Typen und Variablen

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;
```

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.

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.
}
```

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.

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.

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.

Operatoren

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
```

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
```

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
```

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
```

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
```

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.

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.

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.

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.

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.

Zusammenfassung

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).

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

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;

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;

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
    }
```

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;
}
```

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;
```

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
```

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;
}
```

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;
    }
```

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;
    }
```

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++;
    }
```

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);
```

Fragen

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.

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.