Vorlesungszusammenfassung

Internationale Hochschule Duales Studium

Studiengang: Informatik

Klausurvorbereitung C/C++

Patryk Hegenberg

Matrikelnummer: 102209025

Dedendorf 8

27333 Bücken

Betreuende Person: Frank Krickel

Abgabedatum: keins

Inhaltsverzeichnis

Inhaltsverzeichnis I

Tabellenverzeichnis IV

Abbildungsverzeichnis IV

1. Typen und Variablen 1

1.1. Elementare Datentypen: 1

1.2. Vereinheitlichte Initialisierungssyntax: 1

1.3. Automatische Typ-Deduktion mit „auto“: 1

1.4. Lokale Variablen: 2

1.5. Globale Variablen: 3

1.6. Konstanten: 5

1.7. Operatoren 6

1.7.1. Arithmetische Operatoren: 6

1.7.2. Zuweisungsoperatoren: 7

1.7.3. Inkrement- und Dekrement-Operatoren: 7

1.7.4. Vergleichsoperatoren: 8

1.7.5. Logische Operatoren: 8

1.8. if-else-Anweisungen: 9

1.9. switch-Anweisung: 11

1.10. for-Schleife: 12

1.11. while-Schleife: 14

1.12. do-while-Schleife: 16

1.13. Zusammenfassung 18

1.13.1. Elementare Datentypen: 18

1.13.2. Vereinheitlichte Initialisierungssyntax: 18

1.13.3. Typen & Variablen: 18

1.13.4. Automatische Typ-Deduktion mit „auto“: 18

1.13.5. Lokale Variablen: 18

1.13.6. Globale Variablen: 18

1.13.7. Konstanten: 19

1.13.8. Operatoren: 19
1.13.9. if-else-Anweisungen: 19
1.13.10. switch-Anweisung: 19
1.13.11. for-Schleife: 20
1.13.12. while-Schleife: 20
1.13.13. do-while-Schleife: 20
1.14. Fragen 21
1.14.1. Multiple-Choice 21
1.14.2. Erklärungen 23
2. Ein. und Ausgabe 25
2.1. Eingabe mit cin: 25
2.2. Ausgabe mit cout: 26
2.3. Fehlschläge bei der Eingabe: 27
3. Strings 28
3.1. String-Länge: 28
3.2. String-Verkettungen: 28
3.3. String-Wandlungen: 29
3.4. String-Substring: 29
3.5. Weitere nützliche Funktionen: 30
3.6. Standard-Konstruktor: 31
3.7. Konstruktor mit C-String: 31
3.8. Kopier-Konstruktor: 31
3.9. Konstruktor mit Zeichen und Länge: 32
3.10. Initialisierung mit einem Zeichen: 32
3.11. Konstruktor mit Iteratoren: 33
4. Vektoren 33
4.1. Merkmale 33
4.2. Vektor erstellen: 34
4.3. Elemente hinzufügen: 34
4.4. Elemente zugreifen: 34

4.5. Vektor-Größe: 35
4.6. Vektor leeren: 35
4.7. Vektor durchlaufen: 36
4.8. Vektor mit Initialisierungsliste erstellen: 36
4.9. Weitere nützliche Funktionen: 37
4.9.1. empty(): 37
4.9.2. pop_back(): 37
4.9.3. insert(): 37
4.9.4. erase(): 38
4.9.5. resize(): 38
4.9.6. swap(): 38
5. Zusammenfassung 39
5.1. Ein- und Ausgabe in C++: 39
5.1.1. Ausgabe mit std::cout: 39
5.1.2. Eingabe mit std::cin: 39
5.2. Strings in C++: 39
5.2.1. String erstellen: 39
5.2.2. String-Verkettung: 40
5.2.3. String-Länge: 40
5.2.4. String-Eingabe: 40
5.3. Vektoren 41
5.3.1. Vektor-Definition: 41
5.3.2. Vektor erstellen: 41
5.3.3. Elemente hinzufügen: 41
5.3.4. Elemente zugreifen: 41
5.3.5. Vektor-Größe: 41
5.3.6. Vektor leeren: 41
5.3.7. Vektor durchlaufen: 42
5.3.8. Vektor mit Initialisierungsliste erstellen: 42
5.3.9. Weitere nützliche Funktionen: 42

6. Fragen 42
6.1. Multiple-Choice Fragen 42
6.2. Erklärung 45
6.2.1. Antworten 45
7. Container und Iteratoren 49
7.1. C-ARRAYS VS. VECTOR UND ANDERE ELEMENTE DER STL 49
7.1.1. C-Arrays: 49
7.1.2. Vektoren (std::vector): 50
7.1.3. Andere Elemente der STL: 51
7.2. Iteratoren in C++ 52
7.2.1. Iteratoren in Vektoren: 52
7.2.2. Iteratoren in Listen: 53
7.2.3. Iteratoren in Maps: 54
7.3. Listen in C++ 55
7.3.1. Erstellen einer Liste: 55
7.3.2. Einfügen von Elementen in eine Liste: 56
7.3.3. Löschen von Elementen aus einer Liste: 57
7.3.4. Suchen in einer Liste: 57
7.4. Arrays aus der STL (std::array) 58
7.4.1. Erstellen eines std::array: 58
7.4.2. Zugriff auf Elemente eines std::array: 59
7.4.3. Größe eines std::array: 59
7.5. Sets in C++ 60
7.5.1. Erstellen eines Sets: 60
7.5.2. Einfügen von Elementen in ein Set: 61
7.5.3. Löschen von Elementen aus einem Set: 61
7.5.4. Suchen in einem Set: 62
Literaturverzeichnis LXIV

Tabellenverzeichnis

Abbildungsverzeichnis

1. Typen und Variablen

1.1. Elementare Datentypen:

In C++ gibt es verschiedene elementare Datentypen, mit denen du Variablen deklarieren kannst, um verschiedene Arten von Daten zu speichern. Hier sind die häufigsten elementaren Datentypen in C++:

- int: Ganzzahliger Datentyp, der ganze Zahlen speichert, z. B. 1, -5, 100, usw.
- float und double: Gleitkommazahlen, die Dezimalzahlen mit Fließkomma repräsentieren. float speichert eine kleinere Genauigkeit als double.
- char: Ein einzelnes Zeichen, z. B. ‚A', ‚b', ‚1', ‚?', usw.
- bool: Boolscher Datentyp, der entweder true oder false speichert.

Beispiel:

```
int age = 25;
float pi = 3.14;
char grade = 'A';
bool isPassed = true;
```

1.2. Vereinheitlichte Initialisierungssyntax:

C++11 führte eine vereinheitlichte Initialisierungssyntax ein, die es dir ermöglicht, Variablen zu initialisieren, indem du geschweifte Klammern {} verwendest. Dies funktioniert für alle Datentypen und ermöglicht auch das Vermeiden von unbeabsichtigten Typumwandlungen (Type Conversions).

Beispiel:

```
int a{ 10 }; // Initialisierung eines int mit 10
float b{ 3.14 }; // Initialisierung eines float mit 3.14
char c{ 'C' }; // Initialisierung eines char mit 'C'
bool d{ true }; // Initialisierung eines bool mit true
```

Diese vereinheitlichte Initialisierungssyntax ist besonders nützlich, wenn du mit benutzerdefinierten Datentypen oder Containern arbeitest.

1.3. Automatische Typ-Deduktion mit „auto":

C++11 führte das Schlüsselwort auto ein, das es dem Compiler ermöglicht, den Datentyp einer Variablen automatisch aus ihrem Initialisierungswert abzuleiten. Dies kann den Code kürzer und lesbarer machen, insbesondere wenn komplexe Typen involviert sind.

Beispiel:

```
auto x = 42; // x wird automatisch als int erkannt
auto y = 3.14; // y wird automatisch als double erkannt
auto name = "John"; // name wird als const char* erkannt
```

Bei Verwendung von auto beachte bitte folgende Punkte:

- auto kann nur für lokale Variablen verwendet werden, die bei der Initialisierung einen Wert erhalten.
- auto kann in Funktionen mit Rückgabetypen verwendet werden, aber nicht für Funktionen mit Parametern.
- Bei der Verwendung von auto wird der Datentyp statisch zur Compilezeit ermittelt und kann sich während der Lebensdauer der Variablen nicht ändern.

Die automatische Typ-Deduktion ist besonders nützlich, wenn du mit komplexen Datentypen arbeitest, wie zum Beispiel bei der Verwendung von Iteratoren oder komplexen Container-Typen.

Beispiel:

```
#include <vector>
std::vector<int> numbers = {1, 2, 3, 4, 5};
for (auto it = numbers.begin(); it != numbers.end(); ++it) {
    // Hier ist der Typ von "it" ein "std::vector<int>::iterator"
    // Aber wir müssen es nicht manuell angeben, da wir "auto" verwenden.
}
```

1.4. Lokale Variablen:

Lokale Variablen sind Variablen, die innerhalb eines bestimmten Gültigkeitsbereichs (normalerweise innerhalb einer Funktion oder eines Blocks) deklariert werden und nur innerhalb dieses Gültigkeitsbereichs sichtbar und zugänglich sind. Sobald der Gültigkeitsbereich verlassen wird, werden die lokalen Variablen automatisch zerstört, und der von ihnen belegte Speicher wird freigegeben.

Beispiel:

```
#include <iostream>

void exampleFunction() {
    int x = 10; // x ist eine lokale Variable
```

```
    std::cout << "The value of x: " << x << std::endl;
} // x wird am Ende der Funktion zerstört
```

In diesem Beispiel ist x eine lokale Variable, die innerhalb der Funktion exampleFunction() deklariert wurde. Sobald die Funktion beendet ist, wird x automatisch zerstört und der Speicher wird freigegeben.
Lokale Variablen bieten verschiedene Vorteile:

- Begrenzte Sichtbarkeit: Lokale Variablen sind nur innerhalb ihres Gültigkeitsbereichs sichtbar, was dazu beiträgt, Namenskonflikte zu vermeiden und den Code klarer zu machen.
- Ressourcenmanagement: Durch die automatische Zerstörung lokaler Variablen am Ende ihres Gültigkeitsbereichs wird eine effiziente Nutzung von Speicher und Ressourcen gewährleistet.
- Speichersicherheit: Da lokale Variablen innerhalb des Stapels (Stacks) des Programms gespeichert werden, sind sie normalerweise effizienter und sicherer als globale Variablen.

```
#include <iostream>

int main() {
    int a = 5; // Lokale Variable innerhalb der main-Funktion
    if (a > 0) {
        int b = 10; // Lokale Variable innerhalb des if-Blocks
        std::cout << "a is positive." << std::endl;
        std::cout << "b is: " << b << std::endl;
    } // b wird am Ende des if-Blocks zerstört
    // std::cout << b; // Dies würde zu einem Fehler führen, da b außerhalb seines
Gültigkeitsbereichs ist
    return 0;
} // a wird am Ende der main-Funktion zerstört
```

In diesem Beispiel gibt es zwei lokale Variablen, a und b. a ist innerhalb der gesamten main()-Funktion sichtbar, während b nur innerhalb des if-Blocks sichtbar ist.

1.5. Globale Variablen:

Globale Variablen sind Variablen, die außerhalb aller Funktionen und Blöcke im globalen Gültigkeitsbereich deklariert werden. Das bedeutet, dass sie in jedem Teil des Codes, einschließlich

Funktionen, sichtbar und zugänglich sind. Globale Variablen behalten ihren Wert über den gesamten Lebenszyklus des Programms bei, solange das Programm läuft.
Beispiel:

```
#include <iostream>

int globalVar = 100; // globale Variable

void function1() {
    std::cout << "globalVar from function1: " << globalVar << std::endl;
}

void function2() {
    globalVar = 200; // Zugriff und Änderung einer globalen Variablen
}

int main() {
    std::cout << "globalVar from main: " << globalVar << std::endl;
    function1();
    function2();
    std::cout << "globalVar after function2: " << globalVar << std::endl;
    return 0;
}
```

In diesem Beispiel wird globalVar außerhalb der main()-Funktion deklariert und hat somit globalen Gültigkeitsbereich. Die Funktionen function1() und function2() können auf globalVar zugreifen und diese auch ändern.
Globale Variablen bieten einige Vorteile, wie z.B.:

- Einfacher Zugriff: Globale Variablen sind überall im Code zugänglich, was den Zugriff auf gemeinsam genutzte Daten vereinfachen kann.
- Globale Konfiguration: Sie können verwendet werden, um Konfigurationsdaten oder Einstellungen zu speichern, die von verschiedenen Teilen des Programms verwendet werden.

