrzeug {
    virtual void fahren() {}
};
class Auto : public Fahrzeug {
    void fahren() override {}
};
```

74. Exception Handling und Stabilität

Fachbegriffe:

- **Exception Safety:** Sicherstellung, dass ein Programm bei Fehlern stabil bleibt.
- **Stack Unwinding:** Aufräumen von Ressourcen bei Fehlern.

Warum verwendet man sie?

- Um Programme trotz Fehlern stabil und berechenbar zu halten.

Was passiert, wenn man sie nicht verwendet?

- Ressourcen könnten verloren gehen, oder das Programm könnte abstürzen.

Beispiel:

```
try {
    throw exception();
} catch (...) {
    cout << "Fehler aufgefangen";
}
```

75. Statische Methoden

Fachbegriffe:

- **Statische Methoden:** Methoden, die zur Klasse gehören und nicht zu einer bestimmten Instanz. Sie werden mit dem Schlüsselwort `static` deklariert.
- **Aufruf:** Sie können direkt über den Klassennamen aufgerufen werden, ohne ein Objekt zu erstellen.

Warum verwendet man sie?

- Um Funktionen zu implementieren, die keinen Zugriff auf die Instanzvariablen benötigen.
- Ideal für Hilfsfunktionen, Zähler oder andere Aufgaben, die unabhängig von Objekten sind.

Was passiert, wenn man sie nicht verwendet?

- Funktionen, die nicht auf Instanzdaten angewiesen sind, müssten trotzdem in einer Instanz definiert werden, was unnötigen Speicherverbrauch verursacht und die Semantik des Codes verschlechtert.

Beispiel:

```
class Mathe {
public:
    static int addiere(int a, int b) {
        return a + b; // Keine Instanzdaten nötig
    }
};

// Aufruf der statischen Methode
int ergebnis = Mathe::addiere(5, 10);
```