Jedoch haben globale Variablen auch einige Nachteile:

- Namenskonflikte: Da globale Variablen überall sichtbar sind, besteht ein erhöhtes Risiko von Namenskonflikten, insbesondere in großen Projekten.

- Unerwartetes Verhalten: Änderungen an globalen Variablen können unerwartetes Verhalten in verschiedenen Teilen des Codes verursachen, insbesondere wenn mehrere Threads im Spiel sind.
- Schwierigeres Debugging: Globale Variablen können das Debugging erschweren, da es schwerer sein kann, ihre Werte und Zustände nachzuvollziehen.

Es ist daher oft ratsam, den Einsatz globaler Variablen auf das Notwendige zu beschränken und stattdessen den Gebrauch von lokalen Variablen zu bevorzugen, wann immer es möglich ist.

1.6. Konstanten:

Konstanten sind Variablen, deren Wert während der Ausführung des Programms nicht geändert werden kann. In C++, können Konstanten auf zwei Arten deklariert werden: mit dem Schlüsselwort const oder mit dem Makro `#define`.

- Konstanten mit dem Schlüsselwort const:
  - Mit dem Schlüsselwort const kannst du Variablen deklarieren, die einen festen Wert haben und nicht verändert werden können. Der Compiler stellt sicher, dass keine Änderungen an diesen Variablen vorgenommen werden.

Beispiel:

```
const int num = 10; // num ist eine Konstante mit dem Wert 10
```

- Konstanten mit dem Makro `#define`:
  - Du kannst auch Konstanten mit dem Präprozessor-Makro `#define` deklarieren. Diese Art der Konstantendeklaration ist jedoch weniger flexibel und kann zu Problemen führen, da der Präprozessor einfach den Text ersetzt, ohne Rücksicht auf den Datentyp.

Beispiel:

```
#define PI 3.14 // PI ist eine Konstante mit dem Wert 3.14
```

Es wird jedoch empfohlen, das Schlüsselwort const zu verwenden, da es sicherer ist und den Datentyp der Konstante berücksichtigt.

Vorteile von Konstanten:

- Sicherheit: Konstanten schützen die Daten vor versehentlichen Änderungen und erhöhen die Sicherheit des Codes.

- Klarheit: Durch die Verwendung von Konstanten statt „magischen Zahlen“ im Code wird der Code lesbarer und klarer.
- Optimierung: Der Compiler kann Konstanten besser optimieren, da er ihre Werte kennt und sie nicht ändern kann.

Beispiel:

```
#include <iostream>

int main() {
    const int num = 42;
    // num = 50; // Fehler! Konstanten können nicht geändert werden.
    std::cout << "The value of num: " << num << std::endl;
    return 0;
}
```

In diesem Beispiel ist num eine Konstante mit dem Wert 42. Wenn du versuchst, den Wert von num zu ändern, wird ein Fehler gemeldet.

1.7. Operatoren

1.7.1. Arithmetische Operatoren:

Arithmetische Operatoren werden verwendet, um mathematische Berechnungen auf Zahlen durchzuführen. Hier sind die wichtigsten arithmetischen Operatoren:

- `+` (Addition): Führt eine Addition von zwei Zahlen durch.
- `-` (Subtraktion): Führt eine Subtraktion von zwei Zahlen durch.
- `*` (Multiplikation): Führt eine Multiplikation von zwei Zahlen durch.
- `/` (Division): Führt eine Division von zwei Zahlen durch.
- `%` (Modulo): Berechnet den Rest einer Division von zwei Zahlen.

Beispiel:

```
int a = 10, b = 5;
int sum = a + b; // sum ist 15
int difference = a - b; // difference ist 5
int product = a * b; // product ist 50
int quotient = a / b; // quotient ist 2
int remainder = a % b; // remainder ist 0
```

1.7.2. Zuweisungsoperatoren:

Zuweisungsoperatoren werden verwendet, um Werte einer Variablen zuzuweisen oder zu aktualisieren. Sie bieten eine kompakte Möglichkeit, arithmetische Operationen mit Zuweisungen zu kombinieren.

- `=` (Zuweisung): Weist einer Variablen einen Wert zu.
- `+=` (Addition und Zuweisung): Addiert den rechten Ausdruck zum Wert der Variablen und weist das Ergebnis der Variablen zu.
- `-=` (Subtraktion und Zuweisung): Subtrahiert den rechten Ausdruck vom Wert der Variablen und weist das Ergebnis der Variablen zu.
- `*=` (Multiplikation und Zuweisung): Multipliziert den Wert der Variablen mit dem rechten Ausdruck und weist das Ergebnis der Variablen zu.
- `/=` (Division und Zuweisung): Dividiert den Wert der Variablen durch den rechten Ausdruck und weist das Ergebnis der Variablen zu.
- `%=` (Modulo und Zuweisung): Berechnet den Modulo des Wertes der Variablen und des rechten Ausdrucks und weist das Ergebnis der Variablen zu.

Beispiel:

```
int x = 10;
x += 5; // x ist jetzt 15
x -= 3; // x ist jetzt 12
x *= 2; // x ist jetzt 24
x /= 4; // x ist jetzt 6
x %= 5; // x ist jetzt 1
```

1.7.3. Inkrement- und Dekrement-Operatoren:

Inkrement- und Dekrement-Operatoren werden verwendet, um den Wert einer Variablen um 1 zu erhöhen oder zu verringern.

- `++` (Inkrement): Erhöht den Wert einer Variablen um 1.
- `--` (Dekrement): Verringert den Wert einer Variablen um 1.

Die Inkrement- und Dekrement-Operatoren können als Präfix (++x, –x) oder als Suffix (x++, x–) verwendet werden. Der Unterschied besteht darin, wann der Wert der Variablen geändert wird. Bei der Verwendung als Präfix wird der Wert zuerst erhöht oder verringert und dann im Ausdruck verwendet. Bei der Verwendung als Suffix wird der aktuelle Wert der Variablen im Ausdruck verwendet und dann erst erhöht oder verringert.

Beispiel:

```
int i = 5;
int j = ++i; // i ist jetzt 6, j ist 6
int k = i--; // i ist jetzt 5, k ist 6
```

1.7.4. Vergleichsoperatoren:

Vergleichsoperatoren werden verwendet, um Werte zu vergleichen und einen booleschen Ausdruck (true oder false) zurückzugeben.

- == (Gleich): Überprüft, ob zwei Werte gleich sind.
- != (Ungleich): Überprüft, ob zwei Werte ungleich sind.
- > (Größer als): Überprüft, ob der linke Wert größer ist als der rechte Wert.
- < (Kleiner als): Überprüft, ob der linke Wert kleiner ist als der rechte Wert.
- >= (Größer oder gleich): Überprüft, ob der linke Wert größer oder gleich dem rechten Wert ist.
- <= (Kleiner oder gleich): Überprüft, ob der linke Wert kleiner oder gleich dem rechten Wert ist.

Beispiel:

```
int a = 5, b = 10;
bool isEqual = (a == b); // isEqual ist false
bool isNotEqual = (a != b); // isNotEqual ist true
bool isGreater = (a > b); // isGreater ist false
bool isLess = (a < b); // isLess ist true
bool isGreaterOrEqual = (a >= b); // isGreaterOrEqual ist false
bool isLessOrEqual = (a <= b); // isLessOrEqual ist true
```

1.7.5. Logische Operatoren:

Logische Operatoren werden verwendet, um logische Ausdrücke zu kombinieren und zu verknüpfen.

- && (Logisches UND): Der Ausdruck ist wahr, wenn beide Operanden wahr sind.
- || (Logisches ODER): Der Ausdruck ist wahr, wenn mindestens einer der Operanden wahr ist.
- ! (Logisches NICHT): Invertiert den Wert eines Ausdrucks. Wenn der Ausdruck wahr ist, wird er zu falsch, und wenn der Ausdruck falsch ist, wird er zu wahr.

Beispiel:

```
bool condition1 = true, condition2 = false;
bool result1 = condition1 && condition2; // result1 ist false
bool result2 = condition1 || condition2; // result2 ist true
bool result3 = !condition1; // result3 ist false
```

1.8. if-else-Anweisungen:

if-else-Anweisungen ermöglichen die Ausführung von Codeblöcken basierend auf einer Bedingung. Die Bedingung wird ausgewertet, und je nachdem, ob sie wahr oder falsch ist, wird der entsprechende Codeblock ausgeführt.

Die grundlegende Syntax einer if-else-Anweisung lautet:

```
if (Bedingung) {
    // Codeblock, der ausgeführt wird, wenn die Bedingung wahr ist
} else {
    // Codeblock, der ausgeführt wird, wenn die Bedingung falsch ist
}
```

Beispiel:

```
#include <iostream>

int main() {
    int num = 10;
    if (num > 5) {
        std::cout << "Die Zahl ist größer als 5." << std::endl;
    } else {
        std::cout << "Die Zahl ist nicht größer als 5." << std::endl;
    }
    return 0;
}
```

In diesem Beispiel wird überprüft, ob num größer als 5 ist. Wenn die Bedingung wahr ist, wird der Code im ersten Codeblock ausgeführt („Die Zahl ist größer als 5.“). Andernfalls wird der Code im zweiten Codeblock ausgeführt („Die Zahl ist nicht größer als 5.“).

Du kannst auch if-Anweisungen ohne den else-Teil verwenden:

```
if (Bedingung) {
    // Codeblock, der ausgeführt wird, wenn die Bedingung wahr ist
}
```

Mehrfachverzweigungen können mit der else if-Anweisung realisiert werden:

```
if (Bedingung1) {
    // Codeblock, der ausgeführt wird, wenn Bedingung1 wahr ist
} else if (Bedingung2) {
    // Codeblock, der ausgeführt wird, wenn Bedingung1 falsch und Bedingung2 wahr ist
} else {
    // Codeblock, der ausgeführt wird, wenn keine der vorherigen Bedingungen wahr ist
}
```

Beispiel:

```
#include <iostream>

int main() {
    int num = 10;
    if (num > 0) {
        std::cout << "Die Zahl ist positiv." << std::endl;
    } else if (num < 0) {
        std::cout << "Die Zahl ist negativ." << std::endl;
    } else {
        std::cout << "Die Zahl ist null." << std::endl;
    }
    return 0;
}
```

In diesem Beispiel wird überprüft, ob num positiv, negativ oder null ist und entsprechend eine Meldung ausgegeben.
if-else-Anweisungen sind eine grundlegende Kontrollstruktur in C++, die es ermöglicht, den Programmfluss basierend auf bestimmten Bedingungen zu steuern. Du kannst auch geschachtelte if-else-Anweisungen erstellen, um komplexere Logik zu implementieren. Es ist jedoch wichtig, die Klammern sorgfältig zu setzen, um den gewünschten Codeblock richtig zu definieren.

1.9. switch-Anweisung:

Die switch-Anweisung ermöglicht die Auswahl zwischen mehreren möglichen Werten einer Variablen und führt den entsprechenden Codeblock aus, der mit dem gewählten Wert verknüpft ist. Dies ermöglicht eine kompakte und übersichtliche Möglichkeit, den Programmfluss basierend auf verschiedenen Bedingungen zu steuern.

Die grundlegende Syntax einer switch-Anweisung lautet:

```
switch (Ausdruck) {
    case Wert1:
        // Codeblock, der ausgeführt wird, wenn der Ausdruck den Wert Wert1 hat
        break;
    case Wert2:
        // Codeblock, der ausgeführt wird, wenn der Ausdruck den Wert Wert2 hat
        break;
    // Weitere case-Blöcke für andere Werte
    default:
        // Codeblock, der ausgeführt wird, wenn der Ausdruck keinen der vorherigen Werte
hat
        break;
}
```

Der switch-Ausdruck kann nur Ganzzahlen (integers) und Zeichen (char) sein. Gleitkommazahlen oder Strings können nicht in einem switch verwendet werden.

Beispiel:

```
#include <iostream>

int main() {
    int choice;
    std::cout << "Wähle eine Option: 1 (Eins), 2 (Zwei), 3 (Drei): ";
    std::cin >> choice;

    switch (choice) {
        case 1:
            std::cout << "Du hast Eins gewählt." << std::endl;
            break;
        case 2:
```

```
            std::cout << "Du hast Zwei gewählt." << std::endl;
            break;
        case 3:
            std::cout << "Du hast Drei gewählt." << std::endl;
            break;
        default:
            std::cout << "Ungültige Auswahl." << std::endl;
            break;
    }

    return 0;
}
```

In diesem Beispiel wird der Benutzer aufgefordert, eine Zahl einzugeben. Die switch-Anweisung prüft den Wert von choice und führt den entsprechenden Codeblock aus, abhängig davon, welchen Wert choice hat.

Beachte, dass jeder case-Block mit dem passenden Wert verglichen wird. Sobald ein passender Wert gefunden wurde, wird der zugehörige Codeblock ausgeführt, und die switch-Anweisung wird mit break beendet. Ohne das break würde die Ausführung fortgesetzt und auch die nachfolgenden case-Blöcke ausgeführt, was unter Umständen unerwünschtes Verhalten verursachen könnte.

Die default-Klausel ist optional, aber oft nützlich, um sicherzustellen, dass die switch-Anweisung immer eine Aktion ausführt, selbst wenn keine Übereinstimmung gefunden wurde.

Die switch-Anweisung ist besonders nützlich, wenn du mehrere Auswahlmöglichkeiten hast und den Code übersichtlich halten möchtest. Sie bietet eine gute Alternative zu verschachtelten if-else-Anweisungen, insbesondere wenn die Bedingungen auf einfache Ganzzahlvergleiche beschränkt sind.

1.10. for-Schleife:

Die for-Schleife ist eine der grundlegenden Schleifenstrukturen in C++, die verwendet wird, um einen Codeblock mehrmals auszuführen. Sie ist besonders nützlich, wenn du die Anzahl der Schleifendurchläufe im Voraus kennst oder einen bestimmten Bereich durchlaufen möchtest.

Die Syntax einer for-Schleife lautet:

```
for (Initialisierung; Bedingung; Inkrement) {
    // Codeblock, der wiederholt ausgeführt wird, solange die Bedingung wahr ist
}
```

- Initialisierung: Hier kannst du eine Variable initialisieren, die in der Schleife verwendet wird. Es wird normalerweise einmal ausgeführt, bevor die Schleife beginnt.
- Bedingung: Dies ist der Ausdruck, der bei jedem Schleifendurchlauf überprüft wird. Solange die Bedingung wahr ist, wird der Codeblock ausgeführt. Wenn die Bedingung falsch wird, endet die Schleife.
- Inkrement: Hier kannst du die Variable ändern oder inkrementieren, die in der Schleife verwendet wird. Es wird nach jedem Schleifendurchlauf ausgeführt.

Beispiel:

```
#include <iostream>

int main() {
    for (int i = 1; i <= 5; i++) {
        std::cout << "Schleifendurchlauf #" << i << std::endl;
    }

    return 0;
}
```

In diesem Beispiel wird die for-Schleife fünfmal durchlaufen, da die Bedingung i <= 5 erfüllt ist. Bei jedem Durchlauf wird der Wert von i ausgegeben.

Die for-Schleife ist auch nützlich, um durch Container (z. B. Arrays oder Vektoren) oder über eine Folge von Zahlen zu iterieren:

Beispiel - Durchlaufen eines Arrays:

```
#include <iostream>

int main() {
    int myArray[] = {1, 2, 3, 4, 5};

    for (int i = 0; i < 5; i++) {
        std::cout << "Element #" << i << ": " << myArray[i] << std::endl;
```

```
    }

    return 0;
}
```

In diesem Beispiel wird die for-Schleife verwendet, um alle Elemente des Arrays myArray zu durchlaufen und ihre Werte auszugeben.

Beispiel - Durchlaufen einer Zahlenfolge:

```
#include <iostream>

int main() {
    for (int i = 10; i >= 1; i--) {
        std::cout << i << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

In diesem Beispiel wird die for-Schleife verwendet, um die Zahlen von 10 bis 1 absteigend auszugeben.

Die for-Schleife bietet eine kompakte Möglichkeit, Codeblöcke zu wiederholen, und ist besonders nützlich, wenn du die Schleifenzähler kontrollieren möchtest.

1.11. while-Schleife:

Die while-Schleife ist eine weitere Schleifenstruktur in C++, die verwendet wird, um einen Codeblock so lange auszuführen, wie eine bestimmte Bedingung wahr ist. Im Gegensatz zur for-Schleife wird die while-Schleife verwendet, wenn du die Anzahl der Schleifendurchläufe im Voraus nicht genau kennst, sondern nur eine Bedingung hast, die überprüft wird, bevor jeder Schleifendurchlauf erfolgt.

Die Syntax einer while-Schleife lautet:

```
while (Bedingung) {
    // Codeblock, der wiederholt ausgeführt wird, solange die Bedingung wahr ist
}
```

Die Schleife beginnt damit, dass die Bedingung ausgewertet wird. Wenn die Bedingung wahr ist, wird der Codeblock ausgeführt. Nachdem der Codeblock ausgeführt wurde, wird die Bedingung erneut überprüft, und wenn sie immer noch wahr ist, wird der Codeblock erneut ausgeführt. Dieser Vorgang wird so lange wiederholt, bis die Bedingung falsch ist, dann endet die Schleife, und die Programmausführung setzt fort.

Beispiel:

```
#include <iostream>

int main() {
    int count = 1;

    while (count <= 5) {
        std::cout << "Schleifendurchlauf #" << count << std::endl;
        count++; // Inkrementiere den Zähler für den nächsten Schleifendurchlauf
    }

    return 0;
}
```

In diesem Beispiel wird die while-Schleife fünfmal durchlaufen, da die Bedingung count <= 5 erfüllt ist. Bei jedem Durchlauf wird der Wert von count ausgegeben und der Zähler inkrementiert.

Die while-Schleife ist auch nützlich, wenn du auf eine Benutzereingabe oder ein bestimmtes Ereignis wartest, um die Schleife zu beenden:

Beispiel - Benutzereingabe einlesen:

```
#include <iostream>

int main() {
    char input;

    std::cout << "Drücke 'q' und dann Enter, um die Schleife zu beenden." << std::endl;

    while (std::cin >> input) {
        if (input == 'q') {
```

```
            break; // Beende die Schleife, wenn 'q' eingegeben wurde
        }
    }

    std::cout << "Schleife beendet." << std::endl;
    return 0;
}
```

In diesem Beispiel wartet die while-Schleife darauf, dass der Benutzer eine Eingabe macht. Wenn der Benutzer ‚q' eingibt und die Enter-Taste drückt, wird die Schleife mit break beendet. Die while-Schleife ist eine vielseitige Schleifenstruktur, die besonders nützlich ist, wenn du eine Schleife ausführen möchtest, solange eine bestimmte Bedingung erfüllt ist, aber du die Anzahl der Schleifendurchläufe nicht genau kennst. Sie bietet Flexibilität und Kontrolle über den Schleifenablauf.

1.12. do-while-Schleife:

Die do-while-Schleife ist eine weitere Schleifenstruktur in C++, die ähnlich wie die while-Schleife funktioniert. Der Hauptunterschied besteht darin, dass der Codeblock zuerst einmal ausgeführt wird, bevor die Bedingung überprüft wird. Dies bedeutet, dass der Codeblock mindestens einmal ausgeführt wird, auch wenn die Bedingung von Anfang an falsch ist.
Die Syntax einer do-while-Schleife lautet:

```
do {
    // Codeblock, der mindestens einmal ausgeführt wird
} while (Bedingung);
```

Der Codeblock wird zuerst ausgeführt, dann wird die Bedingung überprüft. Solange die Bedingung wahr ist, wird der Codeblock wiederholt ausgeführt. Die Schleife endet, wenn die Bedingung falsch ist.
Beispiel:

```
#include <iostream>

int main() {
    int count = 1;

    do {
```

```
        std::cout << "Schleifendurchlauf #" << count << std::endl;
        count++; // Inkrementiere den Zähler für den nächsten Schleifendurchlauf
    } while (count <= 5);

    return 0;
}
```

In diesem Beispiel wird die do-while-Schleife fünfmal durchlaufen, da die Bedingung count <= 5 am Ende jedes Durchlaufs überprüft wird. Der Codeblock wird zuerst einmal ausgeführt, bevor die Bedingung überprüft wird.

Im Vergleich zur while-Schleife ist die do-while-Schleife nützlich, wenn du sicherstellen möchtest, dass der Codeblock mindestens einmal ausgeführt wird, unabhängig davon, ob die Bedingung zu Beginn wahr ist oder nicht.

Beispiel - Benutzereingabe einlesen:

```
#include <iostream>

int main() {
    char input;

    do {
       std::cout << "Drücke 'q' und dann Enter, um die Schleife zu beenden." << std::endl;
        std::cin >> input;
    } while (input != 'q');

    std::cout << "Schleife beendet." << std::endl;
    return 0;
}
```

In diesem Beispiel wird die do-while-Schleife dazu verwendet, die Benutzereingabe zu lesen, und der Codeblock wird mindestens einmal ausgeführt. Die Schleife wird beendet, wenn der Benutzer ‚q' eingibt und die Enter-Taste drückt.

Die do-while-Schleife ist eine praktische Schleifenstruktur, wenn du sicherstellen möchtest, dass ein Codeblock mindestens einmal ausgeführt wird, bevor die Bedingung überprüft wird.

1.13. Zusammenfassung

1.13.1. Elementare Datentypen:

Elementare Datentypen sind grundlegende Datentypen in C++, die verwendet werden, um verschiedene Arten von Daten zu speichern, z. B. Ganzzahlen, Gleitkommazahlen, Zeichen und Wahrheitswerte. Beispiele für elementare Datentypen: int (Ganzzahl), float und double (Gleitkommazahlen), char (Zeichen), bool (Wahrheitswert).

1.13.2. Vereinheitlichte Initialisierungssyntax:

C++11 führte die vereinheitlichte Initialisierungssyntax ein, die es ermöglicht, Variablen mit geschweiften Klammern {} zu initialisieren. Beispiel: int num{ 10 };

1.13.3. Typen & Variablen:

In C++ müssen Variablen deklariert und initialisiert werden, bevor sie verwendet werden können. Die Syntax für die Deklaration einer Variablen ist: Datentyp Variablenname; Beispiel: int age;

1.13.4. Automatische Typ-Deduktion mit „auto“:

C++11 führte das auto-Schlüsselwort ein, das die automatische Typ-Deduktion ermöglicht. Der Datentyp einer Variable wird anhand ihres zugewiesenen Werts automatisch ermittelt. Beispiel: auto value = 10;

1.13.5. Lokale Variablen:

Lokale Variablen werden innerhalb eines Codeblocks (z. B. einer Funktion) deklariert und sind nur innerhalb dieses Codeblocks sichtbar. Beispiel:

```
    void someFunction() {
        int localVar = 20;
        // Rest des Codes
    }
```

1.13.6. Globale Variablen:

Globale Variablen werden außerhalb aller Funktionen deklariert und sind im gesamten Programm sichtbar. Sie können in verschiedenen Funktionen verwendet werden. Beispiel:

```
    #include <iostream>

    int globalVar = 100;

    void function1() {
        std::cout << "Global Variable: " << globalVar << std::endl;
```

```
}

void function2() {
    std::cout << "Global Variable: " << globalVar << std::endl;
}
```

1.13.7. Konstanten:

Konstanten sind Variablen, deren Wert während der Programmausführung nicht geändert werden kann. Sie werden mit dem const-Schlüsselwort deklariert. Beispiel:

```
const double PI = 3.14159;
```

1.13.8. Operatoren:

C++ unterstützt verschiedene Arten von Operatoren, darunter arithmetische, Zuweisungs-, Vergleichs- und logische Operatoren. Arithmetische Operatoren: `+` (Addition), `-` (Subtraktion), `*` (Multiplikation), `/` (Division), `%` (Modulo). Beispiel:

```
int a = 10, b = 20;
int sum = a + b; // sum ist 30
```

1.13.9. if-else-Anweisungen:

Die if-else-Anweisung ermöglicht die Ausführung von Codeblöcken basierend auf einer Bedingung. Beispiel:

```
int num = 15;
if (num > 10) {
    std::cout << "Die Zahl ist größer als 10." << std::endl;
} else {
    std::cout << "Die Zahl ist nicht größer als 10." << std::endl;
}
```

1.13.10. switch-Anweisung:

Die switch-Anweisung ermöglicht die Auswahl zwischen mehreren möglichen Werten einer Variablen und führt den entsprechenden Codeblock aus. Beispiel:

```
char grade = 'B';
switch (grade) {
    case 'A':
        std::cout << "Sehr gut!" << std::endl;
        break;
```

```
        case 'B':
            std::cout << "Gut!" << std::endl;
            break;
        // Weitere case-Blöcke für andere Noten
        default:
            std::cout << "Ungültige Note." << std::endl;
            break;
    }
```

1.13.11. for-Schleife:

Die for-Schleife wird verwendet, um einen Codeblock eine bestimmte Anzahl von Malen auszuführen. Beispiel:

```
    for (int i = 0; i < 5; i++) {
        std::cout << "Schleifendurchlauf #" << i << std::endl;
    }
```

1.13.12. while-Schleife:

Die while-Schleife wird verwendet, um einen Codeblock auszuführen, solange eine bestimmte Bedingung wahr ist. Beispiel:

```
    int count = 1;
    while (count <= 5) {
        std::cout << "Schleifendurchlauf #" << count << std::endl;
        count++;
    }
```

1.13.13. do-while-Schleife:

Die do-while-Schleife wird verwendet, um einen Codeblock mindestens einmal auszuführen, bevor die Bedingung überprüft wird. Beispiel:

```
int num = 1;
do {
    std::cout << "Zahl: " << num << std::endl;
    num++;
} while (num <= 5);
```

1.14. Fragen

1.14.1. Multiple-Choice

Frage 1: Welches Schlüsselwort ermöglicht die automatische Typ-Deduktion einer Variablen in C++?

1. int
2. auto
3. var
4. type

Frage 2: Welche der folgenden Schleifen führt den Codeblock mindestens einmal aus, bevor die Bedingung überprüft wird?

1. for-Schleife
2. while-Schleife
3. do-while-Schleife
4. switch-Anweisung

Frage 3: Was ist der Zweck einer const-Variablen in C++?

1. Sie ist eine Variable, die den Wert ändern kann.
2. Sie ist eine Variable, die den Speicherplatz nicht belegt.
3. Sie ist eine Variable, die während der Programmausführung nicht geändert werden kann.
4. Sie ist eine Variable, die nur in Funktionen verwendet werden kann.

Frage 4: Welche der folgenden Operatoren führt eine Ganzzahldivision durch?

1. +
2. -
3. *
4. /

Frage 5: Wie deklariert man eine globale Variable in C++?

1. Durch Eingabe des global-Schlüsselworts vor dem Variablennamen.
2. Durch Deklaration innerhalb einer Funktion.
3. Durch Deklaration mit dem global-Attribut.
4. Durch Deklaration außerhalb aller Funktionen.

Frage 6: Was ist der Unterschied zwischen einer Initialisierung und einer Deklaration einer Variablen in C++?

1. Es gibt keinen Unterschied; beide Begriffe werden synonym verwendet.
2. Eine Initialisierung setzt den Wert einer Variablen, während eine Deklaration ihren Datentyp angibt.
3. Eine Deklaration gibt einer Variablen einen Namen, während eine Initialisierung ihren Speicherplatz reserviert.
4. Eine Initialisierung erfolgt mit = und einer Konstante, während eine Deklaration das var-Schlüsselwort verwendet.

Frage 7: Welche der folgenden Aussagen zum Schlüsselwort const in C++ ist korrekt?

1. Eine const-Variable kann während der Programmausführung nicht geändert werden.
2. Das Schlüsselwort const wird verwendet, um einen Datentyp zu definieren.
3. Eine const-Variable kann nur in Funktionen, aber nicht global, verwendet werden.
4. Eine const-Variable muss beim Deklarieren sofort initialisiert werden.

Frage 8: Welches der folgenden Schlüsselwörter in C++ wird verwendet, um die Schleifensteuerung vorzeitig zu beenden und zur nächsten Schleife oder zum Ende der Schleife zu springen?

1. continue
2. exit
3. break
4. return

Frage 9: Welche der folgenden Aussagen über die for-Schleife in C++ ist falsch?

1. Die for-Schleife kann nicht zum Durchlaufen von Arrays verwendet werden.
2. Die for-Schleife hat einen Initialisierungs-, einen Bedingungs- und einen Inkrementierungsteil.
3. Der Initialisierungsteil einer for-Schleife wird einmal ausgeführt, bevor die Schleife beginnt.
4. Die for-Schleife wird verwendet, wenn die Anzahl der Schleifendurchläufe bekannt ist.

Frage 10: Was ist der Zweck der do-while-Schleife im Vergleich zur while-Schleife?

1. Die do-while-Schleife wird einmal ausgeführt, bevor die Bedingung überprüft wird, während die while-Schleife dies nicht tut.
2. Die do-while-Schleife ist effizienter als die while-Schleife.
3. Die do-while-Schleife kann nicht für Schleifen verwendet werden, die mehr als 10 Mal wiederholt werden sollen.
4. Es gibt keinen Unterschied zwischen der do-while- und der while-Schleife.

Übersicht der richtigen Antworten:

1. 2) auto
2. 3) do-while-Schleife
3. 3) Sie ist eine Variable, die während der Programmausführung nicht geändert werden kann.
4. 4) /
5. 5) Durch Deklaration außerhalb aller Funktionen.
6. 2) Eine Initialisierung setzt den Wert einer Variablen, während eine Deklaration ihren Datentyp angibt.
7. 1) Eine const-Variable kann während der Programmausführung nicht geändert werden.
8. 3) break
9. 1) Die for-Schleife kann nicht zum Durchlaufen von Arrays verwendet werden.
10. 1) Die do-while-Schleife wird einmal ausgeführt, bevor die Bedingung überprüft wird, während die while-Schleife dies nicht tut.

1.14.2. Erklärungen

1. Erkläre den Unterschied zwischen den Datentypen float und double in C++.
2. Erkläre, wie die automatische Typ-Deduktion mit dem Schlüsselwort auto funktioniert und wann sie in C++ verwendet wird.
3. Erkläre den Unterschied zwischen lokalen und globalen Variablen in C++ und wann es sinnvoll ist, welche zu verwenden.
4. Erkläre den Zweck der const-Variablen in C++ und wann sie in einem Programm nützlich sind.
5. Erkläre, wie die for-Schleife in C++ funktioniert und gebe ein Beispiel für ihre Verwendung.
6. Erkläre, wie die while-Schleife in C++ funktioniert und gebe ein Beispiel für ihre Verwendung.
7. Erkläre den Unterschied zwischen der do-while-Schleife und der while-Schleife in C++ und wann es sinnvoll ist, welche zu verwenden.
8. Erkläre den Zweck der switch-Anweisung in C++ und gebe ein Beispiel für ihre Verwendung.
9. Erkläre, wie break und continue in C++ verwendet werden und was sie in einer Schleife bewirken.

10. Erkläre, wie man eine globale Variable in C++ deklariert und warum es wichtig ist, sie richtig zu verwenden.

Richtige Antworten:

1. float wird für Gleitkommazahlen mit einfacher Genauigkeit verwendet, während double für Gleitkommazahlen mit doppelter Genauigkeit verwendet wird. double bietet mehr Genauigkeit als float.
2. Die automatische Typ-Deduktion mit auto erlaubt es dem Compiler, den Datentyp einer Variablen anhand ihres zugewiesenen Werts automatisch zu bestimmen. Es wird verwendet, um den Code lesbarer und flexibler zu gestalten, insbesondere bei der Verwendung von komplexen oder generischen Datentypen.
3. Lokale Variablen werden innerhalb eines Codeblocks deklariert und sind nur in diesem Codeblock sichtbar. Globale Variablen werden außerhalb von Funktionen deklariert und sind im gesamten Programm sichtbar. Lokale Variablen werden verwendet, wenn die Variable nur in einem begrenzten Bereich gültig sein soll, während globale Variablen verwendet werden, wenn die Variable von verschiedenen Teilen des Programms aus zugänglich sein muss.
4. const-Variablen sind Variablen, deren Wert während der Programmausführung nicht geändert werden kann. Sie werden verwendet, um Konstanten zu definieren oder um sicherzustellen, dass ein Wert nicht versehentlich geändert wird.
5. Die for-Schleife in C++ besteht aus einem Initialisierungs-, einem Bedingungs- und einem Inkrementierungsteil. Der Initialisierungsteil wird einmalig ausgeführt, bevor die Schleife beginnt. Der Bedingungsteil wird vor jedem Schleifendurchlauf überprüft, und der Inkrementierungsteil wird nach jedem Schleifendurchlauf ausgeführt.
6. Die while-Schleife wird verwendet, um einen Codeblock auszuführen, solange eine bestimmte Bedingung wahr ist. Bevor der Codeblock ausgeführt wird, wird die Bedingung überprüft, und wenn sie wahr ist, wird der Codeblock ausgeführt.
7. Die do-while-Schleife wird mindestens einmal ausgeführt, bevor die Bedingung überprüft wird, während die while-Schleife dies nicht tut. Dies macht do-while nützlich, wenn du sicherstellen möchtest, dass der Codeblock mindestens einmal ausgeführt wird, unabhängig davon, ob die Bedingung am Anfang wahr ist oder nicht.

8. Die switch-Anweisung in C++ ermöglicht eine Auswahl zwischen mehreren möglichen Werten einer Variablen. Sie verwendet case-Blöcke, um zu entscheiden, welcher Codeblock ausgeführt wird, basierend auf dem Wert der Variable.
9. break wird in einer Schleife oder einer switch-Anweisung verwendet, um die Schleife vorzeitig zu beenden oder den Codeblock der switch-Anweisung zu verlassen. continue wird verwendet, um den aktuellen Schleifendurchlauf zu beenden und zum nächsten Schleifendurchlauf zu springen.
10. Eine globale Variable in C++ wird außerhalb aller Funktionen deklariert. Es ist wichtig, globale Variablen sorgfältig zu verwenden, da sie überall im Programm zugänglich sind und Änderungen an globalen Variablen unvorhersehbare Auswirkungen auf andere Teile des Programms haben können. In größeren Programmen sollten globale Variablen auf ein Minimum beschränkt werden und stattdessen lokale Variablen bevorzugt werden.

2. Ein. und Ausgabe

Die Ein- und Ausgabe (E/A) ist ein wichtiger Aspekt der Programmierung, da sie es ermöglicht, mit dem Benutzer zu interagieren und Informationen auszugeben. In C++ werden die Standard-Bibliotheksfunktionen cin und cout verwendet, um die Eingabe von Daten zu lesen und Ausgaben auf der Konsole zu erzeugen.

2.1. Eingabe mit cin:

Die Funktion cin ist Teil der C++-Standardbibliothek und wird verwendet, um Benutzereingaben von der Konsole einzulesen. Mit cin kannst du Werte unterschiedlicher Datentypen einlesen, wie Ganzzahlen, Gleitkommazahlen, Zeichen und Zeichenketten.

Um eine Ganzzahl einzulesen, verwendest du den >> -Operator zusammen mit der Variable, in der du den Wert speichern möchtest:

```
#include <iostream>

int main() {
    int num;
    std::cout << "Gib eine Ganzzahl ein: ";
    std::cin >> num;
    std::cout << "Du hast die Zahl " << num << " eingegeben." << std::endl;
```

```
    return 0;
}
```

Um eine Zeichenkette einzulesen, kannst du getline() verwenden:

```
#include <iostream>
#include <string>

int main() {
    std::string name;
    std::cout << "Gib deinen Namen ein: ";
    std::getline(std::cin, name);
    std::cout << "Hallo, " << name << "!" << std::endl;

    return 0;
}
```

getline() wird verwendet, um eine komplette Zeile einzulesen, einschließlich Leerzeichen, bis zur Eingabetaste (Enter) des Benutzers.

2.2. Ausgabe mit cout:

Die Funktion cout ist ebenfalls Teil der C++-Standardbibliothek und wird verwendet, um Ausgaben auf der Konsole zu erzeugen. Mit cout kannst du Werte unterschiedlicher Datentypen ausgeben.

Um beispielsweise eine Ganzzahl und eine Zeichenkette auszugeben, verwendest du den << -Operator, um die Werte zu concaten:

```
#include <iostream>
#include <string>

int main() {
    int age = 25;
    std::string name = "John Doe";

    std::cout << "Name: " << name << std::endl;
    std::cout << "Alter: " << age << " Jahre" << std::endl;
```

```
    return 0;
}
```

Die Ausgabe wäre:

makefile

Name: John Doe Alter: 25 Jahre

2.3. Fehlschläge bei der Eingabe:

Bei der Benutzereingabe mit cin ist es wichtig zu bedenken, dass cin auf eine Eingabe trifft, die nicht dem erwarteten Datentyp entspricht, die Eingabe fehlschlägt und der Wert der Variablen unverändert bleibt. Dies kann zu unerwünschtem Verhalten führen. Daher ist es ratsam, die Eingabe zu überprüfen und Fehler abzufangen.

Im folgenden Beispiel wird eine Schleife verwendet, um fehlerhafte Eingaben zu behandeln und den Benutzer aufzufordern, es erneut zu versuchen:

```
#include <iostream>

int main() {
    int num;
    std::cout << "Gib eine Ganzzahl ein: ";

    // Überprüfung der Eingabe
    while (!(std::cin >> num)) {
        std::cout << "Ungültige Eingabe. Versuche es erneut: ";
        std::cin.clear();
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    }

    std::cout << "Du hast die Zahl " << num << " eingegeben." << std::endl;

    return 0;
}
```

In diesem Beispiel wird die Schleife so lange wiederholt, bis eine gültige Ganzzahl eingegeben wird. cin.clear() wird verwendet, um den Fehlerzustand von cin zu löschen, und cin.ignore() wird verwendet, um den Puffer zu leeren und alle überschüssigen Zeichen zu ignorieren.

Die Ein- und Ausgabe-Funktionen cin und cout ermöglichen es, interaktive Programme zu erstellen, Benutzereingaben zu verarbeiten und nützliche Informationen an den Benutzer auszugeben.

3. Strings

Ein String ist eine Sequenz von Zeichen, die in C++ durch die Standard-Bibliotheksklasse std::string repräsentiert wird. Die std::string-Klasse ist Teil der C++-Standardbibliothek und bietet eine Vielzahl von Funktionen, um mit Zeichenketten zu arbeiten.

3.1. String-Länge:

Die Funktionen length() und size() in der std::string-Klasse werden verwendet, um die Länge eines Strings zu ermitteln. Beide Funktionen geben die Anzahl der Zeichen im String zurück.

Beispiel:

```
#include <iostream>
#include <string>

int main() {
    std::string str = "Hallo, Welt!";
    std::cout << "Länge des Strings: " << str.length() << std::endl;
    // oder: std::cout << "Länge des Strings: " << str.size() << std::endl;

    return 0;
}
```

3.2. String-Verkettungen:

Die std::string-Klasse ermöglicht die einfache Verkettung von Strings mithilfe des +-Operators. Damit kannst du zwei oder mehr Strings zu einer einzigen Zeichenkette zusammenfügen.

Beispiel:

```
#include <iostream>
#include <string>

int main() {
    std::string str1 = "Hallo, ";
    std::string str2 = "Welt!";
    std::string result = str1 + str2;
    std::cout << result << std::endl;
```

```
    return 0;
}
```

Das Ergebnis der Verkettung der Strings str1 und str2 ist der neue String result, der „Hallo, Welt!“ lautet.

3.3. String-Wandlungen:

Die std::string-Klasse bietet auch Funktionen, um Strings in andere Datentypen umzuwandeln. Beispielsweise kannst du einen String in eine Ganzzahl oder eine Gleitkommazahl umwandeln.

Beispiel:

```
#include <iostream>
#include <string>

int main() {
    std::string num_str = "42";
    int num = std::stoi(num_str);
    std::cout << "Die Zahl ist: " << num << std::endl;

    return 0;
}
```

In diesem Beispiel wird die Funktion std::stoi() verwendet, um den String „42“ in die Ganzzahl 42 umzuwandeln. Es gibt auch andere Funktionen wie std::stod() für Gleitkommazahlen und std::stol() für lange Ganzzahlen.

3.4. String-Substring:

Mit der Funktion substr() kannst du einen Teil eines Strings extrahieren. substr() erwartet zwei Argumente: den Startindex und die Länge des Substrings.

Beispiel:

```
#include <iostream>
#include <string>

int main() {
    std::string str = "Hallo, Welt!";
    std::string sub = str.substr(7, 5);
    std::cout << "Substring: " << sub << std::endl;
```

```
    // Ausgabe: "Substring: Welt"

    return 0;
}
```

Hier wird substr(7, 5) verwendet, um den Teil des Strings ab dem 8. Zeichen (Index 7) mit einer Länge von 5 Zeichen zu extrahieren, was den Substring „Welt“ ergibt.

3.5. Weitere nützliche Funktionen:

Die std::string-Klasse bietet eine Vielzahl von nützlichen Funktionen, die beim Arbeiten mit Zeichenketten hilfreich sind. Einige Beispiele sind:

find(): Sucht nach einem Teilstring in einem String und gibt den Index des ersten Vorkommens zurück. replace(): Ersetzt Teilstrings in einem String durch andere Zeichenketten. compare(): Vergleicht zwei Strings lexikographisch und gibt eine Zahl zurück, die angibt, ob sie gleich sind oder welcher String vor dem anderen liegt.

Beispiel:

```
#include <iostream>
#include <string>

int main() {
    std::string str = "Hallo, Welt!";
    size_t found = str.find("Welt");

    if (found != std::string::npos) {
        std::cout << "Teilstring gefunden bei Index: " << found << std::endl;
    }

    str.replace(7, 4, "Erde");
    std::cout << "Neuer String: " << str << std::endl;

    if (str.compare("Hallo, Erde!") == 0) {
        std::cout << "Strings sind gleich." << std::endl;
    }

    return 0;
}
```

Hier wird find() verwendet, um nach dem Teilstring „Welt“ zu suchen, replace() um „Welt“ durch „Erde“ zu ersetzen und compare() um den String mit „Hallo, Erde!“ zu vergleichen.
Strings in C++ sind äußerst vielseitig und bieten eine Fülle von Funktionen, die das Arbeiten mit Zeichenketten erleichtern.

3.6. Standard-Konstruktor:

Der Standard-Konstruktor erstellt einen leeren String.

```
#include <iostream>
#include <string>

int main() {
    std::string str; // Leerer String wird erstellt
    std::cout << "Leerer String: " << str << std::endl;

    return 0;
}
```

3.7. Konstruktor mit C-String:

Du kannst einen C-String verwenden, um einen std::string zu initialisieren.

```
#include <iostream>
#include <string>

int main() {
    const char* cstr = "Hallo, Welt!";
    std::string str(cstr); // String mit C-String initialisieren
    std::cout << "String mit C-String: " << str << std::endl;

    return 0;
}
```

3.8. Kopier-Konstruktor:

Der Kopier-Konstruktor erstellt einen neuen String, der eine Kopie eines vorhandenen Strings ist.

```
#include <iostream>
#include <string>
```

```cpp
int main() {
    std::string original = "Hallo";
    std::string kopie(original); // Kopier-Konstruktor
    std::cout << "Kopierter String: " << kopie << std::endl;

    return 0;
}
```

3.9. Konstruktor mit Zeichen und Länge:

Du kannst auch einen Konstruktor verwenden, der eine bestimmte Anzahl von Zeichen eines C-Strings kopiert.

```cpp
#include <iostream>
#include <string>

int main() {
    const char* cstr = "Hello, World!";
    std::string str(cstr, 5); // Die ersten 5 Zeichen des C-Strings werden kopiert
    std::cout << "String mit begrenzter Länge: " << str << std::endl;

    return 0;
}
```

3.10. Initialisierung mit einem Zeichen:

Du kannst einen String mit einer bestimmten Anzahl von wiederholten Zeichen initialisieren.

```cpp
#include <iostream>
#include <string>

int main() {
    char zeichen = 'A';
    std::string str(5, zeichen); // String wird mit 5x 'A' initialisiert
    std::cout << "String mit wiederholtem Zeichen: " << str << std::endl;

    return 0;
}
```

3.11. Konstruktor mit Iteratoren:

Du kannst auch einen Konstruktor verwenden, der zwei Iteratoren akzeptiert, um einen String aus einem Bereich von Zeichen zu erstellen.

```
#include <iostream>
#include <string>

int main() {
    std::string original = "Hello, World!";
    std::string str(original.begin() + 7, original.end()); // String wird aus dem Bereich 'World!' erstellt
    std::cout << "String mit Iteratoren: " << str << std::endl;

    return 0;
}
```

Das sind nur einige Beispiele für die Verwendung der Konstruktoren von std::string. Die C++-Standardbibliothek bietet noch weitere Konstruktoren und Funktionen, um mit Strings zu arbeiten.

4. Vektoren

Vektoren sind eine der vielseitigsten Datenstrukturen in C++, die Teil der Standard Template Library (STL) sind. Ein Vektor ist ein dynamischer Container, der es ermöglicht, eine Liste von Elementen zu speichern und effizient auf sie zuzugreifen. Es ist vergleichbar mit einem Array, aber im Gegensatz zu Arrays, deren Größe zur Kompilierzeit festgelegt wird, kann die Größe eines Vektors zur Laufzeit geändert werden.
Um Vektoren in C++ zu verwenden, musst du die vector-Header-Datei in dein Programm einbeziehen:

```
#include <iostream>
#include <vector>
```

4.1. Merkmale

1. Der Vektor ist ein sequentieller Container, d.h
    1. die Elemente liegen hintereinander (in einer Sequenz)
    2. der Nutzer des Containers bestimmt die Reihenfolge der Elemente
    3. der Container selber verändert die Reihenfolge nicht.

2. Der Vektor unterstützt wahlfreien Zugriff, d. h. man kann direkt auf das n-te Element zugreifen (ohne Performance-Probleme).
3. Der Vektor benötigt relativ wenig Speicher, da er kaum interne Verwaltungs-Informationen benötigt und die Elemente bündig im Speicher aneinander liegen.
4. Anfügen neuer Elemente am Ende des Vektors und Löschen von Elementen am Ende des Vektors geht sehr schnell.
5. Muss dagegen vorne ein Element eingefügt oder gelöscht werden, so ist dies sehr aufwändig und damit langsam. Einfüge- und Lösch-Operationen mitten im Vektor sind abhängig von der Position langsam oder schnell – im Mittel aber schlecht.
6. Beim Suchen nach einem Element im Vektor muss der Vektor von vorn nach hinten durchlaufen werden – d.h. die benötigte Suchzeit steigt linear zur Anzahl der Elemente im Vektor.

4.2. Vektor erstellen:

Du kannst einen Vektor erstellen, indem du einfach den Datentyp angeben und ihn initialisierst.

Zum

Beispiel:

```
std::vector<int> numbers; // Erstellt einen leeren Vektor von Ganzzahlen
std::vector<std::string> names; // Erstellt einen leeren Vektor von Zeichenketten
```

4.3. Elemente hinzufügen:

Du kannst Elemente am Ende des Vektors mit der Funktion push_back() hinzufügen. push_back() akzeptiert ein Element als Argument und fügt es am Ende des Vektors hinzu:

```
std::vector<int> numbers;
numbers.push_back(10);
numbers.push_back(20);
numbers.push_back(30);
```

4.4. Elemente zugreifen:

Du kannst auf die Elemente eines Vektors mithilfe des Indexoperators [] zugreifen. Beachte, dass der Index des ersten Elements 0 ist:

```
std::vector<int> numbers;


numbers.push_back(10);
numbers.push_back(20);
```

```
numbers.push_back(30);


std::cout << "Erstes Element: " << numbers[0] << std::endl; // Ausgabe: 10
std::cout << "Zweites Element: " << numbers[1] << std::endl; // Ausgabe: 20
std::cout << "Drittes Element: " << numbers[2] << std::endl; // Ausgabe: 30
```

Beachte, dass du auf ein Element mithilfe des Index zugreifen kannst, aber es wird nicht überprüft, ob der Index gültig ist. Du solltest sicherstellen, dass du nur auf gültige Indizes zugreifst, um undefiniertes Verhalten zu vermeiden.

4.5. Vektor-Größe:

Du kannst die Anzahl der Elemente in einem Vektor mit der Funktion size() ermitteln. Die Funktion size() gibt die Anzahl der Elemente im Vektor als size_t zurück:

```
std::vector<int> numbers;


numbers.push_back(10);
numbers.push_back(20);
numbers.push_back(30);


std::cout << "Anzahl der Elemente: " << numbers.size() << std::endl; // Ausgabe: 3
```

4.6. Vektor leeren:

Du kannst alle Elemente aus einem Vektor entfernen und ihn leeren, indem du die Funktion clear() verwendest:

```
std::vector<int> numbers;


numbers.push_back(10);
numbers.push_back(20);
numbers.push_back(30);


numbers.clear(); // Vektor wird geleert


std::cout << "Anzahl der Elemente nach dem Leeren: " << numbers.size() << std::endl; //
Ausgabe: 0
```

4.7. Vektor durchlaufen:

Du kannst alle Elemente in einem Vektor durchlaufen und auf sie zugreifen. Hier sind zwei gängige Methoden, um dies zu tun:

Mit einer Schleife und dem Indexoperator []:

```
std::vector<int> numbers;

numbers.push_back(10);
numbers.push_back(20);
numbers.push_back(30);

for (size_t i = 0; i < numbers.size(); ++i) {
    std::cout << numbers[i] << " ";
}
// Ausgabe: 10 20 30
```

Mit einer Range-basierten Schleife (C++11 und höher):

```
std::vector<int> numbers;

numbers.push_back(10);
numbers.push_back(20);
numbers.push_back(30);

for (int num : numbers) {
    std::cout << num << " ";
}
// Ausgabe: 10 20 30
```

Die Range-basierte Schleife ist in der Regel bevorzugt, da sie weniger anfällig für Fehler ist und den Code lesbarer macht.

4.8. Vektor mit Initialisierungsliste erstellen:

Du kannst auch einen Vektor mit einer Initialisierungsliste erstellen. Das ist besonders nützlich, wenn du den Vektor mit Anfangswerten initialisieren möchtest:

```
std::vector<int> numbers = {10, 20, 30};

for (int num : numbers) {
    std::cout << num << " ";
```

```
}
// Ausgabe: 10 20 30
```

4.9. Weitere nützliche Funktionen:

Die std::vector-Klasse bietet viele nützliche Funktionen, um mit Vektoren zu arbeiten. Hier sind einige davon:

4.9.1. empty():

Überprüft, ob der Vektor leer ist und gibt true zurück, wenn er leer ist, andernfalls false.

```
std::vector<int> numbers;

if (numbers.empty()) {
    std::cout << "Der Vektor ist leer." << std::endl;
}
```

4.9.2. pop_back():

Entfernt das letzte Element des Vektors.

```
std::vector<int> numbers = {10, 20, 30};

numbers.pop_back(); // Entfernt das letzte Element (30)

for (int num : numbers) {
    std::cout << num << " ";
}
// Ausgabe: 10 20
```

4.9.3. insert():

Fügt ein Element an einer bestimmten Position im Vektor ein.

```
std::vector<int> numbers = {10, 20, 30};

numbers.insert(numbers.begin() + 1, 15); // Fügt die Zahl 15 an der Position 1 ein

for (int num : numbers) {
    std::cout << num << " ";
}
// Ausgabe: 10 15 20 30
```

4.9.4. erase():
Entfernt ein oder mehrere Elemente aus dem Vektor.

```
std::vector<int> numbers = {10, 20, 30, 40, 50};

numbers.erase(numbers.begin() + 2); // Entfernt das Element an der Position 2 (Wert: 30)

for (int num : numbers) {
    std::cout << num << " ";
}
// Ausgabe: 10 20 40 50
```

4.9.5. resize():
Ändert die Größe des Vektors.

```
std::vector<int> numbers = {10, 20, 30};

numbers.resize(5); // Vergrößert den Vektor auf die Größe 5, fügt 2 zusätzliche Elemente hinzu

for (int num : numbers) {
    std::cout << num << " ";
}
// Ausgabe: 10 20 30 0 0
```

4.9.6. swap():
Vertauscht den Inhalt zweier Vektoren.

```
std::vector<int> numbers1 = {1, 2, 3};
std::vector<int> numbers2 = {10, 20, 30};

numbers1.swap(numbers2);

for (int num : numbers1) {
    std::cout << num << " ";
}
// Ausgabe: 10 20 30

for (int num : numbers2) {
    std::cout << num << " ";
```

```
}
// Ausgabe: 1 2 3
```

Das sind einige der nützlichen Funktionen, die in der std::vector-Klasse verfügbar sind. Vektoren bieten eine flexible Möglichkeit, Daten in C++ zu verwalten und sie sind eine der am häufigsten verwendeten Container in der STL.

5. Zusammenfassung

5.1. Ein- und Ausgabe in C++:

5.1.1. Ausgabe mit std::cout:

Die std::cout-Funktion wird verwendet, um Daten auf der Konsole auszugeben:

```
#include <iostream>

int main() {
    std::cout << "Hallo, Welt!" << std::endl;
    return 0;
}
```

5.1.2. Eingabe mit std::cin:

Die std::cin-Funktion wird verwendet, um Daten von der Konsole einzulesen:

```
#include <iostream>

int main() {
    int zahl;
    std::cout << "Geben Sie eine Zahl ein: ";
    std::cin >> zahl;
    std::cout << "Sie haben die Zahl " << zahl << " eingegeben." << std::endl;
    return 0;
}
```

5.2. Strings in C++:

Strings werden verwendet, um Zeichenketten in C++ zu speichern und zu manipulieren.

5.2.1. String erstellen:

Du kannst einen String erstellen, indem du den Datentyp std::string verwendest:

```
#include <iostream>
#include <string>
```

```
int main() {
    std::string text = "Hallo, Welt!";
    std::cout << text << std::endl;
    return 0;
}
```

5.2.2. String-Verkettung:

Strings können mithilfe des +-Operators verkettet werden:

```
#include <iostream>
#include <string>

int main() {
    std::string vorname = "Max";
    std::string nachname = "Mustermann";
    std::string vollerName = vorname + " " + nachname;
    std::cout << "Voller Name: " << vollerName << std::endl;
    return 0;
}
```

5.2.3. String-Länge:

Du kannst die Länge eines Strings mit der Funktion length() oder size() ermitteln:

```
#include <iostream>
#include <string>

int main() {
    std::string text = "Hallo, Welt!";
    std::cout << "Länge des Strings: " << text.length() << std::endl;
    return 0;
}
```

5.2.4. String-Eingabe:

Du kannst auch Strings von der Konsole mit std::cin einlesen:

```
#include <iostream>
#include <string>

int main() {
```

```
    std::string name;
    std::cout << "Geben Sie Ihren Namen ein: ";
    std::cin >> name;
    std::cout << "Hallo, " << name << "!" << std::endl;
    return 0;
}
```

5.3. Vektoren

Vektoren sind eine dynamische Datenstruktur in C++, die Teil der Standard Template Library (STL) ist. Sie erlauben das Speichern und Verwalten einer Liste von Elementen in C++.

5.3.1. Vektor-Definition:

Um Vektoren in C++ zu verwenden, musst du die vector-Header-Datei einbeziehen:

```
#include <iostream>
#include <vector>
```

5.3.2. Vektor erstellen:

Du kannst einen Vektor erstellen, indem du den Datentyp angibst und ihn initialisierst:

```
std::vector<int> numbers; // Vektor von Ganzzahlen
std::vector<std::string> names; // Vektor von Zeichenketten
```

5.3.3. Elemente hinzufügen:

Du kannst Elemente am Ende des Vektors mit push_back() hinzufügen:

```
numbers.push_back(10);
numbers.push_back(20);
numbers.push_back(30);
```

5.3.4. Elemente zugreifen:

Du kannst auf die Elemente eines Vektors mithilfe des Indexoperators [] zugreifen:

```
std::cout << "Erstes Element: " << numbers[0] << std::endl; // Ausgabe: 10
std::cout << "Zweites Element: " << numbers[1] << std::endl; // Ausgabe: 20
std::cout << "Drittes Element: " << numbers[2] << std::endl; // Ausgabe: 30
```

5.3.5. Vektor-Größe:

Du kannst die Anzahl der Elemente in einem Vektor mit size() ermitteln:

```
std::cout << "Anzahl der Elemente: " << numbers.size() << std::endl; // Ausgabe: 3
```

5.3.6. Vektor leeren:

Du kannst alle Elemente aus einem Vektor entfernen und ihn leeren mit clear():

```
numbers.clear(); // Vektor wird geleert
```

5.3.7. Vektor durchlaufen:

Du kannst alle Elemente in einem Vektor durchlaufen und auf sie zugreifen:

```
for (int i = 0; i < numbers.size(); ++i) {
    std::cout << numbers[i] << " ";
}
// Ausgabe: 10 20 30
```

Du kannst auch eine Range-basierte Schleife verwenden (C++11 und höher):

```
for (int num : numbers) {
    std::cout << num << " ";
}
// Ausgabe: 10 20 30
```

5.3.8. Vektor mit Initialisierungsliste erstellen:

Du kannst auch einen Vektor mit einer Initialisierungsliste erstellen:

```
std::vector<int> numbers = {10, 20, 30};
```

5.3.9. Weitere nützliche Funktionen:

Die std::vector-Klasse bietet viele weitere nützliche Funktionen wie empty(), pop_back(), insert(), erase(), resize() und swap(), um mit Vektoren zu arbeiten.

6. Fragen

6.1. Multiple-Choice Fragen

Frage 1: Was ist die richtige Art und Weise, ein Element zum Ende eines Vektors hinzuzufügen?

1. numbers.add(42);
2. numbers.push_back(42);
3. numbers.insert(42);
4. numbers.append(42);

Frage 2: Wie erhältst du die Anzahl der Elemente in einem Vektor?

1. numbers.size();
2. numbers.length();
3. numbers.count();
4. numbers.elements();

Frage 3: Welche Funktion wird verwendet, um Daten auf der Konsole auszugeben?

1. std::print();
2. std::out();
3. std::cout();
4. std::write();

Frage 4: Wie liest du eine Ganzzahl von der Konsole ein?

1. std::input();
2. std::cin();
3. std::read();
4. std::get();

Frage 5: Wie erstellst du einen String in C++?

1. std::string name = „Max“;
2. string name = ‚Max‘;
3. String name = „Max“;
4. str name = „Max“;

Frage 6: Welche Funktion wird verwendet, um die Länge eines Strings zu ermitteln?

1. text.length();
2. text.count();
3. text.size();
4. text.length();

Frage 7: Was ist der Unterschied zwischen std::cin und std::getline in C++?

1. std::cin liest nur einzelne Zeichen ein, während std::getline eine ganze Zeile einliest.
2. Es gibt keinen Unterschied, beide Funktionen haben die gleiche Funktionalität.
3. std::cin liest Zeichenketten ein, während std::getline nur einzelne Zeichen einliest.
4. std::getline liest nur einzelne Zeichen ein, während std::cin eine ganze Zeile einliest.

Frage 8: Welche Funktion wird verwendet, um einen String in einen Integer umzuwandeln?

1. std::stoi()
2. std::to_int()
3. std::string_to_int()
4. std::string::to_int()

Frage 9: Was passiert, wenn std::getline einen leeren String liest?

1. Es wird eine Ausnahme (Exception) ausgelöst.

2. Der eingelesene String wird auf „NULL“ gesetzt.
3. Der eingelesene String wird ein leeres Zeichenkettenobjekt.
4. Es wird ein Zeichen für das Ende der Datei (EOF) zurückgegeben.

Frage 10: Welche Funktion wird verwendet, um eine Zeichenkette an eine andere anzuhängen?

1. append()
2. concat()
3. add()
4. attach()

Frage 11: Was ist die Ausgabe des folgenden Codes?

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    numbers.erase(numbers.begin() + 2);
    for (int num : numbers) {
        std::cout << num << " ";
    }
    return 0;
}
```

1. 1 2 3 4 5
2. 1 2 4 5
3. 1 2 3 5
4. Es gibt einen Kompilierungsfehler.

Frage 12: Was ist die korrekte Art, eine Datei zum Schreiben zu öffnen?

1. std::file_output file(„daten.txt“);
2. std::file_open file(„daten.txt“, std::ios::write);
3. std::ofstream file(„daten.txt“);
4. std::file_open(„daten.txt“, std::ios::write);

Richtige Antworten:

1. 2) numbers.push_back(42);

2. 1) numbers.size();
3. 3) std::cout();
4. 2) std::cin();
5. 1) std::string name = „Max“;
6. 1) text.length();
7. 1) std::cin liest nur einzelne Zeichen ein, während std::getline eine ganze Zeile einliest.
8. 1) std::stoi()
9. 3) Der eingelesene String wird ein leeres Zeichenkettenobjekt.
10. 1) append()
11. 2) 1 2 4 5
12. 3) std::ofstream file(„daten.txt“);

6.2. Erklärung

Frage 1: Erkläre den Unterschied zwischen einer for-Schleife und einer while-Schleife in C++. Wann sollte man welche Schleife verwenden? **Frage 2**: Wie liest man mehrere Daten aus einer Zeile mit std::cin in C++? Beschreibe den Vorgang und gib ein Beispiel an. **Frage 3**: Erläutere die Verwendung von Iteratoren in C++ Vektoren. Wie durchläuft man einen Vektor mithilfe von Iteratoren und was sind die Vorteile dieses Ansatzes? **Frage 4**: Was sind C++ Stringstreams und wie werden sie verwendet? Gebe ein Beispiel, wie man C++ Stringstreams benutzt, um Daten aus einem String zu extrahieren. **Frage 5**: Erkläre den Unterschied zwischen einer std::ifstream und einer std::ofstream in C++. Wofür werden sie verwendet und wie öffnet man eine Datei zum Lesen bzw. Schreiben? **Frage 6**: Wie kann man in C++ eine Funktion erstellen, die eine Variable als Referenz übernimmt und was sind die Vorteile einer solchen Referenzparameter-Funktion?

6.2.1. Antworten

Frage 1: Eine for-Schleife wird verwendet, wenn die Anzahl der Schleifendurchläufe im Voraus bekannt ist oder wenn man eine bestimmte Anzahl von Wiederholungen benötigt. Die for-Schleife besteht aus einer Initialisierung, einer Bedingung und einer Aktualisierung. Eine while-Schleife wird verwendet, wenn die Anzahl der Schleifendurchläufe nicht im Voraus bekannt ist oder wenn die Schleife abhängig von einer Bedingung ausgeführt werden soll. Die while-Schleife besteht nur aus einer Bedingung.

Frage 2: Um mehrere Daten aus einer Zeile mit std::cin in C++ einzulesen, kann man den Eingabestrom std::cin mit >> und std::getline kombinieren. Mit >> können einzelne Daten wie Ganzzahlen oder Gleitkommazahlen eingelesen werden, während std::getline verwendet wird, um den Rest der Zeile einzulesen, einschließlich Leerzeichen.
Beispiel:

```
#include <iostream>
#include <string>

int main() {
    int num;
    std::string text;

    std::cout << "Geben Sie eine Zahl und einen Text ein: ";
    std::cin >> num;
    std::getline(std::cin, text);

    std::cout << "Zahl: " << num << std::endl;
    std::cout << "Text: " << text << std::endl;

    return 0;
}
```

Frage 3: Iteratoren werden in C++ Vektoren verwendet, um auf die Elemente des Vektors zuzugreifen und den Vektor zu durchlaufen. Ein Iterator ist ein Zeiger auf ein Element im Vektor, der es erlaubt, auf das aktuelle Element zuzugreifen und zum nächsten Element zu gehen.
Beispiel:

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};

    // Durchlaufen des Vektors mit einem Iterator
    for (std::vector<int>::iterator it = numbers.begin(); it != numbers.end(); ++it) {
        std::cout << *it << " ";
```

```
    }
    // Ausgabe: 1 2 3 4 5

    return 0;
}
```

Der Vorteil von Iteratoren besteht darin, dass sie in Kombination mit verschiedenen Container-Typen (wie Vektoren, Listen, Maps usw.) verwendet werden können, ohne die Schleifenlogik ändern zu müssen.

Frage 4: C++ Stringstreams (std::stringstream) sind Streams, die zum Lesen und Schreiben von Zeichenketten verwendet werden. Sie ermöglichen es, Daten in einen String zu schreiben und aus einem String zu lesen, als ob sie mit std::cin und std::cout arbeiten würden.

Beispiel:

```
#include <iostream>
#include <sstream>
#include <string>

int main() {
    std::string data = "42 3.14 Hallo";
    std::istringstream stream(data);

    int num;
    double decimal;
    std::string text;

    stream >> num >> decimal >> text;

    std::cout << "Zahl: " << num << std::endl;
    std::cout << "Dezimalzahl: " << decimal << std::endl;
    std::cout << "Text: " << text << std::endl;

    return 0;
}
```

Frage 5: std::ifstream wird verwendet, um eine Datei zum Lesen zu öffnen, während std::ofstream verwendet wird, um eine Datei zum Schreiben zu öffnen.

Öffnen einer Datei zum Lesen:

```
#include <iostream>
#include <fstream>

int main() {
    std::ifstream file("datei.txt");
    if (file.is_open()) {
        // Datei erfolgreich geöffnet, hier können Daten gelesen werden
    } else {
        std::cout << "Datei konnte nicht geöffnet werden." << std::endl;
    }
    return 0;
}
```

Öffnen einer Datei zum Schreiben:

```
#include <iostream>
#include <fstream>

int main() {
    std::ofstream file("ausgabe.txt");
    if (file.is_open()) {
        // Datei erfolgreich geöffnet, hier können Daten geschrieben werden
    } else {
        std::cout << "Datei konnte nicht geöffnet werden." << std::endl;
    }
    return 0;
}
```

Frage 6: In C++ kann man eine Funktion erstellen, die eine Variable als Referenz übernimmt, indem man einen Referenzparameter verwendet. Ein Referenzparameter wird mit einem & vor dem Datentyp deklariert.

Vorteile einer Referenzparameter-Funktion:

- Eine Referenzparameter-Funktion kann den Wert einer Variablen ändern, die von außerhalb der Funktion übergeben wird.
- Durch die Verwendung von Referenzen kann man unnötige Kopien von Daten vermeiden, was die Performance verbessert.

Beispiel:

```
#include <iostream>

void addOne(int &num) {
    num += 1;
}

int main() {
    int number = 5;
    std::cout << "Vor der Funktion: " << number << std::endl; // Ausgabe: 5

    addOne(number);
    std::cout << "Nach der Funktion: " << number << std::endl; // Ausgabe: 6

    return 0;
}
```

In diesem Beispiel wird die Funktion addOne erstellt, die eine Referenz auf eine Ganzzahl als Parameter übernimmt. Die Funktion erhöht den Wert der übergebenen Variablen um 1, und da es sich um eine Referenz handelt, wird der Wert der ursprünglichen Variable number außerhalb der Funktion geändert. Dadurch wird der Wert von number nach dem Aufruf der Funktion auf 6 geändert.
Die Verwendung von Referenzparametern kann sehr nützlich sein, um Änderungen an Variablen in Funktionen vorzunehmen und Kopien von Daten zu vermeiden.

7. Container und Iteratoren

7.1. C-ARRAYS VS. VECTOR UND ANDERE ELEMENTE DER STL

In C++ stehen mehrere Möglichkeiten zur Verfügung, um eine Sammlung von Elementen zu speichern und zu verwalten. Zu den häufig verwendeten Optionen gehören C-Arrays und Vektoren aus der Standard Template Library (STL). Beide bieten die Möglichkeit, mehrere Elemente desselben Datentyps zu speichern, aber sie haben einige Unterschiede in ihrer Funktionalität und Verwendung.

7.1.1. C-Arrays:

C-Arrays sind eine grundlegende Form der Datenspeicherung in C++. Sie sind eine Sammlung von Elementen des gleichen Datentyps, die in einem zusammenhängenden Speicherbereich

gespeichert werden. C-Arrays haben eine feste Größe, die bei ihrer Deklaration angegeben werden muss. Einmal deklariert, kann die Größe des Arrays nicht geändert werden. Die Größe eines C-Arrays ist zur Kompilierzeit festgelegt.
Beispiel für ein C-Array:

```
#include <iostream>

int main() {
    int myArray[5] = {1, 2, 3, 4, 5};

    for (int i = 0; i < 5; ++i) {
        std::cout << myArray[i] << " ";
    }

    return 0;
}
```

C-Arrays haben einige Einschränkungen, wie zum Beispiel das Fehlen von Methoden zur Größenänderung oder zur einfachen Verwaltung. Es liegt in der Verantwortung des Entwicklers, sicherzustellen, dass C-Arrays nicht über ihre Grenzen hinaus zugreifen.

7.1.2. Vektoren (std::vector):

Vektoren sind ein Teil der Standard Template Library (STL) in C++. Sie bieten eine dynamische Sammlung von Elementen des gleichen Datentyps und können während der Laufzeit ihre Größe ändern. Im Gegensatz zu C-Arrays sind Vektoren flexibel und ermöglichen das Hinzufügen und Entfernen von Elementen, ohne dass man sich um Speichermanagement kümmern muss. Die Größe eines Vektors kann zur Laufzeit geändert werden.
Beispiel für einen Vector:

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> myVector = {1, 2, 3, 4, 5};
    myVector.push_back(6); // Element am Ende hinzufügen
```

```
    for (int num : myVector) {
        std::cout << num << " ";
    }

    return 0;
}
```

Vektoren bieten auch viele nützliche Methoden zur Datenmanipulation, wie push_back(), pop_back(), size(), empty() und andere. Die Verwendung von Vektoren wird allgemein empfohlen, da sie sicherer und flexibler sind als C-Arrays.

7.1.3. Andere Elemente der STL:

Neben Vektoren bietet die STL viele andere nützliche Container-Typen und Funktionen. Einige der wichtigsten sind:

std::list: Doppelt verkettete Listen für effizientes Einfügen und Löschen von Elementen. std::map: Assoziative Container für eindeutige Zuordnung von Schlüsseln zu Werten. std::set: Ein Set von eindeutigen Elementen ohne Duplikate. std::unordered_map und std::unordered_set: Ähnlich wie std::map und std::set, aber mit schnellerem Zugriff durch Hashing. std::stack und std::queue: Container-Adapter für Stapel und Warteschlangen. std::algorithm: Eine Reihe von nützlichen Algorithmen wie std::sort, std::find, std::reverse usw.

Beispiel für die Verwendung von std::map:

```
    #include <iostream>
    #include <map>

    int main() {
        std::map<std::string, int> myMap = {{"Alice", 25}, {"Bob", 30}, {"Charlie", 27}
};

        for (auto entry : myMap) {
            std::cout << entry.first << ": " << entry.second << std::endl;
        }
```

```
        return 0;
    }
```

Die STL bietet eine reichhaltige Sammlung von Containern und Funktionen, die die Entwicklung von C++-Programmen vereinfachen und die Produktivität steigern.
Die Entscheidung, ob man C-Arrays oder Vektoren verwendet, hängt von den Anforderungen des Programms ab. In den meisten Fällen sind Vektoren aufgrund ihrer Flexibilität und Sicherheit die bevorzugte Wahl in modernen C++-Programmen. Die STL-Container bieten außerdem eine leistungsfähige und gut getestete Alternative zur Verwaltung von Datenstrukturen und -sammlungen in C++.

7.2. Iteratoren in C++

Iteratoren sind ein leistungsfähiges Konzept in C++, das es ermöglicht, durch die Elemente von Containern wie Vektoren, Listen und Maps zu iterieren. Ein Iterator kann als Zeiger auf ein Element in einem Container betrachtet werden. Er ermöglicht den Zugriff auf das Element, auf das er zeigt, und bietet die Möglichkeit, innerhalb des Containers zu navigieren.

7.2.1. Iteratoren in Vektoren:

Vektoren sind dynamische Arrays, die in der STL implementiert sind und eine flexible Möglichkeit bieten, Elemente zu speichern und zu verwalten. Iteratoren für Vektoren ermöglichen es, die Elemente im Vektor zu durchlaufen, und sie sind eine sicherere Alternative zum Zugriff über den Index.
Beim Zugriff auf Vektoren können zwei Arten von Iteratoren verwendet werden:

- begin(): Gibt einen Iterator zurück, der auf das erste Element im Vektor zeigt.
- end(): Gibt einen Iterator zurück, der auf ein Element nach dem letzten Element im Vektor zeigt. Dieser Iterator dient als Marker, der angibt, dass die Schleife beendet werden sollte.

Beispiel:

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> myVector = {1, 2, 3, 4, 5};

    // Durchlaufen des Vektors mit einem Iterator
    for (std::vector<int>::iterator it = myVector.begin(); it != myVector.end(); ++it) {
```

```
        std::cout << *it << " ";
    }

    return 0;
}
```

In modernen C++-Versionen kann die Verwendung von Iteratoren durch die Verwendung von auto vereinfacht werden:

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> myVector = {1, 2, 3, 4, 5};

    // Durchlaufen des Vektors mit einem Iterator und 'auto'
    for (auto it = myVector.begin(); it != myVector.end(); ++it) {
        std::cout << *it << " ";
    }

    return 0;
}
```

7.2.2. Iteratoren in Listen:

Listen sind doppelt verkettete Listen in der STL, die Einfügen und Löschen von Elementen effizient unterstützen. Ähnlich wie bei Vektoren können Iteratoren verwendet werden, um durch die Elemente der Liste zu iterieren.

Beispiel:

```
#include <iostream>
#include <list>

int main() {
    std::list<int> myList = {1, 2, 3, 4, 5};

    // Durchlaufen der Liste mit einem Iterator
    for (std::list<int>::iterator it = myList.begin(); it != myList.end(); ++it) {
        std::cout << *it << " ";
```

```
    }

    return 0;
}
```

Wie bei Vektoren kann auch hier die moderne Syntax mit auto verwendet werden:

```
#include <iostream>
#include <list>

int main() {
    std::list<int> myList = {1, 2, 3, 4, 5};

    // Durchlaufen der Liste mit einem Iterator und 'auto'
    for (auto it = myList.begin(); it != myList.end(); ++it) {
        std::cout << *it << " ";
    }

    return 0;
}
```

7.2.3. Iteratoren in Maps:

Maps sind assoziative Container in der STL, die eine eindeutige Zuordnung zwischen Schlüsseln und Werten bieten. Iteratoren in Maps sind auf ein Paar von Schlüssel-Wert-Paaren ausgerichtet.

Beispiel:

```
#include <iostream>
#include <map>

int main() {
    std::map<std::string, int> myMap = {{"Alice", 25}, {"Bob", 30}, {"Charlie", 27}};

    // Durchlaufen der Map mit einem Iterator
    for (std::map<std::string, int>::iterator it = myMap.begin(); it != myMap.end(); +
+it) {
        std::cout << it->first << ": " << it->second << std::endl;
    }
```

```
    return 0;
}
```

Die moderne Syntax mit auto kann auch hier verwendet werden:

```
#include <iostream>
#include <map>

int main() {
  std::map<std::string, int> myMap = {{"Alice", 25}, {"Bob", 30}, {"Charlie", 27}};

 // Durchlaufen der Map mit einem Iterator und 'auto'
 for (auto it = myMap.begin(); it != myMap.end(); ++it) {
   std::cout << it->first << ": " << it->second << std::endl;
 }

 return 0;
}
```

Iteratoren ermöglichen eine effiziente und sichere Möglichkeit, auf die Elemente in Containern zuzugreifen und sie zu durchlaufen. Sie sind in vielen Situationen nützlich, wenn es darum geht, mit Datenstrukturen zu arbeiten und ihre Elemente zu verwalten. Iteratoren sind ein wesentlicher Bestandteil der C++-Programmierung und erleichtern die Manipulation von Containern erheblich.

7.3. Listen in C++

Listen sind eine wichtige Datenstruktur in C++, die in der Standard Template Library (STL) implementiert ist. Eine Liste ist eine doppelt verkettete Liste, bei der jedes Element auf das vorherige und das nächste Element verweist. Listen bieten effiziente Einfüge- und Löschoperationen und eignen sich gut für Szenarien, in denen häufiges Einfügen oder Löschen von Elementen erforderlich ist.

7.3.1. Erstellen einer Liste:

Um eine Liste in C++ zu erstellen, müssen wir die Header-Datei inkludieren und den Container-Typ std::list verwenden. Der Typ der Elemente, die in der Liste gespeichert werden, muss ebenfalls angegeben werden.

Beispiel:

```cpp
#include <iostream>
#include <list>

int main() {
    std::list<int> myList; // Eine leere Liste von Ganzzahlen

    // Elemente zur Liste hinzufügen
    myList.push_back(10);
    myList.push_back(20);
    myList.push_back(30);
    myList.push_front(5);

    // Durchlaufen der Liste mit einem Iterator und Ausgabe der Elemente
    for (std::list<int>::iterator it = myList.begin(); it != myList.end(); ++it) {
        std::cout << *it << " ";
    }

    return 0;
}
```

7.3.2. Einfügen von Elementen in eine Liste:

Listen unterstützen zwei grundlegende Möglichkeiten zum Einfügen von Elementen: push_back() und push_front(). push_back() fügt ein Element am Ende der Liste ein, während push_front() ein Element am Anfang der Liste einfügt.

Beispiel:

```cpp
#include <iostream>
#include <list>

int main() {
    std::list<std::string> myList;

    myList.push_back("Alice");
    myList.push_back("Bob");
    myList.push_front("Charlie");

    for (const std::string& name : myList) {
```

```
        std::cout << name << " ";
    }

    return 0;
}
```

7.3.3. Löschen von Elementen aus einer Liste:

Das Löschen von Elementen aus einer Liste kann mit den Methoden pop_back() und pop_front() erfolgen. pop_back() entfernt das letzte Element der Liste, während pop_front() das erste Element der Liste entfernt.

Beispiel:

```
#include <iostream>
#include <list>

int main() {
    std::list<int> myList = {10, 20, 30, 40, 50};

    myList.pop_back();
    myList.pop_front();

    for (int num : myList) {
        std::cout << num << " ";
    }

    return 0;
}
```

7.3.4. Suchen in einer Liste:

Um ein bestimmtes Element in einer Liste zu suchen, können wir eine Schleife mit einem Iterator verwenden, um durch die Liste zu iterieren und das gewünschte Element zu finden.

Beispiel:

```
#include <iostream>
#include <list>

int main() {
    std::list<std::string> myList = {"Alice", "Bob", "Charlie", "Alice"};
```

```
    // Suchen nach dem ersten Vorkommen von "Alice"
    for (std::list<std::string>::iterator it = myList.begin(); it != myList.end(); ++it)
{
        if (*it == "Alice") {
            std::cout << "Alice gefunden!" << std::endl;
            break;
        }
    }

    return 0;
}
```

Listen bieten eine effiziente Möglichkeit, Elemente hinzuzufügen und zu löschen, insbesondere wenn die Elemente häufig in der Mitte der Liste eingefügt oder entfernt werden müssen. Die doppelt verkettete Struktur der Liste ermöglicht es, auf die vorherigen und nächsten Elemente effizient zuzugreifen. Listen sind in vielen Szenarien nützlich, wie z.B. wenn Elemente in der Reihenfolge ihrer Einfügung gespeichert werden müssen und häufige Einfüge- und Löschoperationen erforderlich sind.

7.4. Arrays aus der STL (std::array)

Ein „std::array“ ist ein Container-Typ aus der Standard Template Library (STL) in C++, der eine feste Größe hat und zur Kompilierzeit dimensioniert wird. Es bietet eine moderne und sichere Alternative zu herkömmlichen C-Arrays, da es die Größe des Arrays während der Laufzeit überwacht und sicherstellt, dass keine Zugriffsverletzungen auftreten.

7.4.1. Erstellen eines std::array:

Um ein „std::array“ zu erstellen, müssen wir die -Header-Datei inkludieren und den Container-Typ std::array verwenden. Der Typ der Elemente, die im Array gespeichert werden, muss ebenfalls angegeben werden.

Beispiel:

```
#include <iostream>
#include <array>

int main() {
    std::array<int, 5> myArray = {1, 2, 3, 4, 5};
```

```
    for (int num : myArray) {
        std::cout << num << " ";
    }

    return 0;
}
```

In diesem Beispiel haben wir ein „std::array“ mit dem Namen „myArray“ erstellt, das fünf Ganzzahlen enthält.

7.4.2. Zugriff auf Elemente eines std::array:

Der Zugriff auf die Elemente eines „std::array“ erfolgt ähnlich wie bei C-Arrays über den Index. Beachten Sie jedoch, dass „std::array“ bei Verwendung des Index-Operators ([]) die Grenzen überprüft und einen Laufzeitfehler (Out-of-Bounds-Exception) auslöst, wenn versucht wird, auf ein ungültiges Element zuzugreifen.

Beispiel:

```
#include <iostream>
#include <array>

int main() {
    std::array<int, 5> myArray = {1, 2, 3, 4, 5};

    // Zugriff auf Elemente über den Index-Operator
    std::cout << myArray[0] << std::endl; // 1
    std::cout << myArray[3] << std::endl; // 4
    // std::cout << myArray[6] << std::endl; // Fehler: Index 6 liegt außerhalb des gültigen Bereichs

    return 0;
}
```

7.4.3. Größe eines std::array:

Die Größe eines „std::array“ kann zur Kompilierzeit mithilfe der Methode size() ermittelt werden. Da die Größe zur Kompilierzeit festgelegt wird, kann sie nicht während der Laufzeit geändert werden.

Beispiel:

```cpp
#include <iostream>
#include <array>

int main() {
    std::array<int, 5> myArray = {1, 2, 3, 4, 5};

    std::cout << "Größe des Arrays: " << myArray.size() << std::endl; // 5

    return 0;
}
```

„std::array“ bietet eine sicherere Alternative zu herkömmlichen C-Arrays, da es die Größe des Arrays zur Kompilierzeit kennt und sicherstellt, dass der Zugriff auf gültige Elemente erfolgt. Es ist in modernen C++-Programmen beliebt und wird empfohlen, wenn eine feste Größe und eine sicherere Datenstruktur erforderlich sind. „std::array“ kann in vielen Situationen verwendet werden, in denen die Größe des Arrays zur Kompilierzeit bekannt ist und keine dynamische Größenänderung erforderlich ist.

7.5. Sets in C++

Ein Set ist eine Container-Klasse in C++, die eine geordnete Sammlung einzigartiger Elemente enthält. In einem Set können keine Duplikate von Elementen vorkommen, und die Elemente werden in aufsteigender Reihenfolge gespeichert. Das Thema „Sets“ ist wichtig, um eindeutige Elemente zu speichern und effizient auf sie zuzugreifen.

7.5.1. Erstellen eines Sets:

Um ein Set in C++ zu verwenden, müssen wir die -Header-Datei inkludieren und den Container-Typ std::set verwenden. Der Typ der Elemente, die im Set gespeichert werden, muss ebenfalls angegeben werden.

Beispiel:

```cpp
#include <iostream>
#include <set>

int main() {
    std::set<int> mySet = {5, 2, 8, 3, 1};

    for (int num : mySet) {
```

```
        std::cout << num << " ";
    }

    return 0;
}
```

In diesem Beispiel haben wir ein Set mit dem Namen „mySet“ erstellt und mit einigen Ganzzahlen initialisiert.

7.5.2. Einfügen von Elementen in ein Set:

Das Einfügen von Elementen in ein Set kann mit der Methode insert() erfolgen. Da Sets keine Duplikate zulassen, wird ein bereits vorhandenes Element nicht erneut hinzugefügt.

Beispiel:

```
#include <iostream>
#include <set>

int main() {
    std::set<std::string> mySet;

    mySet.insert("Apple");
    mySet.insert("Banana");
    mySet.insert("Orange");
    mySet.insert("Apple"); // Wird ignoriert, da "Apple" bereits im Set vorhanden ist

    for (const std::string& fruit : mySet) {
        std::cout << fruit << " ";
    }

    return 0;
}
```

7.5.3. Löschen von Elementen aus einem Set:

Das Löschen von Elementen aus einem Set kann mit der Methode erase() erfolgen, indem das Element angegeben wird, das gelöscht werden soll.

Beispiel:

```
#include <iostream>
#include <set>
```

```
int main() {
    std::set<int> mySet = {1, 2, 3, 4, 5};

    mySet.erase(3); // Löscht das Element mit dem Wert 3

    for (int num : mySet) {
        std::cout << num << " ";
    }

    return 0;
}
```

7.5.4. Suchen in einem Set:

Um nach einem bestimmten Element in einem Set zu suchen, können wir die Methode find() verwenden. Diese Methode gibt einen Iterator zurück, der auf das gesuchte Element zeigt. Wenn das Element nicht gefunden wird, gibt find() den Iterator, der auf das Ende des Sets zeigt (mySet.end()), zurück.

Beispiel:

```
#include <iostream>
#include <set>

int main() {
    std::set<int> mySet = {1, 2, 3, 4, 5};

    std::set<int>::iterator it = mySet.find(3);
    if (it != mySet.end()) {
        std::cout << "Element gefunden: " << *it << std::endl;
    } else {
        std::cout << "Element nicht gefunden." << std::endl;
    }
    return 0;
    }
```

Sets sind nützlich, wenn eine geordnete Sammlung von eindeutigen Elementen benötigt wird. Sie eignen sich gut für Aufgaben wie das Entfernen von Duplikaten aus einer Liste von Elemen-

ten oder das schnelle Suchen nach bestimmten Werten. Die Verwendung eines Sets erfordert in der Regel ein gewisses Verständnis der Sortierung der Elemente und ihrer Einzigartigkeit. Sets sind eine wertvolle Ergänzung der C++-Standardbibliothek und bieten eine effiziente Implementierung, um eindeutige Daten zu verwalten.

Literaturverzeichnis